CGiS:
High-Level Data-Parallel GPU Programming

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Diplom-Informatiker
Philipp Lucas

Saarbrücken
August 2007

Dekan:	Prof. Dr.-Ing. Thorsten Herfet
Prüfungsausschuß:	Prof. Dr. Raimund Seidel (Vorsitzender) Prof. Dr. Reinhard Wilhelm (Gutachter) Prof. Dr.-Ing. Philipp Slusallek (Gutachter) Dr. Bodo Manthey (akademischer Mitarbeiter)
Tag des Kolloqiums:	08. Januar 2008

Hiermit versichere ich an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, 27. 08. 2007

Abstract

In the last few years, PC technology underwent a paradigm shift. The current trend leads aways from raising sequential performance to enhancing the available parallelism. The rapid performance increase of Graphics Processing Units (*GPUs*) is a part of this trend. However, it is difficult to harness the computational potential because for the longest time GPUs could be directed only through graphics APIs and in low-level code. The language CGIS has been developed to remedy this situation. CGIS is a data-parallel programming language, which offers a high-level abstraction of GPUs, letting programmers use GPUs as co-processors for massively parallel algorithms. This work presents the language and the compiler for CGIS in the context of general purpose programming on GPUs (*GPGPU*).

Zusammenfassung

Seit einigen Jahren zeichnet sich bei handelsüblichen PCs ein Trend weg von der Erhöhung der sequentiellen Leistung hin zur Parallelverarbeitung ab. Ein Bestandteil dieses Trends ist die rasche Leistungsentwicklung der Grafikkarten (*GPUs*), deren Rechenleistung die aktueller CPUs mittlerweile übertrifft. Es ist jedoch schwierig, diese Leistung auch abzurufen, da diese Geräte lange Zeit nur hardwarenah und über Grafik-APIs ansteuerbar waren. Um dies zu ändern, ist CGIS entwickelt worden, eine datenparallele Programmiersprache, die die GPUs abstrahiert und ihre Benutzung als Co-Prozessoren für massiv-datenparallele Algorithmen ermöglicht. Diese Arbeit stellt die Sprache und den Compiler im Kontext dieser Entwicklung vor.

Extended Abstract

Recent years have witnessed a dramatic performance increase for ordinary PCs. The execution units achieve higher throughput because of an increased core frequency. To achieve a high degree of utilisation of these devices, CPUs employ more and more auxiliary units. Caches, branch prediction units and reordering logics take up large amounts of transistors in modern CPUs.

But this trend is reaching its limits. Instead of only enhancing the execution of a sequential instruction stream, the number of parallel execution units is increased. Such a development also takes place in another component of today's PCs, in the graphics hardware.

Current graphics processing units (*GPUs*) employ a multitude of parallel execution units. The original application area of GPUs is the creation of realistic images. The algorithm used for this purpose, *rasterisation*, is naturally parallel. Therefore, increasing the parallelism in execution units immediately translates to a higher performance for that purpose.

Besides increasing the parallelism in GPUs, another development has set in, motivated by the ever increasing demands of the content industry. In particular game development depends on more and more realistic image synthesis, incorporating multiple effects such as reflection, shadows or motions. To this end, GPUs have evolved from an implementation of a particular algorithm which could only be configured by changing particular parameters into fully programmable processors. This ability is exploitable also for more general applications (*General Purpose Programming on GPUs; GPGPU*). And indeed, successes have been achieved even on early models of programmable graphics hardware: GPUs can be used as co-processors to relieve the CPU of some computational tasks in parallel algorithms.

But programming GPUs is not an easy task, especially for a purpose for which they were not originally created. The hardware still is characterised by their legacy as a device to compute images. For instance, GPU programs are severly restricted in memory accesses, and the control flow capabilities were limited for a long time as well. But GPUs are still suited very well for *streaming* algorithms, which read from one stream, perform the same computation on all elements of that stream, and write into another stream. As it turns out, surprisingly many algorithms can be cast favourably into this programming model.

The second obstacle for a widespread usage of GPUs as numerical co-processors lies in inadequate abstraction. In principle, GPUs can be programmed only through graphics APIs (OpenGL, DirectX). The programming model of these APIs does not form a natural abstraction for general purpose algorithms. The programmer is forced to express a data-parallel algorithm in terms of the graphics paradigm, that is, in terms of colourings of objects. Therefore, to really support GPGPU for non-specialist programmers,

programming languages have to achieve a high level of abstraction, completely relieving the programmer from interacting with the graphics APIs.

To this end, several languages have been developed in the last few years. But as it happens, most of these languages do not achieve both a high degree of abstraction and a high resemblance to traditional programming languages. Therefore, in a concurrent development, the programming language CGIS has been developed.

CGIS is an imperative, data-parallel programming language, offering a common abstraction for GPUs and SIMD capable CPUs. CGIS enables the programmer to express a massively parallel algorithm naturally as a sequence of parallel executions of sequential programs (also called *kernels*). This allows outsourcing a parallel part of an application onto the GPU. The CGIS compiler processes the program code, outputting assembly code for the GPU programs and C++ code to interact with the GPU through OPENGL. The programmer interfaces with this generated code through a few easy API functions for data exchange and for directing the execution of the co-program. It remains completely hidden that the code is executed on the GPU. In fact, in the very same way can CGIS be compiled into SIMD code, and that code is used in the very same way.

To achieve this, the CGIS compiler faces the common tasks of any compiler translating from a high-level language into a low-level hardware language. However, differences arise due to the peculiarities of the language and the target. On the positive side, the compiler does not need extract the parallelism from the program text, as compilers for traditional languages have to do: The parallelism is already inherent in the expression of the algorithm as a CGIS program. Other differences result in the presence of vectorial types of small width both in the language and in the hardware. This raises new optimisation opportunities and new pitfalls in compilation. Also, the restricted memory access capabilities enforce an abstraction, which in turn necessitates a mechanism for automatic data reordering.

By means of several applications, it was shown that CGIS does indeed enable applications to easily use GPUs as co-processors. It has been shown to offer a performance increase for parallel algorithms with respect to modern CPUs. Therefore, CGIS may serve to spread the usage of GPGPU and thereby enable programmers to harness the power of GPUs for different purposes than computer games. It may also serve as one point in the development of the grand unifying parallel programming language of the future which is needed to uniformly use the emerging multi-core architectures.

This work starts with presenting the state of the art in programmable graphics hardware. Afterwards, it presents various competing approaches for programming GPUs. In detail, it defines the programming language CGIS and argues for its design. Then, it presents the compiler and the runtime system. To demonstrate the usability of CGIS, it lastly shows a variety of applications.

Erweiterte Zusammenfassung

Durch den technologischen Fortschritt hat die Leistung der CPUs von handelsüblichen PCs in den letzten Jahren einen rasanten Aufschwung genommen. Die Erhöhung der Taktfrequenz bedeutet einen erhöhten Durchsatz der Rechenwerke, und immer größere Teile einer CPU werden von Zwischenspeichern, Sprungvorhersageeinheiten oder Umordnungslogik eingenommen, um die Auslastung der sequentiellen Einheiten zu erhöhen.

Doch die Verbesserung der sequentiellen Leistung von Prozessoren stößt an ihre Grenzen. Mehr und mehr werden Fortschritte der Leistungsfähigkeit nicht durch Erhöhung der Geschwindigkeit bei der Abarbeitung eines sequentiellen Instruktionsstromes erzielt, sondern durch Erhöhung der Anzahl der gleichzeitig eingesetzten Ausführungseinheiten. Diese Entwicklung findet auch in einem anderen Teil heutiger PCs statt, nämlich bei den Grafikkarten.

Heutige Grafikkarten (*Graphics Processing Units, GPUs*) verfügen über eine Vielzahl von parallel einsetzbaren Ausführungseinheiten. Der eigentliche Einsatzbereich von GPUs ist die Erzeugung realistisch wirkender Bilder auf dem Bildschirm. Da der hierzu verwendete Algorithmus (*Rasterisierung*) von Grund auf parallel ist, konnte durch eine Vervielfachung der Ausführungseinheiten sofort auch eine Erhöhung der Leistung erzielt werden.

Die rasch steigenden Anforderungen insbesondere der Computerspiele-Industrie nach noch realistischeren Darstellungen mit speziellen Effekten wie Spiegelung, Schattenwurf und Bewegungen führte zur Einführung der GPU-Programmierbarkeit. Anstatt nur Konfigurationsparameter für einen vorgebenenen Algorithmus anbieten zu können, sind heutige GPUs in der Lage, ganze Programme auszuführen. Es ist diese Programmierbarkeit, die auch für allgemeinere Anwendungen genutzt werden kann. Diese Nutzung wird als GPGPU (*General Purpose GPU Programming*; allgemeine Programmierung von GPUs) bezeichnet. In der Tat sind bereits früh vielversprechende Resultate erzielt worden, die belegten, daß GPUs nicht nur theoretisch, sondern auch in praktischen Anwendungen eine höhere Leistung als CPUs erzielen können. GPUs können als Koprozessoren für parallele Algorithmen Anwendung finden und die CPU entlasten.

Doch GPU-Programmierung ist keine leichte Aufgabe. Dies liegt an der Hardware selbst und an den Abstrakionsmechanismen. Die Hardware ist durch Anpassung an ihre primäre Nutzungsart gekennzeichnet, die Verwendung zur Bilderzeugung. Daher ist der Zugriff auf globalen Speicher für GPU-Programme stark eingeschränkt, was in der Programmierung ebenso Schwierigkeiten verursacht wie die lange Zeit massiv eingeschränkten Kontrollflußoperationen. Tatsächlich eignen sich GPUs insbesondere für *Streaming*-Algorithmen, die aus einem Datenstrom lesen, dieselbe Operation auf alle Elemente anwenden und in einen anderen Strom hineinschreiben. Eine Vielzahl von Algorithmen läßt sich in diesem Programmiermodell sinnvoll und effizient ausdrücken.

Das zweite Hindernis zur allgemeinen Nutzung von GPUs als Koprozessoren besteht in der ungenügenden Abstraktion. GPUs sind prinzipiell nur über Grafik-APIs (OPENGL oder DIRECTX) ansprechbar; dies ist keine natürliche Abstraktion für allgemeine Anwendungen und zwingt die Programmierer dazu, numerische Algorithmen in das Grafik-Programmiermodell zu zu überführen, also als Berechnungen der Farben virtueller Objekte auszudrücken. Um GPGPU einem weiteren Personenkreis zugänglich zu machen, müssen geeignete Programmiersprachen vollkommen von der eigentlichen Hardware abstrahieren und den Programmierer von der Beschäftigung mit den grafikbasierten APIs befreien.

Zu diesem Zwecke sind in den letzten Jahren einige Sprachen entwickelt worden. Doch diese Sprachen können die Anforderungen an Abstraktion der Hardware und gleichzeitige Ähnlichkeit zu herkömmlichen Programmiersprachen nicht voll erfüllen. Daher ist die Programmiersprache CGIS entwickelt worden.

CGIS ist eine imperative, datenparallele Programmiersprache, die GPUs und die SIMD-Einheiten von CPUs auf gemeinsame Weise abstrahiert. CGIS ermöglicht es dem Programmierer, einen massiv-parallelen Algorithmus in einer natürlichen Form als Folge von parallelen Ausführungen sequentieller Prozeduren (*Kernel*) auszudrücken. Somit kann die Funktionalität eines parallelen Teiles einer Anwendung auf die GPU ausgelagert werden. Der CGIS Compiler übersetzt den Programmcode dann in Assembler-Code für die GPU und in C++-Code zur Steuerung der GPU über OPENGL und zur Ankopplung an die Anwendung. Der Programmier interagiert mit diesem Code nur über API-Funktionen, die zum Datenaustausch und zur Steuerung der Berechnungen dienen: Die GPU selbst bleibt unsichtbar. Auf genau gleiche Weise kann das CGIS-Programm auch für SIMD-fähige CPUs übersetzt und auf diesen ausgeführt werden.

Der CGIS-Compiler muss hierzu die üblichen Arbeiten eines Übersetzers von einer Hochsprache in eine hardwarenahe Sprache ausführen. Es ergeben sich jedoch in der Übersetzung auch einige Unterschiede zu normalen Compilern, sowohl aus der Sprache als auch aus der Zielarchitektur. Im Gegensatz zu herkömmlichen, sequentiellen Programmiersprachen, die auf dem traditionellen Modell eines einzelnen Instruktionsstromes beruhen, ist es nicht nötig, aus CGIS-Programmen den möglichen Parallelismus mühsam zu extrahieren: Er ist bereits in der Darstellung des Algorithmus als CGIS-Programm vorhanden. Der SIMD-Parallelismus, der durch die Existenz von vektoriellen Datentypen niedriger Länge zusätzlich zum Stream-Parallelismus vorhanden ist, eröffnet weitere Möglichkeiten zur Optimierung und stellt weitere Herausforderungen in der Übersetzung. Weitere Unterschiede ergeben sich durch die Einschränkung von GPUs auf nur lesbaren und nur beschreibbaren Speicher, die einen Mechanismus zur automatischen Datenumordnung erfordert.

Anhand einiger Anwendungen konnte gezeigt werden, daß CGIS in der Tat die Benutzung von GPUs als Koprozessoren zur Leistungssteigerung bei parallelem Code gegenüber modernen CPUs ermöglicht. Dadurch kann CGIS als Ansatzpunkt dienen, GPGPU eine weitere Verbreitung zu ermöglichen und somit die Leistungsfähigkeit von GPUs für andere Anwendung als Spiele abzurufen. CGIS kann auch die zukünftigen Entwicklungen von Programmiersprachen für die neuen Multicore-CPUs beeinflussen, für die eine möglichst allgemeingültige Abstraktion notwendig ist.

In dieser Arbeit wird der Stand der Technik in bezug auf GPUs und ihre Programmierung dargelegt. Anschließend wird die Programmiersprache CGIS definiert und es werden die Designentscheidungen begründet. Danach werden der Compiler und das Laufzeitsystem für die GPU-Zielarchitekturen beschrieben. Zur Demonstration von CGIS werden abschließend mehrere Applikationen getestet.

Overview of Contents

Table of Contents

List of Figures

Image attributions: Figure II.1 is taken from [D25]. Figure II.5 was produced by Mayang Murni Adnin, `http://mayang.com/textures/`. Figure II.6 was produced by Paul Baker, `http://www.paulsprojects.net`. Figure II.11 is taken from Microprocessor Report, January 19, 2005. Figure II.12 is adapted from [N07a]. Figure VI.6 is taken from Kodak's sample image set.

List of Programs

List of Tables

Acknowledgements

During the course of the CGIS project, I profited from the influences of or direct cooperation with various people. *Reinhard Wilhelm* gave me the opportunity to choose my area of research and gave me enough leeway to conduct it on my own, as did *Philipp Slusallek*; thanks to them for giving me this freedom. My partner in crime during the ever-changing CGIS project was *Nico Fritz*; many thanks to him for patiently listening to my lamentations about OPENGL, driver issues and the world in general, for talking me out of some of my weirdest ideas, and for just being Nico. I shared my office with Comrade *Oleg Parshin* for the past few years; thanks for contributing much to a pleasurable time, for the traditional hot chocolate breaks and for being a good office mate in general. Thanks also to *Gernot Gebhard*, who wrote early versions of some parts of the compiler and handled the build system; thanks for that, for letting his 8600 serve a noble goal, and for always being open to change OORS to my needs.

I was supported by a grant of the *Deutsche Forschungsgemeinschaft* during a part of my work, and I am indepted to them for that.

Several people were of great help in the composition of this thesis. *Eike Lang* read through the entire manuscript and saved the reader from a dangerous combination of all too archaic phrasings and postmodern colloquialisms. *Fritz Müller* scrutinised the description of CGIS, uncovering gaps which had to be filled.

I also thank my parents, *Manfred* and *Ele*, who gave me the freedom to decide which way I wanted to go in life, and without whom none of this would have been possible.

1

Introduction

"Who are *you*?" said the Caterpillar.
That was not an encouraging
opening for a conversation.
L. CARROLL, *Alice's Adventures in Wonderland*, 1865

1.1 The CGiS Project

When a task is to be performed by a worker and the time to complete the task is deemed unacceptably long, two methods present itself: Working faster, or spreading the task on more shoulders. The evolution of PCs and their CPUs have followed the first model for the longest time, but that model is rapidly facing fundamental difficulties. To keep the execution units busy, modern chips employ ever larger caches to avoid accesses to the slow main memory, and complicated logic for instruction reordering, speculative executions and branch prediction tries to ensure keeping the execution units busy. Current CPUs use a large amount of their transistors to enable a small amount of transistors to do the actual work. Energy dissipation forms an additional, hard limit on the raw speed of the processor.

Thus, concentrating on increasing the performance of a single processor on a sequential instruction stream is slowly being neglected in favour of more parallelism [HP03]. This has started on a low level with limited instruction level parallelism, has expanded to narrow SIMD parallelism and is now including parallelism on the processor level even in commodity PCs. Two-core products of Intel or AMD are standard for modern PCs, with both vendors going for four-core systems. The Cell processor [IST05] offering eight processing units is used in the PlayStation 3 gaming console.

Another component of PCs is featuring parallelism at its very heart: The *Graphics Processing Units (GPU)*. GPUs were originally created for a quite mundane and uniform task: To transform descriptions of three-dimensional scenes into two-dimensional images. To this end, they implement a certain algorithm (*rasterisation*) in hardware. One key aspect of this algorithm is that the computations for different parts of an image are

independent of each other. That independence paved the way for a parallel implementation, and the most advanced GPUs indeed offer *hundreds* of execution units [A07b, N06].

But GPUs offer more than just a massively parallel implementation of a particular algorithm. Consumers demand more and more realistic images from computer games, and the gaming industry needs more power and flexibility to deliver that quality. To this end, several phases of the rasterisation algorithm have become replaceable by user-supplied programs. The expressibility of the instruction set as exposed to these programs has dramatically increased within the current decade: From simple register combinator stages to a processor with integer and floating point types, supporting all usual arithmetical instructions and dynamic control flow. This is exposed through standardised assembly languages and widespread graphics APIs.

This combination of constraints—CPUs facing their limits, GPUs offering parallelism and becoming programmable—gave rise to a completely new line of work: *GPGPU, General Purpose Programming on GPUs*. Various general algorithms have been implemented on GPUs, and GPUs could outperform CPUs on naturally parallel tasks. In these early applications, the GPUs have had to be programmed on a very low level. Vendor specific assembly languages or standardised assembly languages were the means to express the sequential parts of the computation inside of *kernels*, each working on one data element of a stream. The parallel computations of a multitude of concurrent kernels on streams of data were then expressed with standard graphics API functions.

But programming GPUs on a low level is a daunting task. As always, programming in assembly language is difficult, but GPUs pose the additional difficulty of having to cast an algorithm into an unnatural programming model. With only graphics APIs available to interact with GPUs, any computation has to be expressed in terms of the rasterisation algorithm. The programmer is forced on a detour in order to exploit the architecture's computational power. For this reason, *shading languages*, which allow the programmer to express the GPU kernel on a higher level, do not help much: They abstract away from the assembly language, but not from the graphics heritage, i. e., not from the fundamental assumptions and metaphors of the graphics programming model.

Therefore, specialised GPGPU languages have been developed. They have been met with great enthusiasm and form the base for a widespread success of GPGPU for non-graphics specialists. By completely abstracting the target and presenting only some kind of data-parallel hardware, they enable the programmer to concentrate on the algorithm itself.

The present languages, although providing a big step forward from the early days of GPGPU, may not present the definitive solution on the topic. Independently of these approaches, the CGiS project has been devised as to provide a language which offers an adequate abstraction of GPUs and other data-parallel architectures, hides the target from the programmer, and yet offers such a performance that it becomes a valuable tool for programmers seeking to outsource parallel computations to the GPU. CGiS seeks to make GPUs attractive targets as co-processors in numerical applications. This entails the following diffuse goals:

- CGiS shall be accessible to normal programmers.
- CGiS shall be executable on a wide range of targets.
- CGiS shall efficiently make use of the available resources.

We shall later (Section IV.1.a) break these goals down into smaller and more manageable and measurable objectives.

Emerged from this project has the language CGiS. Its description and rationalisation will form an important part of this thesis. CGiS is a data-parallel programming language which allows the programmer to implement a data-parallel algorithm in a natural way. It compiles down not only to GPU code, but also to SIMD code for modern CPUs, thus offering a unified abstraction for these two diverse targets. Expectedly, the different targets need quite different implementation strategies. This work is concerned only with the GPU implementation; a later thesis [F08] shall present the efforts needed to translate CGiS code into SIMD parallelism on CPUs.

The GPU compiler has many tasks to perform. Apart from the traditional transformations and optimisations which apply to GPUs in the same way as for traditional CPUs, both the source language and the target architecture offer special features which have to be catered for or may be exploited. A pertaining feature is the presence of narrow vectorial data-types as basic types and SIMD operations on them. These operations provide an additional parallelism within the stream parallelism. Because the architecture offers easy reordering of components and smooth switching between scalar and SIMD computations, and because the language exposes these features to the programmer and abstracts away from further differences for the sake of orthogonality, the compiler has to work with subunits of the basic types and adjust the algorithms accordingly.

Other peculiarities of the target concern the memory model. Because of their heritage as a hardware implementation of rasterisation, GPUs offer only a quite restricted memory model. To GPU programs, the memory is divided into non-overlapping regions of arrays of a primitive type with restrictions on readability or writeability. To offer an additional abstraction which, on the one hand, helps the programmer to implement an algorithm in a natural way, and, on the other hand, is still efficiently implementable, is a requirement the language design and the compiler implementation have to meet.

Now all of this work would be of very limited use if CGiS were not to meet the goal which was the very reason for its conception, namely to facilitate using GPUs as co-processors for naturally parallel algorithms. As it turns out, CGiS does meet this goal. Several data-parallel algorithms with a wide variety of characteristics have been implemented in CGiS. They result in impressive speed-ups of up to a factor of 50, reinforcing the usability of GPUs as co-processors.

All in all, CGiS is a language which can help the programmer to harness the power of graphics hardware for purposes other than just gaming. It can pave the way to a more widespread use of GPGPU, but it may also have a future for emerging parallel architectures of CPUs. Recent architectural developments lead into the direction of higher numbers of full processor cores, or to a merger of the GPU and CPU architectures, featuring yet larger number of smaller processing units also on CPUs. Processors such as the Cell processor or Niagara offer 8 cores with varying interdependence, and proposals merging CPU and GPU will lead to yet more parallelism on the CPU chips [IST05, S07b, I07, R07]. A unified model for programming the emerging multicore architectures is a prerequisite for harnessing their power. CGiS can serve as an example for a particular kind of abstraction; its ability to abstract away from the target and compile down to GPU code and SIMD code shows that a unified abstraction mechanism is possible, albeit for a restricted domain of naturally parallel, computationally dense algorithms.

I.2 Outline of the Work

The remaining parts of this treatise are organised as follows. Chapter II presents the target under consideration, the GPUs. We shall investigate their features and see how GPUs evolved from their small beginnings into the powerful devices they are today. That chapter also gives a short historical overview of GPGPU. Chapter III presents a multitude of languages for GPU programming. First, a distinction between different classes of languages is developed, and it is argued that a GPGPU language has to offer a certain level of abstraction. Then, the languages are described and their strengths and weaknesses are worked out. In the end, it shall transpire that these GPGPU languages are not the ideal tool for a widespread use of GPGPU. Chapter V then presents CGiS. It opens up with breaking down the fundamental goals of CGiS into a number of objectives pertaining to various parts of the CGiS system. This discussion is followed by a description of the syntax and semantics of CGiS, during which the text presents the rationale behind the decisions and argues for the language meeting its design goals. Chapter V presents the software components of the CGiS system. Among these, the compiler stands out as the tool for translating the high-level description of an algorithm into low-level graphics operations and assembly code. This translation needs a number of common and unique transformations and offers opportunities for common and unique optimisations as well. The runtime system directing the GPUs is also featured in this chapter. The chapter concludes with describing the other components of the CGiS system, both the supportive components of the main compiler and the GPU target, and the parts pertaining to other targets or written by other people. Chapter VI concludes the description of the CGiS system by showing it in action. We shall see a number of applications which have been implemented in CGiS. By way of comparison with optimised CPU implementations, we shall see the strengths of CGiS for naturally parallel algorithms and investigate the usability of CGiS for the stated purpose. As it will turn out, CGiS shows to be a fine tool for GPGPU. Chapter VII forms the dual to this chapter. It sums up the complete work and puts the achievements into the greater context, and it presents an outlook into the future, outlining future developments regarding CGiS and emerging technological trends.

II

Graphics Processing Units

The safest way, the straight and narrow
No confusion, no surprise
A. ELDRIGDE, *Alice*, 1982

II.1 GPU Architecture

Starting from the earliest electronic computing devices, a way to visually display data was implemented. Even as early as in the 1960s, such visualisation was aided by special hardware, to relieve the main computational devices of that burden. [MS68] is particularly interesting to our topic, for the authors report on a *special purpose display processor*, that grew step-by-step into a *general purpose computer* with special instructions for graphics. Back in these days, this meant that the hardware grew from a device which could load coordinates into registers and perform a DISPLAY operation, to a device capable of iterating over lists of coordinates, into a processor with jumps and subroutines. Myer and Sutherland argue that such an evolution leads to needlessly complicated devices which themselves need more specialised auxiliary processors (a "wheel of reincarnation"), and they hope that their experience "may speed others on toward 'Nirvana'" — an aim which has not been achieved, because a very similar development is happening just a few decades later in the consumer graphics market.

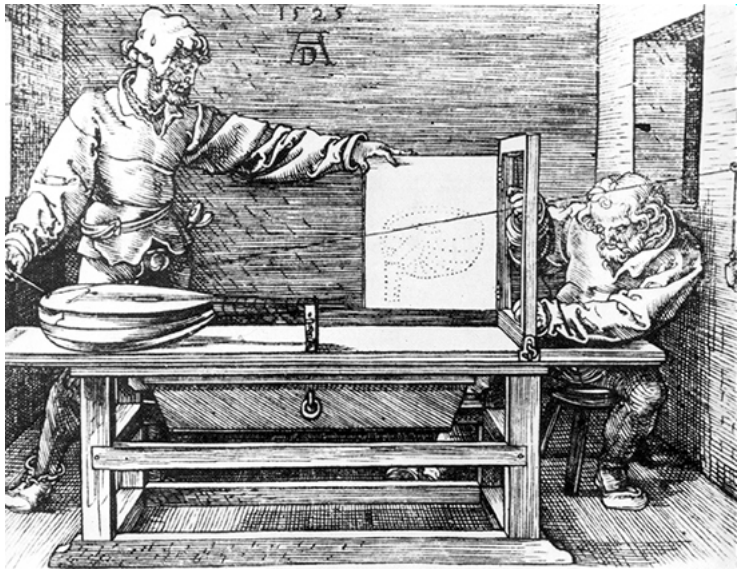
This chapter is devoted to explaining the current situation of consumer and professional hardware for displaying graphics. This section explains rasterisation, both as an algorithm and as the hardware implementation present in today's GPUs. Sections II.2 and II.3 explain how these GPUs are programmed. This is compared to CPUs in Section II.4. Section II.5 explains the likely development of GPUs in the near future. Section II.6 presents an overview of GPGPU, and Section II.7 sums up and concludes this chapter.

II.1.a Rasterisation

The purpose of graphics hardware can be summed up quite succinctly: *computing a two-dimensional view of a three-dimensional scene*. This entails identification of visibility and lighting with ad-hoc algorithms or simulations of physical properties. Various algorithms differ not only in the computations taking place, but also in the particular choices of input data.

The *rasterisation* algorithm takes as its input data a set of surfaces, specified by a geometry and defining points. A circle, for example, might be specified by its centre, its orientation in space and its radius; or just by its centre and two points on the diameter. A triangle may be specified by its three vertices. From a particular view point, these objects are then projected onto a two-dimensional plane, making up the final image (Figure II.1). Said image consists of *pixels*: equilateral, solidly coloured, two-dimensional building blocks of the image. By this projection it is also specified which pixels an object covers. This particular computation is responsible for the name *rasterisation*.

Figure II.1 Projection of a scene onto a plane (uncomputerised version)

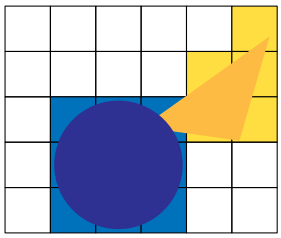


Because distinct points in the three-dimensional space might be projected onto the same points in the two-dimensional plane, and because differing points might fall into the same pixel[1], the algorithm also has to compute the occlusion of objects. This is done by regarding as visible a part of that object, of which the point projected onto the pixel lies nearest to the view point (Figure II.2).

To compute the complete image, the colours of pixels have to be computed from the specified colours of objects. This process is known as *shading*. In general, the colour of a point of an object's surface is interpolated in some way from colours specified on some selected points of a surface and by projecting precomputed images (textures) onto the object (see Section II.1.b for an example). This shading phase is independent from the algorithm used to compute the visible points of objects. If visibility is computed by another algorithm, such as ray tracing, a subsequent shading step still has to be performed.

[1] Recall that pixels are two-dimensional, in contrast to ideal points.

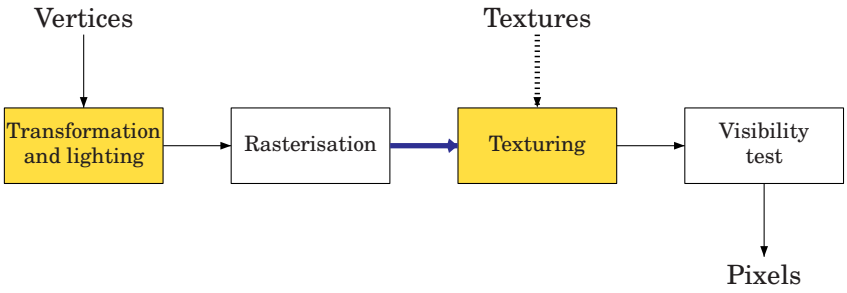
Figure II.2 Covering and occlusion for geometric primitives



II.1.b Rasterisation Hardware

Since about 1999 (the NV10 architecture), graphics hardware has implemented the full rasterisation algorithm including the coordinate transformations and lighting (*T&L*) in specialised processing units. Figure II.3 gives a high-level overview of the *graphics pipeline* employed in such devices.

Figure II.3 The rasterisation pipeline (fixed function)



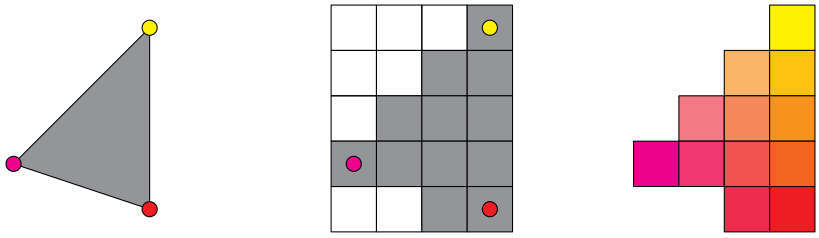
As input, the hardware receives the position of the vertices of triangles making up the surface of the objects in the scene. These points are then projected onto a two-dimensional plane: This phase is called *transformation*. Also, a basic lighting algorithm might be run, which presupposes that light from a certain light source reaches an object. The rasterisation phase then computes for each object the pixels which it covers. For each pixel, a *fragment* is generated, which is then passed on to the further parts of the pipeline. A fragment can be thought of as a pixel in the making. This fragment receives *interpolated values* from the vertex inputs; for example, the coordinates of the three corners of a triangle are interpolated to make up the position values of the fragments. The pipeline part upto the rasterisation is called the *vertex pipeline*, the remaining part is the *fragment pipeline*.

The fragment pipeline is responsible for the shading. To this end, colours of pixels are computed with a shading model such as Gouraud or Phong Shading [G71, P75, SAGMRSTW05], and *textures* are applied to the object. A texture is a small picture which gets spanned over the triangle. Figure II.4 gives an example of shading without texturing, Figure II.5 explains texturing. The input attributes of the vertices (colours or points on a texture image) are interpolated for the points of the objects and processed with colouring or texturing. The end result is a specification of colour values for each

pixel, divided into its red, green and blue constituents and an alpha value which can be used for transparency calculations.

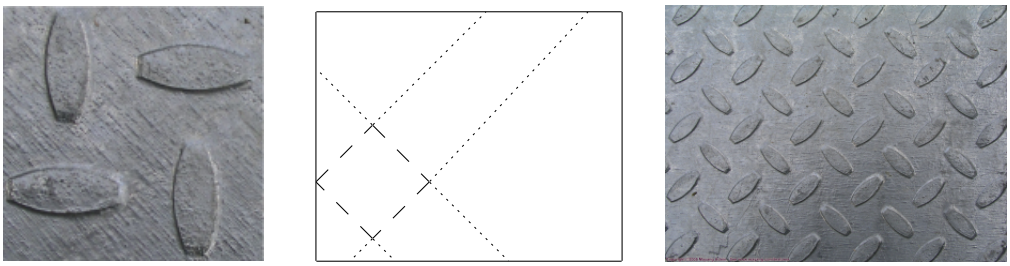
To sum it up, the graphics pipeline receives as inputs sets of three-tuples of point coordinates and attributes, which give rise to some number n of interpolated input-data, which are computed upon, leading to m four-component colour values ($m \leqslant n$, for a fragment may be occluded). The pixels are then stored in the *framebuffer*, which in traditional applications is presented in a screen window.

Figure II.4 Rasterisation and shading of a triangle



(a) Input triangle (b) Fragments (c) Output pixels

Figure II.5 Texturing



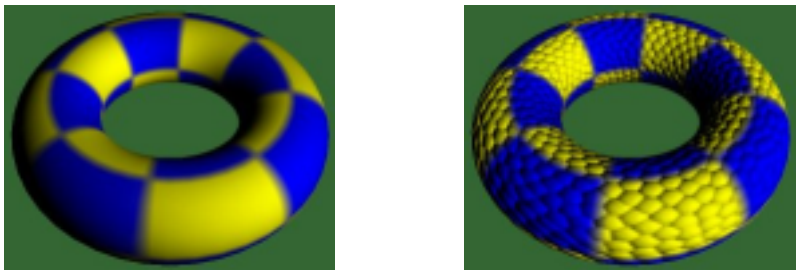
(a) Texture (b) Surface (c) Result

In the last decade, a development set in which resulted in more complicated functionalities implemented in the graphics pipeline. In particular, artists required more and more intricate algorithms for the interaction between texturing and basic object shading, and the graphics hardware was required to employ more complicated algorithms. For example, *bump mapping* means using data stored in a texture not directly as an image, but to interpret the values as *normals* of the points of the objects; these normals then govern the colours applied to the object from a light source. With this trick, one can easily create an illusion of higher scene complexity (Figure II.6).

In the end, the hardware and software were required to support a lot of closely related, but differing models of texturing, and graphics APIs became cumbersome to use, offering lots of similar, configurable functions for those tasks. Various versions of register combiners have been specified (for example, [N00]), which combine inputs such as colours, normals and texture components arithmetically to produce the final output. With the increasing power and complexity, a more traditional and more general way of specifying algorithms became desirable.

Thus set in a development to make the hardware *programmable*. Instead of choosing among a restricted set of algorithms and selecting values of certain parameters, or choos-

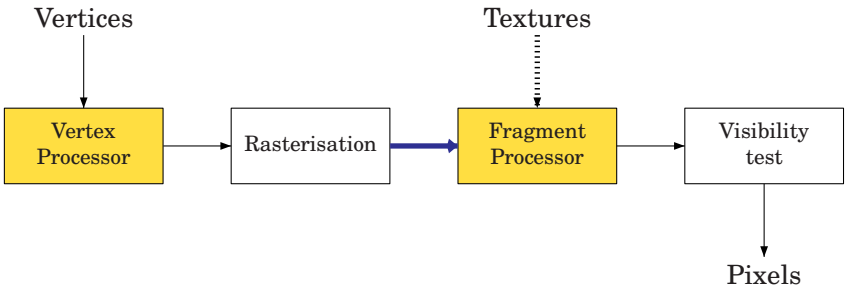
Figure II.6 Bump mapping



(a) Torus with shading (b) Torus with shading and bump mapping

ing inputs and operators of certain combination stages, programmers could now simulate desired effects using programmable processors. Figure II.7 gives an overview of this programmable rasterisation pipeline; compare with Figure II.3. For example, the texturing part is now replaced by a programmable fragment processor, which operates on fragments and computes pixel colours using textures as *arbitrary* additional input.

Figure II.7 The rasterisation pipeline (programmable)



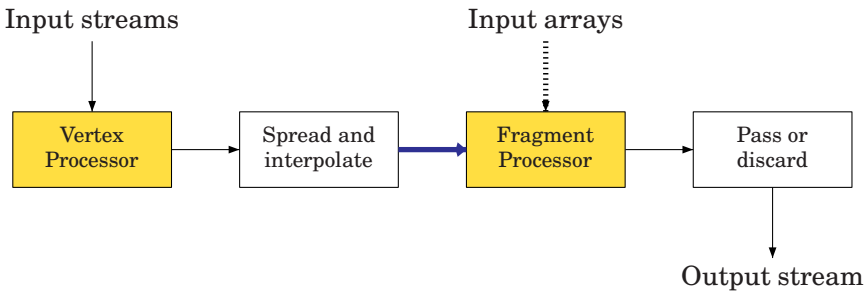
It is exactly this programmable pipeline which can be used for more general purpose programming. In this view, the pipeline gets sets of three-tuples of various four-component inputs, which give rise to a set of interpolated four-component intermediate values, which get computed upon, leading to a set of four-component output values (Figure II.8). The textures become simple random-access read-only arrays. Thus will we regard a GPU in the remaining part of this document: *Some device to compute upon streams of four-component values and constant arrays*.

The important point about this view is the mention of *streams*, and the underlying notion of parallelism. This is the main power exhibited by GPUs, and the next section investigates this aspect.

II.1.c Technology

The performance of GPUs follows from the rasterisation algorithm. To execute that algorithm efficiently, GPUs need only quite simple execution units, but they can make use of a lot of them by computing on various fragments in parallel. Therefore, GPUs employ a *large number* of *simple* processing units.

Figure II.8 The rasterisation pipeline (as a general purpose device)



Common GPUs now feature several dozen SIMD (4-component) vector ALUs. For example, the ATI Radeon X1950 has up to 48 pixel shader processors [A07c] and 8 vertex shader processors; the NVIDIA GeForce 8800 features up to 128 single-float ALUs which are dynamically dispatched among the vertex and pixel processors and among the vector components, as are the 320 processing units of the Radeon HD 2900 [A07b, N06].

Although the vertex processor is also programmable, it is not traditionally used for general purpose computations. In the early days of GPGPU programming, the vertex processor was the only part capable of working on floating point numbers (Section II.6.a, [THO02]). Some other features such as loops and conditionals also appeared first in the vertex processor. However, subsequent advances down the graphics pipeline have made GPGPU on the fragment processor possible, and therefore, GPGPU programs ran exclusively on the fragment processor[2], because this device offered a greater degree of parallelism. Additionally, until very recently, the vertex processor had no access to the texture memory, in which the main chunks of data reside. With the most recent technological advance of *unified* shading units which can be dynamically dispatched to any abstract processing units, the processors are very similar at their cores: Their different usages amount to additional, specific capabilities in the assembly languages on top of the large, common set of capabilities.

The core speed of GPUs is less than that of contemporary CPUs (see also Section VI.1.) Memory access is faster, however, because the electronic parameters are in favour of the engineers: The position of all memory chips is predetermined, and the wire lines are much shorter than on CPU motherboards. Thus, the interface to the memory chips can be driven at a higher speed. Together with a higher bus width of 256 Bit or even 384 Bit, and specially selected memory chips capable of higher core speed, this amounts to a bandwidth of more than 50 GByte/s even for middle-class GPUs, and more than 80 GByte/s for the top products [N06]. In contrast, even dual-channel DDR2-1066 memory offers no more than 17 GByte/s.

[2]Small parts have to be run on the vertex processor to set-up indices of streams and compute other interpolatable data. This vanishes compared to the computational effort spent in the fragment programs.

II.2 Use of GPUs

The vendors *ATI* and *NVIDIA* will often be mentioned in this text. This section describes the current state of availability of GPUs of various vendors, their standardisation and the history of their development as far as necessary to understand the following sections. Plans, projections and speculations about the future shall be dealt with in Section II.5.

II.2.a Vendors

The world's largest manufacturer of GPUs is Intel [P07]. Intel sells so-called *embedded (integrated) graphics devices*: GPUs which are embedded into the mainboard chipset responsible for I/O operations, such as communicating to the harddisks, the memory devices or to add-on cards. Such devices are also manufactured by other vendors, such as NVIDIA, ATI, VIA and SiS. These integrated devices are still sufficient for applications which are not extremely power-hungry; indeed, their practicability for day-to-day business work is exactly what makes Intel so successful in the market.

But at any point in time, these embedded graphics devices are lacking in power compared to contemporary *dedicated (discrete) GPUs*, which are plugged in to the standard expansion slots of PCs. These dedicated GPUs are the devices driving forward the innovation, and it is these GPUs which are used by gamers as well as users of professional CAD or modelling applications. Indeed, GPUs for audiences seemingly so diverse as gamers desiring higher frame-rates in first-person shooter games and architects longing for a detailed and fluent presentation of their newest designs, are not so much different on the hardware level. With the lines of so-called professional graphics cards *FireGL* and *Quadro* produced by ATI and NVIDIA, the main difference between these versions of the hardware and their gamer oriented counterparts lies in the software drivers. Whereas it might be negligible for a computer game when a few trees in an outdoor scene are displayed wrongly, this is unacceptable for a landscape architect. Thus, drivers are specially certified for specific software (versions). Also, the memory on professional GPUs is sometimes larger or faster than on the consumer hardware.

All in all, standard dedicated GPUs are the main target for GPGPU in this work. It is possible that applications would achieve a higher performance on professional GPUs, but only for a tremendous price difference. It is certain that embedded GPUs are lacking in performance compared to dedicated GPUs, but there is no fundamental difference between those two kinds of GPUs: Embedded GPUs are mainly just equivalent to a previous step on the evolutionary ladder of GPUs.

In other words, capabilities of a current *generation* of dedicated GPUs might be implemented in the next generation of embedded GPUs. Sorting the multitude of chips manufactured by the vendors with respect to their features into a few categories, or generations, of chips, is common. Section II.2.b deals with identifying the hardware by compatibility to specific software, yet of course the vendors themselves use generations as codenames for a range of chips, just as is the case for CPUs. For example, the chips of the NVIDIA's GeForce 7x00 family have codenames in the range G7*x* (G70 for the original 7800, G71 for the more advanced 7900, G72 for the lower-endian 7300,...), whereas the GeForce 6x00 family has codenames in the range NV4*x*. Such family names are often used as placeholders to denote capabilities of hardware, such as *"NV40 cards offer control flow in the fragment processor"*.

A note about nomenclature. Henceforth, the name *GPU* denotes the processing device of the graphics hardware together with any memory associated with it. The name *CPU* is used in the same way for the rest of the computer system. If the emphasis is on

the use of a GPU as a co-processor in a system, the term *host* is also used for the non-GPU part of a system. Also, instead of *texture* or *framebuffer*, the terms *input buffer* or *output buffer* are used, except for when the actual graphics meaning is in the foreground.

II.2.b APIs

Ensuring compatibility to existing software is a crucial step in designing new hardware. Although gamers are quite willing to spend money on newer and faster graphics cards to speed up their games, the market penetration of GPUs relies on being principally compatible to older software and to a wide range of games. Compatibility has two aspects:

- compatibility to previous generations of the same hardware family
- compatibility to other hardware families

Practically, both aspects are satisfied by a single notion of compatibility, namely *compatibility to programming interfaces*. These interfaces, the two APIs used to program GPUs, are Microsoft's DIRECTX[3] [M07a] and the committee-maintained OPENGL [SA06].

DIRECTX

DIRECTX is a monolithic standard by Microsoft solely for implementation in its Windows operating system, both on PCs and on gaming consoles[4]. The ubiquity of Windows makes DIRECTX the prime specification used for comparisons of capability; both the main DIRECTX versions and the versions of the programming languages for GPUs (*Vertex Shader* and *Pixel Shader* in Microsoft terminology; the term shader comes from the graphics heritage, see Section II.1.a). Table II.1 gives an overview of the versions; for example, a GPU might be called a "DX9-GPU" or to "support PS 2.0" to signify its capabilites, without precluding the GPU's use under OPENGL or in another operating system.

Table II.1 DIRECTX and Pixel Shader versions

DIRECTX	PS
8.0	1.1
8.1	1.4
9.0	2.0
9.0c	3.0
10.0	4.0

OPENGL

OPENGL strives to be platform independent, i. e., correct OPENGL programs should run on a wide variety of operating systems and hardware; if the platform has the necessary capabilities, of course. Furthermore, it is not controlled by a single software vendor, but by a conglomerate of software and hardware vendors. Formerly known as *Architectural Review Board (ARB)*, in September 2006 it became a working group under the hood of the *Khronos Group*. Khronos is focused on the development of free APIs for dynamic media creation and acceleration in general.

[3] Specifically, we are talking about DIRECT3D,
[4] The 'X' in DIRECTX gave rise to the 'X' in Xbox.

In any case, the members of the OPENGL governing body support or need vastly different characteristics of GPUs. Because of its very nature, the OPENGL standardisation focuses on an easily specifiable and easily usable *extensions* mechanism [K07]. These extensions are written and published by single vendors or by sets of vendors. GPU vendors are then free to implement in their hardware and their software drivers facilities to support those extensions and to advertise them to the outside. For example, there are several NVIDIA extensions of register combiners, various ATI and NVIDIA extensions for fragment programs, common extensions for programmability [O02a, O02b], and new extensions for NVIDIA's newest hardware generations [N07b, N07c].

Usage for GPGPU

Of course, GPGPU applications which should run on a non-Windows platform must use OPENGL. On Windows systems, programmers have a choice between DIRECTX and OPENGL, which can be made based on personal familiarity with the APIs, available documentation and need for integration with existing software. But even on Windows, the extension mechanism of OPENGL offers an incentive to use that API: Advances in hardware are exposed to the programmer sometimes much earlier than under DIRECTX. For example, the advanced programmability features of NVIDIA's G80 generation, the first DIRECTX 10 hardware, have been accessible starting with the advent of the GeForce 8800 in Autumn 2006; yet DIRECTX 10 itself is available only on Windows Vista, which was released later than the hardware. With the OPENGL extensions, on the other hand, programmers could use the new features as early as the hardware arrived.

The API choice made by the authors of specific applications or languages will be mentioned in the relevant sections (Section II.6 and Chapter III). CGIS targets OPENGL (core and extensions) because of the platform compatibility.

II.3 Programming GPUs

The sheer number of ALUs is as indicative of the usefulness of an architecture as is the core frequency of the processing units – a bit, but not much. Usefulness to a programmer translates to power of the programming environment (programming language and runtime environment), and that, in turn, is dependent on the fundamental capabilities of the underlying hardware.

This section describes the hardware capabilities as the foundation of the description of programming languages in Chapter III. It is divided in sections on the computational capabilities and on the memory subsystem.

II.3.a Instruction Set

When evaluating an instruction set, it is useful to divide it into *data manipulation* instructions and *control flow* instructions. We shall see that GPUs are limited in both areas compared to CPUs, yet more so for the control flow instructions. This section further investigates other peculiarities of data manipulation instructions to explain to the reader in more detail the state of the art, and to make easier the comparison with CPUs in Section II.4. The syntax and terminology follow the OPENGL extensions [N07b, N07c].

Arithmetical Instructions

Apart from instructions which fetch data from the input buffers or write data into the output buffers, all instructions work solely on registers. General registers are four-component single-precision floating point registers. The components of these registers are labelled either 'x', 'y', 'z', 'w'; or 'r', 'g', 'b', 'a'.[5] ***Notational convention:*** For the rest of this work, a notation of the kind `[type][n]` always denotes an n-component vector type with components of type *type*, e. g., `float3` is a vector of three `float` components.

Many arithmetical instructions work componentwise on those registers. For example, the instruction `ADD r1, r2, r3;` adds the contents of the components of registers `r2` and `r3` and writes them into `r1`. Some complicated instructions take only one argument, which then has to be selected[6]: `SIN r1, r2.x;` computes the sine of `r2`'s `x`-component and writes it into all components of `r1`. GPUs feature also some vector-arithmetical instructions, such as instructions computing the cross product or the inner product of vectors.

To support the shading computations which to implement GPUs were created, the processors feature a few instructions which are uncommon on CPUs. As a prime example of the specificalities, consider the `LIT` instruction. As its input, it receives a four-vector $[x, y, \_, w]$ (the third component is unused), and it computes the output $[1.0, x, y^w, 1.0]$. For GPGPU purposes, this instruction is rather unusable. On the other hand, an instruction designed to interpolate the colours of a point on a vector between two coloured points is very useful in practice: The `LRP` instruction, fed with inputs λ, a, b, computes $\lambda a + (1-\lambda)b$. This instruction is expedient to implement guarded assignments, as will be shown in Section V.3.b.

Masking and Swizzling

Masking and swizzling are two modifications to instructions which take advantage of the vectorial capabilities of GPUs. These features form one of the advantages of GPUs' instruction sets with respect to SIMD CPUs.

Masking entails precluding certain components of a vectorial result from being written into of a target register. For example, the instruction `MUL r1, r2, r3;` performs a componentwise multiplication of the registers `r2` and `r3` and writes the result into the respective components of `r1`. `MUL r1.xz, r2, r3;` performs the same computation, yet updates only the `x` and `z` components of `r1`. The contents of the `y` and `w` components are unchanged. Masking is supported on all applicable GPU operations.

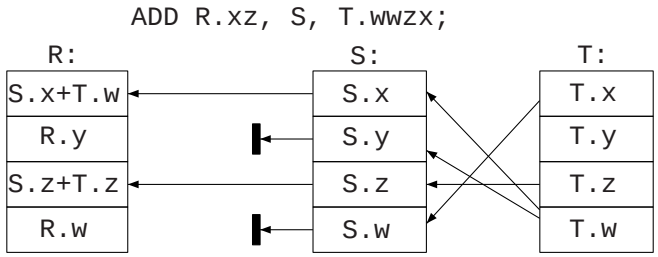
Swizzling specifies a reordering with possible replication of input registers. Compared with the instruction `MUL r1, r2, r3;`, the instruction `MUL r1, r2, r3.wzyx;` uses the components of `r3` in reverse order, yet lets invariant the actual contents of `r3`. Specification of a single component is a necessary *selection* for scalar operations, but a *replication* of that component on vectorial operations. For example, `SCS r1, r2.z;` updates `r1` to $[\cos(z), \sin(z), \perp, \perp]$ for z the `z`-component of `r2`: the `SCS` operation works only on a single scalar value. On the other hand, `ADD r1, r2.y, r3;` is just a shorthand notation for `ADD r1, r2.yyyy, r3;`, because `ADD` expects vectorial operands. Swizzling and replication, too, are supported on all applicable GPU operations.

[5] This shows the heritage of the processors: The `xyzw`-notation denotes homogenous coordinates of points in space, and the `rgba`-notation denotes the three colour components red, green and blue and an α-component used for transparency calculations. Some languages also offer the component specification 's', 't', 'p', 'q', which stems from texture lookup coordinate specifications. In this work, for simplicity only the `xyzw` family is used to specify components except to note in Section IV.2 that CGIS also supports `rgba`.

[6] Selection is a form of the more general *swizzling* concept, explained later.

Figure II.9 shows masking and swizzling together. The specifications of masking and swizzling are called *masks* and *swizzle*, respectively.[7]

Figure II.9 Masking and swizzling in an ADD-instruction operating on a 4-tuple



For implementation of high-level languages, one advantage of masking and swizzling lies in the potential for a better register allocation, as explained in Section V.5.c. Briefly speaking, multiple values might reside in one register, with instructions selecting values with swizzling and choosing those to be updated with masking. GPU languages also expose swizzling and masking to the programmer; here it is unfortunate that two operators with a very different semantics have the same syntax. For GPGPU programming, swizzles and masks can make possible tricks to avoid register transfers.

GPUs also support some other modifications of operands such as negation, and target modifiers such as clamping the result to the range $[0, 1]$.

Condition Codes

Modern GPUs have adopted another familiar feature of CPUs, but with a twist: the *flag register* or *condition code register*. A condition code register consists of a number of flags which are updated according to the result of a computation, and which can be tested in several ways.

For example, a *zero flag* is set whenever the result of a computation is zero, and unset otherwise. Other flags are set or unset depending on the sign of the result or on an arithmetic overflow condition. In GPUs, these flags are not updated on every operation, but operations can be marked to update the flags.

A traditional use for these flags is to implement conditional branching. For example, on Intel CPUs a `CMP a,b;` instruction produces no arithmetical result, yet sets the flags as though the operation `SUB a,b;` had been executed [I06a]. Branch operations then make use of these flags: The `JE` instruction takes a branch if the zero flag is set, hence executing a jump if the preceding comparison has found the operands to be equal.

This is the case also for GPUs. However, GPUs can mask other operations in much the same way as explained for the static masks. That is, a condition code register also has a number of components, which are set independently of each other. A subsequent operation can then specify that its effects shall take place only for those components on which the flags are set in a certain way.

This feature enables GPU programs to use *guards* of instructions, making possible a technique called *if-conversion*. Section V.3.c, which is concerned with this technique, provides more details about the condition code register and how it is written and read.

[7] Swizzling and masking in CGIS are a bit different from that, regarding these suffixes as type modifiers, that is, properties of the operand and target, not of the operation (Section IV.2).

Precision

A perpetual problem for GPGPU computations is that of precision – both of actual hardware precision and of precision assured by specifications and standards. In the early days of programmable GPUs, the data used were fixed-point data with about a dozen bits. Now, the GPUs feature floating-point data-types in 32-bit IEEE format [I85].[8] However, the amount of bits used to store values does not tell much about the precision of the *computation* on those values. [GST07] reports that the `MAD` instructions, which computes $a \cdot b + c$, uses rounding after the multiplication step on NVIDIA, yet performs the full computation before rounding to 32 Bit on ATI. Such information is not part of the specifications proper, but must be obtained case-by-case from vendors.

For applications content with lower precision data, it is possible to pack multiple values of lower precision in a single 32-Bit register. The main use of this feature is not quite in computation, but more in decreasing the number of memory accesses. This number can be severely restricted, as will be explained in the next section.

The newest generation of GPUs features also integer registers and classical logical operations [N07c]. Unfortunately, the instruction set of floating point operations has not completely been extended to the integer operations. Some of the restrictions shall be mentioned in Chapter V. For now it suffices to say that this extension, appearing at the end of 2006 with the G80 GPUs, opens up a much larger world of computation to GPUs.

Control Flow

Loops and Conditionals. The qualitative characteristic of control flow has seen a rapid advance in the last few years. The first control flow constructs allowed were conditionals with constant conditions and loops with constant loop counts in the vertex processor in VS 2.0. This does not seem particularly useful at first, but the constants can be set by the application on the host before running the program. Indeed, constants in GPU kernels are only constant for a particular run of a program, but need not be constant across runs. Thus, these control flow constructs served as parameterisability constructs. In this sense, this limited control flow illustrates the transition from parameterisable algorithms towards full programmability.

VS 2.x saw the introduction of data dependent break instructions, which could prematurely end loops. Pixel Shader got control flow instructions a little bit later than the vertex shader. In the NVIDIA GPUs, the NV40 generation supported loops with specified upper bounds and data-dependent breaks; the G80 generation finally supports full looping without prespecified maximal upper iteration bounds.

Subroutines. Whereas the aforementioned control flow constructs nowadays offer the same power as their CPU counterparts, the situation is dire for subroutines. Basically, modern GPUs offer subroutine capabilities and support a call-return stack, yet no parameter stack. Thus, the standard ways of parameter passing are not applicable here [WM95]. This is an artefact of the restricted memory system, and therefore it shall be explained in the following Section II.3.b.

Summary

In many cases, GPUs are on par with CPUs, and in some they even offer additional capabilities, in particular with respect to vectorial instructions. They still lack in the areas of precision and control flow. But looking back at the advances since 2000, one cannot help but being amazed at the rapid advances of the development.

[8] Four of such components make up for a 128-bit vector register. Hence the advertisements of "full 128-bit precision".

II.3.b Memory Model

The discussion of the memory model is divided into two parts: *Which* parts of the memory are visible, and *how* are they accessible?

Access

When looking at Figure II.7, it becomes immediately obvious that there is *no read-write-able memory*. Indeed, whereas the host system can both write into and read from the main memory areas shown in the diagrams, the input buffers (holding the textures) and the output buffers (holding the framebuffer), a GPU program cannot. This is not a limitation of the physical memory system on the chip: At the host's bidding, access directions of a memory block might be switched. There is no copying of data involved. In contrast, the restriction is a logical and organisational one: With parallel computations, accesses to memory remain well-defined only when none of the computations desires to write into an area read by another computation. Assigning a chunk of memory a complete no-read or no-write flag overcomes this problem.

Obviously, this poses a great number of difficulties. For example, an arbitrary parameter stack for function calls cannot be implemented in main memory anymore. Also, any non-destructive updates of data structures need to use a temporary copy memory to store the outputs and subsequently copy them into the data structure.

Visibility

The main point regarding visibility is that a GPU program cannot chose which parts of the GPU's memory to see, the CPU has to do so. Input buffers and output buffers cannot be swapped in or out of the GPU's view by actions from the GPU itself. If the programmer needs to change the memory configuration, the CPU has to wait for the GPU to finish its computations, perform that change and let the GPU resume its computation.

But there are further restrictions, namely in number, shape and size of the memory visible to a GPU. First of all, the input and output buffers are rectangular arrays. These arrays must not overlap – if they did, usual aliasing problem could arise, which are even more severe in parallel computations than in the sequential case. The number of such arrays is restricted. For example, the NV30 generation could only write into a single array (one buffer of the framebuffer). Later generations raised this restriction to four or even eight arrays. The input arrays are also restricted in number, but not as much: Hardware could make use of multiple input arrays (textures) since the advent of multitexturing in the 1990s[9]. This is the main benefit for the data packing mentioned in Section II.3.a: If only a lower precision is necessary, programs can output more values in a single buffer than were possible with standard four-component vectors.

Another point about the chunks of memory having to be created and specified by the host system is that the GPU *cannot allocate memory on its own*, not even inside some sort of virtual memory space. This obviously precludes implementation of many standard data structures, from linked lists over binary trees to dynamically growing arrays.

To programmers, this is a radical departure from the standard CPU memory model. It might even be more restrictive than many of the instruction set restrictions. Indeed, integer computations can be simulated, in certain ranges, with floating point computations; conditionals can be substituted by guarded computations; loops might be unrolled, if small enough. Yet there is no substitution possible for sheer memory.

[9] ARB_multitexture was created in 1998 and promoted to core OpenGL 1.3 in 2001.

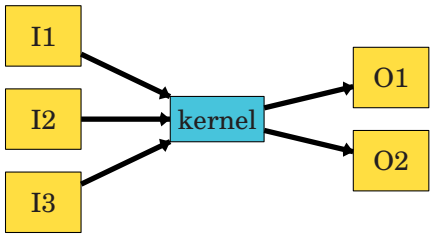
This, of course, pushes itself through to higher-level languages: Some restrictions can be hidden from the programmer, others cannot. For example, a CGIS programmer can use memory both for in- and output; the language semantics specifies sequentialisations for those memory accesses, and the implementation ensures faithfulness to this semantics by creating and updating copies of data as needed. The missing memory allocation cannot be emulated, however, and thus such features – indeed, all pointers[10] – are absent from CGIS.

II.3.c Streaming Computation

The restrictions of the memory model are hard to come to terms with. However, parallel computation with separated in- and output data is a quite well-known programming paradigm, called *stream programming* [S97a]. Stream programming is a very old model of computation; its roots go back to the 1960s: [B75] attributes the notion to [L68].

In the particular case of GPUs, the restrictions translate to a particular kind of stream computing. In this model, a *stream* is a (usually large) uniform sequence of fundamental elements. Streams are operated upon by *kernels*, instances of which are run in parallel on each of the stream elements. Kernels output elements, which in turn form another stream. Kernels might also get their inputs from several streams and output several streams (Figure II.10). However, in the pure streaming model, there is still a *bijection between all involved streams and the kernels*, e. g., no two kernels get the same stream elements as inputs and all streams are consumed and produced completely.

Figure II.10 Stream programming



Comparing Figures II.8 and II.10, the *input arrays* in Figure II.8 can function as streams. But these arrays can in effect be additional input to the kernels, where a kernel can access any element, indeed, an arbitrary number of elements, and elements may be accessed by several kernels. That is, stream programming on GPUs supports *lookup* and thereby *gather* operations. It does not, however, support a *scatter* operation, much to the chagrin of programmers.[11]

Note that the uniformity constraint mentioned above holds only on the hardware level. Obviously, algorithms might make use of non-elementary data structures as stream elements, and input data might naturally lie in a sparse or hierarchical format. These data structures, however, have to be cast into hardware supported formats.[12] Exactly this mapping is often a major stumbling point when porting an algorithm to the GPU.

[10] Array indices are supported.

[11] Of course, there are ways to simulated scattering on GPUs. It is still not possible within the scope of a *single* fragment program.

[12] For example, a stream of structures might be split into separate streams of elemental values, and hierarchical data structures are fed to the kernels as the random-read-access input arrays.

And such should be interpreted the term *general purpose computation*, when applied to GPUs. General purpose computation does not mean to run a database application or a web-browser, but it is firmly confined to the streaming model. Indeed, general purpose means only *more general* computations than those employed in the rasterisation algorithm (Section II.1.a).

II.4 Comparison with CPUs

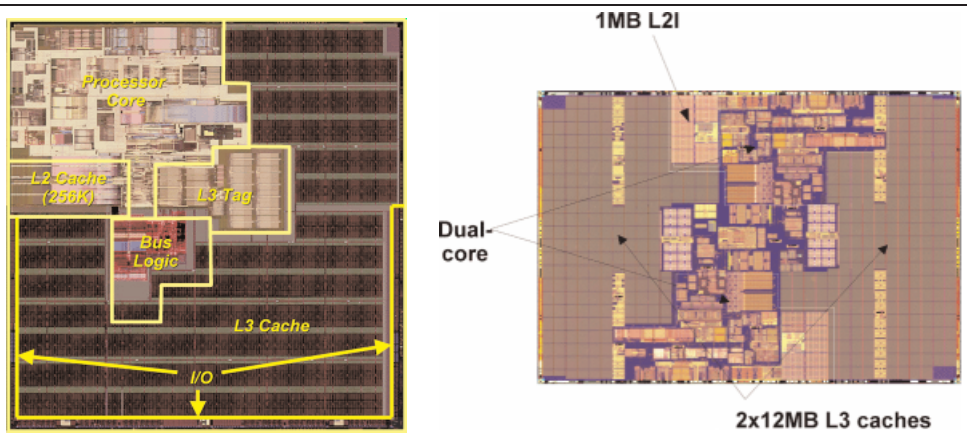
There are various motivations of using GPUs for general purpose programs. GPUs might be better suited to perform a computational task than CPUs, or the direct coupling of visualisation and computation might be attractive, or they might be used just because they are there and an easily accessible *additional* computational device.

Only in the first case GPUs are in direct *competition* with CPUs, yet a *comparison* is always instructive. The previous sections have described GPUs often in relation to CPUs, and Section II.5 is concerned with the future of GPUs. The current section, now, describes the history of CPUs, the technological advantages and shortcomings of current CPUs, and offers an outlook into the future.

II.4.a Technology

CPUs are optimised for high-performance on single threads of code. This is achieved by increasing the speed of the execution units. The Pentium IV class CPUs have core speeds of nearly 4 GHz. Feeding these execution units, however, poses a difficult problem, in particular in face of data-dependencies. To this end, CPUs employ tricks such as out-of-order execution, branch-prediction and especially large caches to ensure that the execution units stay well-fed [HP03]. However, these techniques need quite a large number of transistors; in effect, the real-estate of the chip is divided in an imbalanced way: A large number of transistors is devoted to keeping a small number of transistors productive. Figure II.11 shows actual processor dice.

Figure II.11 Dice of Itanium 2 processors

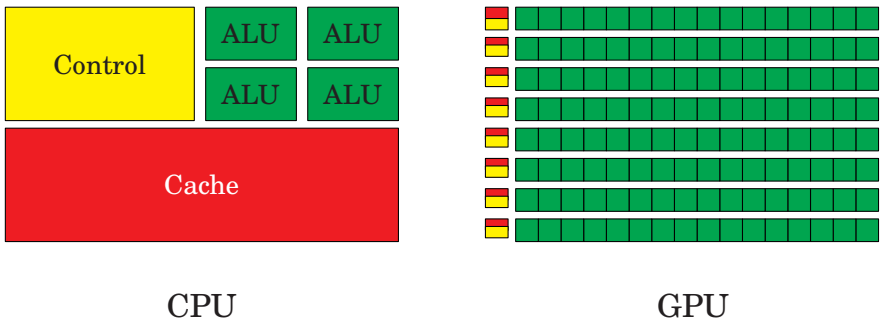


(a) Madison (b) Montecito

As explained in Section II.1.c, GPUs can use larger numbers of simpler execution units for their standard operations. Figure II.12 (adapted from a GPU vendor's brochure)

shows a very schematic view of this situation. Almost all transistors are devoted to execution units. These units can be kept well-fed in typical graphics application, because there are hardly any memory stalls: The sheer number of fragments[13] to be computed upon means that memory access can be masked by just working on another fragment until the memory fetch returns. This is no problem because there are no inter-fragment dependencies, as explained in Section II.1.a.

Figure II.12 Schematic comparison of transistor usage in CPUs and GPUs



This characteristic is, of course, useful only for specific classes of algorithms. Candidates are mentioned at various places in this work, starting from next section. For now, it is important to recognise the difference between CPUs and GPUs, and in particular, why CPUs have faced problems getting faster for normal applications in the last years; Indeed, as an Intel paper of 2003 observed, "graphics processors are getting faster at a faster rate than general-purpose processors" [BHK03]. Increasing the power of GPUs is to a large part a matter of more-of-the-same, in particular, more execution units. Increasing the power of CPUs entails also for some part more-of-the-same, but in particular, more caches to better exploit the existing execution units. For a large part, however, more logic has to be employed, be it better branch prediction inside the CPU or static scheduling by the compiler as for the Itanium architecture [I06b].

Current CPUs incorporate a multitude of cores on a single chip. That means that several independent CPUs, possibly sharing caches, are built on a single die; Intel offers quad-core Xeons, Sun is shipping an 8-core Niagara processor. To the outside, these devices are still separate CPUs, of course. However, even CPU vendors are considering moving to many-core architectures. The recently proposed Polaris chip of Intel features 80 cores with 2 FPUs each [R07]. The Cell processor features *Synergistic Processing Units (SPUs)* which lie between full-fledged CPUs and simple, specialised hardware in terms of programmability (*"The intent of the SPU is to fill a void between general-purpose processors and special-purpose hardware."*, [IST05]). With the recent acquirement of ATI, AMD plans on merging GPU and CPU, providing specialised solutions for different needs: A GPU-heavy chip will lend itself for graphics computations or other kinds of data-parallel computations, a CPU-heavy chip lends itself to data-centric applications. These trends show that upcoming generations of chips might benefit from the experiences in GPGPU, both in designing languages and in creating applications.

All in all, we see that GPUs are currently a specialised architecture, drawing their performance for certain algorithms exactly from that specialisation. In the future, the ar-

[13] NVIDIA: *"thousands of independent, simultaneously executing threads"* [N06].

chitectural experiences gained from designing GPUs and GPU programs will very likely flow back into the development of new CPU architectures.

II.4.b Programmability

Current CPUs offer a variety of SIMD extensions. The most popular extensions are the extensions for CPUs of the x86 family: MMX, 3DNow!, and most importantly SSE [I06a, A00]. SSE offers many arithmetical instructions on vectors, just like GPUs. However, even with their SIMD units, CPUs do not even come close to the number of arithmetical units and to the degree of parallelism offered by GPUs, and of course the memory bandwidth is not affected in any way by the presence of SIMD computational units. For a SIMD capable CPU, the SIMD features are mostly an additional gimmick used to speed up a sequential algorithm, whereas GPUs both require and support different programming approaches. The main design foundation of floating-point units in the GPU is still "many slow units", not a few fast units.

CGIS can also be compiled into SIMD code, targeting either SSE2 extensions for x86 CPUs or the AltiVec extensions for PowerPC code [F06]. This lies out of the scope of this thesis. For more details, consult [FLW07, F08].

II.5 Future Architecture

The main driving force of GPUs are computer games. Highly-powered, specially styled computers are the successors of the once-popular highly-powered, customised cars in a certain demography, and with gaming becoming more and more mainstream also among well-paid employees, there is a constant demand on yet more powerful high-end GPUs. In the mass market, a large part of the measurement of a computer's utility is given by its performance in games. Thus, it is essentially the computer games industry driving the GPU market. GPUs sell (or don't) by their performance in only a selected few games and games-related benchmarks.

In contrast, GPGPU applications are a nice source of press releases and occasional admiration, but not a main force behind hardware (and driver) development: Multigrid solvers do not sell GPUs, and performance of a multigrid solver is a side-effect of performance in first-person shooters. This has begun to change in the very recent past. NVIDIA's CUDA initiative [N07a] and ATI's CTM [A06] are a big step in this direction, and GPGPU gets more mainstream attention [M04]. The recent merger of ATI into AMD, which might lead the way for integrated $\frac{C}{G}$PUs, is another sign that GPGPU will get more attention by the manufacturers.

The remaining relatively-easy-to-remove obstacle for a more widespread success is the lack of double-precision arithmetic. Native double-precision support would open up GPUs to a huge range of scientific computations. The other obstacle, lack of global read-write-memory, is unlikely to be removed in the future within the graphics APIs. The design rationale for DIRECTX 10, which was developed by Microsoft in tight cooperation with the hardware vendors, argues that such a requirement was excluded for performance reasons [B06]. However, the technology is accessible by other means, and NVIDIA's CUDA initiative (Section III.4.b) permits kernels to use shared read-write memory. This, of course, requires synchronisation and communication mechanisms as common in parallel programming.

In any case, the market penetration of GPUs is going to increase, and applications using GPUs as auxiliary devices will be able to expect relatively highly-powered GPUs in all

PCs. The reason is not only games: Even in office PCs, eye-candy of the latest operating systems and windowing systems plays a significant role in the decision process of selecting GPUs to buy. For Windows Vista with the Aero desktop interface, Microsoft requires only an extremely modest 1 GHz processor, yet a DX 9-class GPU [M07b]. This means that even business PCs are going to employ GPUs of relatively high power.

Thus, GPGPU is and will remain a worthwhile activity to contemplate, and GPGPU languages, such as CGiS, will retain their use in the future, and might even increase their importance.

II.6 GPU Applications

Numerous algorithms have been implemented on GPUs. Some applications were written in the days of configurable texturing and blending, others during the exiting days of early GPGPU, and yet others in the current age of a well-established trade. A few were written using the most basic graphics operations, some have made use of abstracted, but low-level languages, others used high-level programming systems.

This section is devoted to GPGPU applications past and present, to instil in the reader a feeling for the possibilities and peculiarities of GPUs as targets for general purpose calculations. It is not meant to be a comprehensive overview; for this, the reader is referred to [OLGHKLP07]. Instead, it shall serve as an illustration of whence GPGPU came and where it stands now.

II.6.a The Early Days of GPGPU

Even before the advent of programmable fragment processors, GPUs had been used for general computations. Two applications shall illustrate this.

Computation with Multitexturing and Blending

[LM01] perform matrix-matrix multiplication on *non-programmable* GPUs. Considering a multiplication $AB = C$, where we assume for simplicity $A, B, C \in \mathbb{R}^{n\times n}$, then each single element $c_{i,j}$ for $i, j \in \{1, \ldots, n\}$ is defined as

$$c_{i,j} = \sum_{\mu=1}^{n} a_{i,\mu} b_{\mu,j}.$$

The authors implement the *addition* operation with a standard feature of the graphics pipeline: With a fragment incoming in the framebuffer at a position where a pixel is already present, *blending* computes the new colour as an operation of the fragments colour and the present pixel colour. For the operation at hand, the framebuffer thus keeps the temporary values at position (i, j), and each single result $a_{i,\mu}b_{\mu,j}$ is combined with an additive blending onto the temporary value. The *multiplication* is performed with another standard feature of GPUs, called *multitexturing*: To a single point, a multitude of textures could be specified, which are then catenated with a specified operation; in this case, multiplication. This multiplication can be performed in parallel. Indeed, the algorithm is a sequence ($\mu = 1 \ldots n$) of parallel computations ($\forall i, j. c_{i,j} += a_{i,\mu}b_{\mu,j}$).

The greatest difficulty faced by the authors is the inadequate precision. At that time, GPUs had only 8-Bit fixed-point formats for I/O. Internal precision was found by tests to be 14-Bit or 16-Bit. Obviously, this is not enough for serious applications, and the authors report scepticism whether GPUs will ever be really useful as auxiliary computational devices.

Computation in the Vertex Processor

[THO02] employ the GPU for floating-point computations. Using DX8-class hardware, they performed various performance tests on embarrassingly parallel work loads and some more complicated examples. Although the vertex processor, which is used for computations, works with full floating point precision, the results have to be passed through the fragment processor and the framebuffer before being available for read-back by the host system. This truncates the overall precision again to 8-Bit.

The precision problem notwithstanding, this paper showed the GPU's *"devastating efficiency"*. In particular, for embarrassingly parallel problems, the run time on the GPU is much more robust to increase of work load size than the CPU, for the time to set up the GPU and to transfer data dominates overall time.[14] For a naïve matrix-matrix multiplication, the authors report a speed-up of 3.2 on a high-end GPU with respect to a high-end CPU; other very promising results were achieved on a randomised 3-SAT.

In the end, the authors express a much more optimistic view of the future of GPGPU. Apart from usual proposals for hardware development, they state a desire for a compiler for GPUs to alleviate the programmers from coding in assembly language.

II.6.b The Rise of GPGPU

More and more complex applications emerged in the following years. Various applications from all kinds of areas have been implemented on GPUs. As an example of the development in this age, this section mentions one representative of the area of rendering and one representative of raw-power linear algebra.

Global Illumination

[PDCJH03] is an example of the wide-spread use of GPUs for visualisation. Obviously, creating images from scene descriptions is the main point of existence for GPUs, as explained in Section II.1. However, there are several algorithms for this which are as far away from rasterisation as database searches or sparse matrix multiplication. As such, ray tracing (e. g., [P04]) is indeed a prime example of a GPGPU computation using the integrated visualisation capabilities of GPUs. In the same way, this holds for *photon mapping*, the topic of [PDCJH03].

Photon mapping is an algorithm for *global illumination* [G95]. Briefly speaking, photons are sent from a light source, cast into a scene, and they hit objects. Upon a hit, this is stored and another photon is sent again into the scene. In fact, the basic idea is very similar to ray tracing, which traces photons *backwards* from the eye to compute what can be seen in a specific view angle.

One of the main contributions of this paper is an implementation of a bitonic sort algorithm.[15] Many sorting algorithms need random writes, which cannot be performed on GPUs (see Section II.3.b). Bitonic merge sort [B68] is a parallely implementable sorting algorithm with fixed comparisons and swaps. The authors show its suitability for implementation on GPUs.

This paper shows that even the desire for better image quality can lend itself to algorithms implementable on GPUs, and that those algorithms can employ implementation techniques suitable in an even more general sense in all kinds of GPGPU applications.

[14] The GPU is more scalable, in other words. Section VI.2 mentions similar results for CGIS applications.

[15] The scene is divided into cells, which may hold photons. The photons to be traced need to be sorted by cell number.

Linear Algebra Operators

The authors of [KW03] see their work as a cornerstone of a future BLAS library [D02] on GPUs. They point out that in contrast to previous work, which just implemented some algorithms on GPUs, their goal is *"to develop a generic framework that enables the implementation of general numeric techniques for the solution of difference equations"* – Thus, this is a step towards making GPUs a commodity for users of scientific applications.

They work on DX9 hardware, an ATI Radeon 9800, which offers 32-Bit floating point I/O and 24-Bit internal precision. They implement a layer of abstraction on top of the hardware, which exposes vectors and matrices as C++ data types and functions performing operators on such objects. In contrast to other approaches using the GPUs for operations on dense matrices only, they also support sparse matrices. With these building blocks, larger algorithms are quite easily implementable. Tests are performed for the conjugate gradient method and the Gauss-Seidel method for solving a system of linear equations.

This is a representative of attempts to make GPUs more accessible to normal people without the need to go down to the details of the graphics system, yet still offering as much performance from the GPU as possible.

II.6.c Current State of GPGPU

The field of GPGPU has changed in the last years. In the beginning, it was exciting to see algorithms ported to the new, strange target. With the initial stir settled down, new work is approaching GPGPU from different angles.

On the one hand, much has been done so far, and therefore many tricks have been implemented and published. Even if the source code is not available or not integratable in one's own framework, one can build on descriptions of previous work for several basic techniques, and need not to reimplement some wheels over and over again.

On the other hand, new, higher-level languages such as Brook for GPUs (Section III.3.a and [BFHSFHH04]) offer a level of abstraction not known or possible to achieve in the early days. Obviously, true *general purpose* programming still is not possible, but the amount of work done so far has made GPGPU a well established field. This is particularly well visible in the development of conferences. Most early papers on GPGPU have been published on graphics conferences, even though they actually had nothing to do with graphics (e. g., [KW03]). Now, there are workshops and conference tracks devoted to GPGPU alone (in 2006, in ICCS and at SC), where researchers can build upon a general knowledge of graphics hardware and of GPGPU not present in other venues.

To provide a feeling for the recent trends, two papers shall be introduced.

Basic Computation

[GST07] investigate the possibility for floating point computation with higher precision on GPUs. Recall that GPUs are still stuck with single-precision floats. By storing values not in a single float register but components of it in two (one part with a lower exponent and one part with a higher exponent), one can achieve a higher storage precision. To perform computations on those values, they recall old techniques from the 60s, where high-precision hardware was not wide-spread, and newer tricks from current multiple precision or arbitrary precision libraries. With this, they give algorithms on how to perform multiplications and additions on such values on GPUs.

They make their case by investigating larger applications, where only small parts of the computation are performed in higher precision arithmetics, whereas the most part is done with standard single-precision values and operations. Experiments with an iterative refinement scheme, where the GPU computes solutions to $Ax = b$ in single-precision and the CPU computes error correction in double precision, show that this way of mixed-precision arithmetic leads to a stable and fast algorithm. In contrast to this, straightforward high-precision emulation on the GPU with its single-precision registers and operations was seen not to be competitive.

All in all, this paper shows two ways to cope with the problem of too small a precision on GPUs. Depending on the actual needs, programmers may find one or the other helpful.

Data Abstraction

The GLIFT library [LKSSO06] provides a template-based data abstraction library for GPUs. Users can specify *physical memory* data structures (modelled as textures) and build from these *virtual memory* data structures. The library then generates *address translators* which offer an abstract access to the underlying, basic physical data structures by means of the more abstract, virtual data structures. Kernels are written in a shading language such as CG (Section III.2.a), which is extended to cope with the new abstract, templatised data types.

In essence, GLIFT offers a solution for a very pressing problem, namely the mapping from recursively defined data structures to the GPU world of textures. Its appeal to the experienced C++ programmer is further enhanced by the STL-like notation. For example, the CG kernels can syntactically work on data elements with an iterator notation.

Although GLIFT provides a very abstract view of the GPU and takes care of several cumbersome duties, it is not described in Chapter III, for it cannot be called a GPU language. As the authors write: *"The goal of Glift is to provide generic GPU data structures and iterators, not provide a runtime GPU execution environment."* It is to be used alongside with other languages such as CG, which can then benefit from the higher-level data structures.

II.7 Summary and Outlook

In this chapter, we have seen GPUs as targets for general purpose computations. Based on the history of GPUs, I have described their current architecture and argued for their strengths and weaknesses. A short overview of GPGPU has provided, by way of example, a feeling for the current state of abstraction and of what kind of help, what foundation to be used from previous work, is available to the programmer.

We have seen that from a curiosity, the field has risen to a well-established trade with a wide range of published literature. But down to what level do potential users have to go in order to exploit the performance? The code still has to be written somehow, in a programming language. Thus, the next chapter investigates the current choices of programming languages and argues for the development of another language, CGIS.

III

GPU Programming Languages

In gewissen Sprachen. . . ist Das,
was hier erreicht ist, nicht einmal zu *wollen*.
FRIEDRICH NIETZSCHE, *Götzen-Dämmerung*, 1889

Computer languages are a means to express an algorithm. Characteristics of a programming language depend on the characteristics of the hardware as well as the characteristics of the problems expected to be expressed in that language. For GPUs, a variety of languages has been proposed and implemented, approaching the problem space from different points of view and offering solutions on different levels of abstraction.

Therefore, before investigating the language CGIS in Chapter IV, the present chapter is concerned with existing GPU languages – in a sense, it could very well just be called "Related Work". Section III.1 describes the distinction between shading languages and general purpose languages. These kinds of languages are in turn the topics of Sections III.2 and III.3. Section III.4 considers the hardware specific languages for ATI and NVIDIA GPUs. Section III.5 sums up and concludes this chapter.

III.1 Levels of Abstraction

GPU programming languages fall into two categories. Seemingly, *shading languages* could be defined as those languages which are used for shading computations (Section II.1.a): Computing the colours of pixels in images. *General purpose languages*, on the other hand, support arbitrary computations. Again, *general purpose* is just a qualifier laying out the difference to shading languages. As mentioned before, general purpose computing on GPUs is not truly general purpose, because the fundamental limitations of graphics hardware cannot be overcome. Thus, also so-called general purpose programming languages need not provide support for implementing a database system or porting NetHack.

However, a distinction along those terms proves not to be useful. General purpose algorithms have been implemented on GPUs before the advent of general purpose languages.

The authors wrote the program in a shading language or in assembly language. Furthermore, some languages generally considered general purpose languages might be used for classical shading purposes. Clearly, a distinction by the *possible* uses of languages does not lead to an appropriate language classification.

Instead, it is instructive to consult Figure III.1, for it serves well to illustrate the classification of languages that shall be used in this work; a classification that very much fits the *pragmatical* difference between shading languages and general-purpose languages.

Figure III.1 Using a GPU as a co-processor

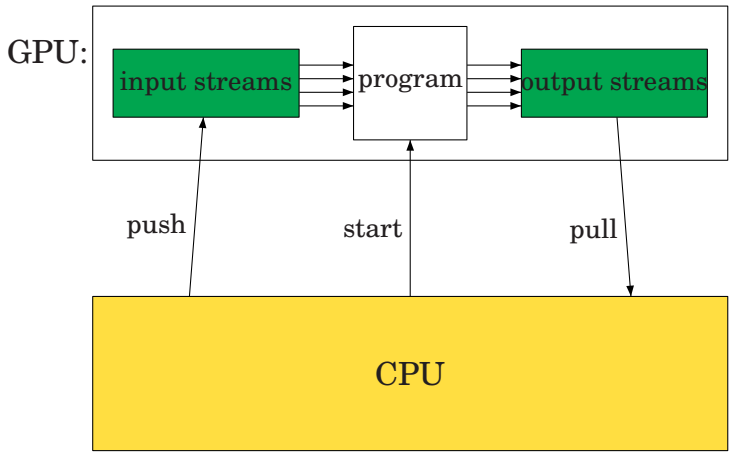


Figure III.1 exemplifies the use of a GPU as a co-processor. In hardware terms, what happens is that the CPU uploads the input streams (Section II.3.c) to the GPU, (uploads and) starts the GPU program, and downloads the output streams. Programming language support is provided on two subsets of these actions:

- Shading languages target solely the *program* box, i. e., they provide a more abstract way to write GPU programs.
- General purpose languages additionally deal with the *push*, *pull* and *start* arrows, i. e., they *completely* abstract the GPU.

The difference made by that abstraction should not be underestimated. These three arrows are all sequences of commands of graphics APIs. To implement those operations, programmers must learn to program a significant part of the graphics hardware – indeed, the largest part – by functions of the graphics API and in terms of the rasterisation algorithm. Clearly, that is not a satisfactory state of affairs. Certainly, one can expect that to achieve high performance on a particular target, the programmer has to have and to apply knowledge about that target. Going through the graphics API, however, poses an *additional, administrative and terminological* burden on the programmer. Section V.1 talks about several issues in the implementation of CGIS, and this can give a glimpse on what expects programmers striving to do this manually, although the presentation obviously will leave out much of the boring details. It suffices to say that this task cannot be imposed on ordinary programmers if GPGPU is to go mainstream.

By this distinction, it becomes obvious that the languages with that high a level of abstraction as those termed *general purpose languages* are much better suited for being

used as tools by general purpose programmers, and hence, the name fits the applicability.

The preceeding discussion described the small focus of shading languages compared to that of general purpose languages as a liability, but for the specific use of describing *shading* programs (Section II.1.a), this is actually an asset: It means that the programmer can plug in these programs to ensure a greater flexibility in a particular part of the graphics pipeline, while still taking advantage of high-performance-optimised fixed function parts. In particular, such shaders are easily integrated into existing graphics libraries. So, although shading languages can and should be criticised from the point of GPGPU, they have their use, and they fill their role well.

III.2 Shading Languages

Shading languages in general have been invented and implemented on CPUs much earlier than the advent of programmable GPUs. The language RENDERMAN [AG99] has long since been used to produce computer generated images. A complete rendering pipeline can be specified in RENDERMAN, and in particular RENDERMAN features a language for shaders. Animation films are usually rendered on large computer farms, for rendering a single frame in the desired quality can take hours. Indeed, interactive execution of RENDERMAN on graphics hardware is the Holy Grail of hardware accelerated, programmable shading – and it remains as elusive as the mythical trinket, so shading languages for GPUs have been designed with focus on the hardware rather than the authors. In this section, the shading languages in the sense of Section III.1 are considered: CG, HLSL and GLSLANG.

III.2.a Cg

CG [N05] is designed and developed by NVIDIA, but heralded as an open standard. Indeed, it can target not only the NVIDIA-specific OPENGL extensions for programming, but also the standardised assembly languages of DIRECTX and OPENGL (Section II.2.b). The subsequent sections III.2.b and III.2.c will strongly build upon the description of CG, because the languages are similar enough that a discussion can for a large part be confined to an exhibition of differences.

CG offers two levels of program specification.

- CG proper is a programming language targeting vertex and fragment processors separately.
- CGFX is a specification of state of the graphics pipeline, i. e., it includes programs for the vertex and fragment processor, but also parameters of functions such as the depth test (Section II.1.a) or blending (Section II.6.a).

We shall delve into these two levels one by one.

CG

Example. The programming language CG is a high-level, imperative language for the vertex and the fragment processor of GPUs.[1] *"The Cg language is based on both the*

[1] One can write a program for the vertex processor, and one can write a program for the fragment processor; One cannot write a program for the vertex and the fragment processor.

syntax and the philosophy of C" [MGAK03]. Each kernel is written in a fairly standard way, as we will see by an example.

Program III.1 shows a simple fragment program. It does nothing more than multiplying the fragment-specific colour with a constant parameter and adding a texture. Instead of investigating the program line-by-line, only the worthwhile details are to be considered; for a detailed explanation of CG, the reader should consult the appropriate documentation [N05].

```
void fragmain(float4 col : COLOR, float2 tex : TEXCOORD,
              out float4 pixel : COLOR,
              uniform samplerRECT texture, uniform float factor){
  float4 texel = texRECT(texture, fraginput.tex);
  float4 fragcolor = fraginput.col;
  pixel = fragcolor*factor + texel;
}
```

Program III.1: A simple CG program

Focusing on the *body* of procedure `fragmain`, we see an ordinary, imperative programming language with a few uncommon constructs – and that is exactly what CG is, on this level. By the keyword `float4`, vectorial datatypes are declared. They partake in usual arithmetical operations specified by the familiar operator symbols. One such operation involves the promotion of a scalar (`factor`) to a vector. The `texRECT` line is simply a stream lookup function, specifying the stream and the position where to look.

The *signature* of `fragmain` presents us with the first peculiarities of CG. Note that a particular parameter (`pixel`) is declared as having `out` flow, and that a *binding* specification (`: COLOR`) is added to it. This specifies that parameter to hold the result value of the fragment program, and additionally specifies whereto that result should be written in terms of the graphics pipeline model (Section II.1.a). Other bindings are specified for the parameters `col` and `tex`. These specify from which output of a vertex program the values are received.[2]

From this short example, it becomes immediately obvious that CG is firmly rooted in the graphics pipeline model. As mentioned at the end of Section III.1, this is not necessarily a problem: Indeed, such a shader can be very easily plugged in to replace another, hand-coded assembly shader or a parameterised, fixed-function texturing phase. In such cases, programmers know their inputs and outputs and the locations of these parameters, so that such a specification is merely an opportunity for specification. For non-graphics programmers, however, it is a liability.

Profiles. CG supports several assembly language output formats. Both OPENGL and DIRECTX are supported, and in OPENGL NVIDIA-specific extensions as well as the standard assembly languages. The compilation of a CG program into the target language can be performed off-line (the programmer runs the CG compiler on the source code and works furtheron solely with the generated assembly code) and on-line, i. e., at runtime of the application (the CG source code is passed to a function of the CG runtime system which compiles the program). Even in the runtime case, however, CG is not compiled to native code but to the API assembly languages [N06]. Obviously, some information is lost in the translation from CG to the assembly language; some of this is retained when translating at runtime, so that, for example, program parameters can be referred to by their names and not by the registers specified by their bindings.

[2] These are the interpolated values of the vertex program's computations on the vertex inputs.

More details about the runtime can be found in the part about CgFX.

Features. As seen in the short example, Cg programs are written either for the fragment or for the vertex processor. Binding specifications link the vertex program's outputs to the fragment program's inputs, and also link the vertex program's inputs and the fragment program's outputs to the preceding respectively the following parts of the graphics pipeline. Programs can work on program-constant parameters; these are the `uniform` parameters of Program III.1. Those constants can be set by the host; thus, they form a means of parameterisation.

Inside the programs, Cg offers arithmetical operators on scalar datatypes, with a few extensions and restrictions. For example, Cg offers a fixed-point binary fraction type and a half-precision type, but not necessarily integers. Vectorial datatypes are present as well as small (upto 4×4) matrices and appropriate operations on them.

Control flow operations are supported as far as possible. The Cg compiler can perform if-conversion for architectures not supporting true conditionals. Standard iteration constructs can be compiled for architectures not supporting looping, if the loop can be unrolled completely at compile time. Functions are supported by inlining; of course, recursion is not possible.

For advanced datatypes, Cg offers `structs` and `interfaces`. A `struct` may have member functions and can implement an `interface`. For example, in Program III.2, a `Light` is declared as an interface, which can be implemented by any structure providing a function `intensity` with the specified signature. The function `fragaux` uses a parameter of such a type. This function is then called with a structure `light` of type `Const_light`, which happens to implement the interface `Light`.

```
interface Light {
  float intensity();
};

void fragaux(Light light, in float4 plain, out float4 mod){
  mod = plain * light.intensity();
}

struct Const_Light : Light {
  float intensity() { return 0.5; }
};

void fragmain(float4 col : COLOR, out float4 pixel : COLOR){
  Const_Light light;
  fragaux(light,col,pixel);
}
```

Program III.2: Structures and interfaces in Cg

In such a way, a limited form of object-orientation may be implemented. Advanced object-oriented features such as real inheritance, combination, extension or implementation of multiple interfaces are not supported.

Array types are also supported. Because of the memory restriction, arrays have to be held in registers, and thus they cannot be very large. Pointers are not supported, alongside with the address operator; Functions can output values via their parameters by the

`out`/`inout` modifiers. CG also features some predefined functions, such as trigonometric functions or clamping.

CgFX

The *effect framework*[3] CgFX is a specification format which holds GPU programs and various additional information about the state of the graphics pipeline. Whereas CG works on programs for the vertex and for the fragment processor separately, in CgFX one can specify that two such programs belong together. It is also possible to set various parameters of the graphics pipeline, such as the blending function, which specifies how an incoming fragment is catenated with a pixel possibly already present in the output buffer, or values of the constant input parameters (`uniform` in Program III.1).

This high-level specification format provides an overview of a desired program *in its context* at a glance without the need to keep track of the statements in the host program used to set up and manipulate the state. As such, it is quite helpful to the graphics programmer.

Evaluation

CG's main advantage is that of *portability*. With its various targets and its independence from vendors and APIs, it can be regarded as something like a portable higher-level assembly language. CG provides a definite step of abstraction over the bare assembly language programming.

However, CG itself it just a shading language, in that it targets exclusively the processors themselves without any further abstraction. Even CgFX is nothing more than a more concise and readable version of the graphics state manipulation needed to run GPU programs. As such, it is not suitable for general purpose programming as defined in this work (Section III.1).

But it was never intended for CG to become a language with such a wider scope. NVIDIA certainly intends CG to be used in GPGPU, but only as an efficient, yet readable language to program the fragment processors, and of course, CG then can be used for GPGPU. Indeed, CG was the first step of any kind above the bare assembly level for multi-platform programming, and the comment *"Cg is intended to be general-purpose..., rather than application specific"* from [MGAK03] should be read in this regard.

III.2.b HLSL

HLSL is Microsoft's *High-Level Shading Language* [M07a, PM03]. HLSL and CG have been constituted together, and thus most of what has been said for CG (Section III.2.a) holds for HLSL as well; the differences in the language are relatively minor (such as the `interface` feature not being present in HLSL). HLSL also has an effects framework similar to CgFX. The main difference is not the language itself, but the pragmatical difference of portability: HLSL is only available for DirectX. There it can target GPUs of various generations using the pixel shader and vertex shader versions supported in DirectX (Table II.1).

For purposes of GPGPU, the same applies as for CG, with the additional constraint of being confined to DirectX and thus to Windows.

[3] Effect is used here in the sense of "effect applied in the synthesis of an image".

III.2.c glslang

The OpenGL Shading Language[4] [K06] is the shading language which is part of the OPENGL specification. On the level of single programs, it is again quite similar to CG (Section III.2.a). Program III.3 shows the GLSLANG equivalent of Program III.1.

```
uniform sampler2D texture;
uniform float factor;

void main(){
    vec4 texel = texture2D(texture, gl_TexCoord[0].xy);
    vec4 fragcolor = gl_Color;
    gl_FragColor = fragcolor*factor + texel;
}
```

Program III.3: A simple GLSLANG program

Note the use of predefined variable names such as `gl_FragColor` to achieve the effect of CG's binding specifications. Some other differences to CG are that GLSLANG does not include fixed point datatypes and half-precision floats, yet does require support for integers[5], and that GLSLANG does not have the concept of interfaces. Additionally, hardware limits and some parts of the state of the graphics pipeline are exposed to the programs with predefined names. For example, the constant `gl_MaxDrawBuffers` holds the hardware-specific number of output buffers accessible to a fragment program, and `gl_ProjectionMatrix` holds a matrix used in computing the projection from a three-dimensional scene onto the two-dimensional plane (Section II.1.a).

In contrast to CG or HLSL, OPENGL provides only support for on-line compilation of GLSLANG code into an opaque format. To use a GLSLANG program, the user has (among many other things) to pass the source string to a function which returns a handle to an opaque object representing the program. There is no assembly language stage which could be inspected. GLSLANG also features a linking step: Vertex and fragment programs may be specified in a multitude of files which are then combined into a single program[6].

Older versions of the GLSLANG specifications [KBR04] had an *issue* list, keeping in tradition with OPENGL extensions, which brings to light several important points regarding the abstraction. As far as capabilities are concerned, it had originally been planned that GLSLANG would support programmable alpha blending (7) and read-access to the output buffer (23); these capabilities have only afterwards been removed. Issue 52 raises the very important point of virtualisation of resources. The policy on this is to distinguish between resources which are *"easy to count"* (such as number of input buffers (texture units)), and others. In the cases where it is reasonably clear to the user how many resources are needed, he is responsible for getting by with them. In other cases (such as length of the resulting GPU program), the implementation is responsible for virtualisation. This virtualisation would entail a quite complex splitting of the program into

[4] The language is here referred to as GLSLANG. GLSL is also a common abbreviation.

[5] GLSLANG has a type `int`, but because of hardware restrictions it need not really be implemented as an `int`: *"At the hardware level, real integers would aid efficient implementation of loops and array indices, and referencing texture units. However, there is no requirement that integers in the language map to an integer type in hardware."* [K06]

[6] This results in a terminology of *shader* and *program* which lies across to the rest of this work. Briefly speaking, to GLSLANG a *shader* is a an artifact written in GLSLANG which is to be compiled, and a *program* is the object in OPENGL which is to be used. For simplicity, I ignore this terminology here, and speak of vertex and fragment programs in the usual sense, whenever possible.

various parts with the associate state management to store and pass temporary parameters, or the emulation of the GPU on the host system. Similarly, issue 38 decrees that texture accesses on the vertex processor should be allowed, even though hardware need not support this.

Newer version of the specification have another issue list with different content. In [SA06], the ruling of Issue 52 has been rendered more precisely as such: *"A shader should not fail to compile... due to lack of instruction space or lack of temporary variables. Implementations should ensure that all valid shaders... may be successfully compiled, linked and executed."*

All in all, we see that GLSLANG strives to offer more *transparency* (virtualisation of resources) at the expense of *control* (assembly language output). But the opacity of the results of compilation and the linking process together also offer more opportunities for optimisation to the runtime system. For example, just as an emulation of texture access in the vertex shader entails pushing these accesses down the pipeline to the fragment program level, it might be conceivable that an implementation detects that a particular computation in the fragment program is constant across fragments, and thus lifts it to the vertex level. Other optimisations involving the linking of vertex and fragment programs are possible, for example cross-program dead-code elimination. Indeed, because the driver has the complete control over the programs, it can use the most detailed knowledge about all of the GPU, the programs and the current state of the graphics system in its compilation process. This is obviously not at all possible when compiling off-line or even on-line down to an assembly language: Information about the program gets lost during translation, and there is no information available about the program's running context. Thus, the opacity approach of GLSLANG is also a source of potential for higher performance.

III.2.d Shading Languages as Target Languages

We have seen several shading languages so far. As has been argued in Section III.1, shading languages are not the right tool for GPGPU; there needs to be a language on a higher level.

The next section evaluates those GPGPU languages, and argues for the need for CGIS, which shall be described in Chapter IV. However, a shading language could be used as the *target* of a GPGPU language. Indeed, BROOK (Section III.3.a) employs CG and HLSL as target languages. In this sense, a shading language can play the role of a *portable intermediate language*. CGIS does not use shading languages in this way, and this section explains the reasons for this.

- HLSL: Section II.2.b explained why CGIS targets OPENGL. Due to the restriction of HLSL to DIRECTX, it is not a possible target language.
- CG: When the work on CGIS started, the CG compiler produced unacceptable code. On the one hand, this is a problem in terms of performance: When a program uses more instructions than another program, it might take a longer time to execute. But more importantly, increasing the length of a program or the number of registers might effect its runnability, because it might just get over the static hardware limit. For this reason, CG was not adopted as the target language for CGIS.[7]

[7] This has changed in the last years. Nowadays, the CG compiler produces high-quality code.

- GLSLANG: A very viable option as a target language would be GLSLANG, in particular because of its tight integration into the OPENGL system. However, it was exactly this integration which resulted in GLSLANG *not* being chosen for CGIS: Because of the opacity and the virtualisation, an inspection of the generated code is not possible. This prevents reasoning about the performance of the generated code, because it is not clear whether a poor performance is resulting from poorly generated GLSLANG code, or a poor processing of the GLSLANG code.

Some other points are applicable to all languages. For one, CGIS is also an experimentation on GPU capabilities and on compiler techniques for GPUs. Thus, the more control the CGIS compiler has on the result, the better. Additionally, the internal representation of CGIS is based on sequences of primitive instructions (Section V.2), as needed for program analysis (Section V.6.a). Retranslating this pseudo-assembly language into a standard imperative language would result in highly atypical code. Thus, it is unclear whether the full optimisation potential provided by the shading language compiler could be realised.

For a similar discussion on a yet lower level of abstraction, see Section III.4.a.

III.3 GPGPU Languages

GPGPU languages target the GPU as a whole system offering computational resources available to data-parallel programs. To this end, they abstract away from the graphics heritage of GPUs. On the one hand, this increases the accessibility of GPUs to non-graphics specialists. On the other hand, with the GPU being hidden from access by the programmer, the compiler must ensure that performance-enhancing features are used effectively, lest the programmer be forced to fall back to lower-level languages for performance-critical code.

III.3.a Brook for GPUs

The stream computing language BROOK was developed as part of the Merrimac project for stream computing in general, not only on GPUs [BFHSFHH04, DHEKLAJKDGB03]. As one possible target architecture, it supports GPUs of various kinds. The system for GPGPU programming, "BROOK for GPUs", is called BROOK for short, just like in the rest of this work.

Writing BROOK Programs

BROOK is implemented as an extension of C with embedded kernels. The user writes C-code to declare streams, CG/HLSL-code to declare kernels and function calls to a runtime library to direct the execution of the program. Program III.4 shows an example. This is a *complete* BROOK program (assuming the data initialisation part in the "..." is added) – no further actions are necessary to direct the GPU.

Let us start the investigation of this program by focusing, for the moment, on the high-level instructions to create streams with data and to execute a program, not caring about whence the program comes. That is, we investigate the `main` function for now.

We see the declaration of *data streams* using angular brackets. They use the data type `float4`, which is simply a struct with four `float` components. By the library function `streamRead`, these streams are then filled with the contents of input arrays. The call

```
kernel void add(float4 IN1<>, float4 IN2<>, out float4 OUT<>){
   OUT = IN1+IN2;
}

void init_arrays(float4[], float4[]) {...}

int main(int,char**){
  float4 in_array1[256], in_array2[256], out_array[256];
  float4 in_stream1<256>, in_stream2<256>, out_stream<256>;

  init_arrays(in_array1, in_array2);

  streamRead(in_stream1,in_array1),streamRead(in_stream2,in_array2);

  add(in_stream1,in_stream2,out_stream);

  streamWrite(out_stream,out_array);
}
```

Program III.4: A simple BROOK program

to the `add` function executes a kernel with these three streams, and the subsequent call to `streamWrite` copies the contents of one stream into an array. It is obvious that the target is totally opaque. The user programs only in terms of streams and kernels.

```
void add (float4 IN1, float4 IN2, out float4 OUT){
  OUT = IN1 + IN2;
}
void main ( uniform _stype2 _tex_IN1 : register (s0),
            float2 _tex_IN1_pos : TEXCOORD0,
            uniform _stype2 _tex_IN2 : register (s1),
            float2 _tex_IN2_pos : TEXCOORD1,
            out float4 __output_0 : COLOR0,
            uniform float4 __workspace    : register (c0)) {
    float4  IN1;
    float4  IN2;
    float4  OUT;
    IN1 = __fetch_float4(_tex_IN1, _tex_IN1_pos );
    IN2 = __fetch_float4(_tex_IN2, _tex_IN2_pos );
    add( IN1, IN2, OUT );
    __output_0 = OUT;
}
```

Program III.5: CG code for Program III.4

The kernel, `add`, is defined in the same source file. Prefixed with `kernel`, it defines essentially a CG/HLSL[8] function with a bit of added concepts for data exchange. For example, the signature operates not in terms of stream elements, but in terms of streams as a whole; nevertheless, the CG kernel sees only the single elements, of course. Pro-

[8] For simplicity, only CG is mentioned in the subsequent examples, because in principle CG could also take on the role of HLSL, but not vice versa.

gram III.5 presents a section of the generated code; the complete program also includes a number of auxiliary library functions, only one of which is used here. It can be seen that the kernel itself is faithfully translated into CG. It is called from a function which only takes care of the data input and output: The function `__fetch_float4` is one of those library functions, which does nothing more than a texture lookup like in Program III.1.

```
reduce void sum(float IN_STREAM<>, reduce float OUT_SCALAR<>) {
  OUT_SCALAR+=IN_STREAM;
}

int main(int,char**) {
  float in_array[256];
  float in_stream<256>;
  int i;
  float result;

  for(i=0; i<256; ++i) in_array[i] = (float)(i);
  streamRead(in_stream,in_array);

  sum(in_stream,result);
}
```

Program III.6: Reduction in BROOK

The high-level approach is of particular importance in more complicated situations. For example, Program III.6 shows how to implement *reduction* in brook: All elements of the specified stream are added into one scalar. On a GPU, this has to be implemented in a multitude of *passes*, because a single kernel cannot necessarily fetch all input elements to add them in one step. This is completely hidden by the high-level specification in BROOK.

BROOK also offers random-reads, where a stream is regarded as an array, as seen in Program III.7. Scattering can be simulated on the CPU.

Using BROOK Programs

The general usage of BROOK is depicted in Figure III.2. There, solid lines denote in- and output, dotted lines denote linkage. The filled rectangular nodes are user supplied code, the filled oval nodes are part of BROOK, the CG compiler is not part of the system and the lower chunk is generated by BROOK.

A BROOK source file is compiled as follows. First, a modified C-Parser[9] parses the input file, separating code which is subject to further modification by BROOK and code that can be passed on unchanged to a later C++-compiler. The BROOK-part of the C-code is then transformed into C++-code, which expands the shorthand declaration in the BROOK code. For example, `float in_stream<256>;` becomes `::brook::stream in_stream(::brook::getStreamType((float*)0),256,-1);`. If we compare the original Program III.4 and the generated Program III.5, we see that the kernels get modified, too, before being passed to the CG compiler. The resulting assembler code is stored as a string in the final C++ code. With some auxiliary code, the whole output is written into a single C++-file.

[9]This limitation to C entails the old-school type casts and the variable declarations at the beginning of blocks which were necessary in the examples.

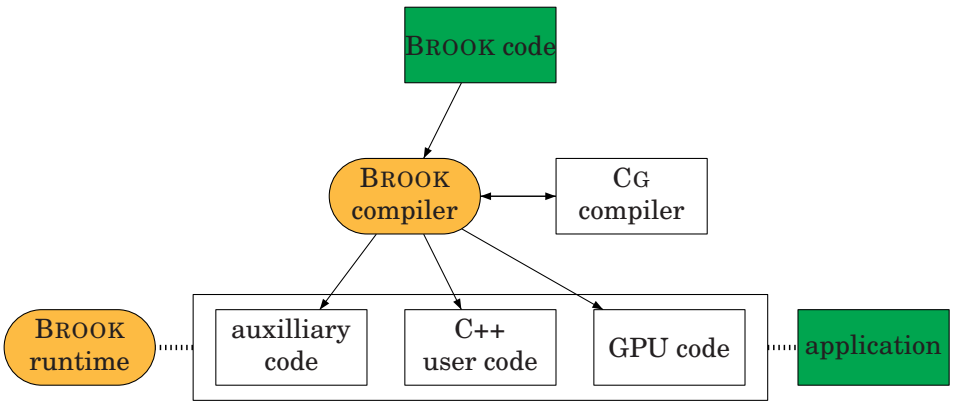
```
kernel void get(float POS_STREAM<>, float INPUT[57],
                out float OUT_STREAM<>){
  OUT_STREAM = INPUT[POS_STREAM.x];
}

int main(int,char**){
  float pos_array[256], out_array[256];
  float pos_stream<256>, out_stream<256>;
  float in_array[57], in_stream<57>;
  int i;

  for(i=0; i<256; ++i) pos_array[i] = (float)((3*i)%57);
  for(i=0; i<57; ++i) in_array[i] = (float)-i;
  streamRead(pos_stream,pos_array);
  streamRead(in_stream,in_array);

  get(pos_stream,in_stream,out_stream);
  streamWrite(out_stream,out_array);
}
```

Program III.7: Random reads in BROOK

This file is then linked to the BROOK runtime system and to the rest of the application. At execution time, the generated functions upload the data to the GPU using the runtime system, direct the execution of the GPU programs and download the data. The GPU is abstracted away completely.

The user can choose among a variety of targets at execution time. Supported are the GPU via both DIRECTX and OPENGL, the CPU and ATI-GPUs via CTM (Section III.4.a). The choice is made by setting an environment variable or by calling a choice function in the library. In any case, the rest of the program is not affected by the choice.

Figure III.2 Working with BROOK



Evaluation

BROOK is, as it will turn out, quite similar to CGIS. As such, several of CGIS' characteristics are present also in BROOK. Differences lie in the capabilities of the abstraction and the compilation (that is, externally visible differences) and in the compilation process (internal differences).

For GPGPU programming, BROOK is a big step forward compared to shading languages. With the targets completely abstracted away, one of the main goals of high-level GPU programming is reached. BROOK has several features to abstract away from the hardware. For example, it is possible to align streams of differing sizes. Also, if the output capabilities of a GPU are too restricted to implement a particular kernel, the compiler issues a multipass rendering, where the targets are written one-by-one. This is a kind of abstraction which would not be possible on CG alone. However, in some cases, the abstraction is not adequate. This entails streams of `int` types, and also the syntax of the kernels itself. Relying on the CG compiler means, on the one hand, to be disburdened from implementation decisions, but it also means that the language syntax is largely fixed. Section IV.2 shows the additional possibilities that present itself when a designer has full control over the input language.

All in all, BROOK today is a very viable choice for GPGPU work.

III.3.b RapidMind

SH [MQP02] is a *metaprogramming language*. Programs are not specified in textual form to be read by a compiler. Instead, a C++ program calls library functions which perform the actions usually done by a compiler frontend, such as generating the abstract syntax tree, building a symbol table or type checking [WM95]. By using the C++ facilities of object hierarchies and operator overloading, the syntax of those function calls is quite close to the syntax one would usually use in writing a program.

The development of SH itself has ceased. In 2004, a newly formed company, *Serious Hack*, took care of SH. Serious Hack has since been renamed to *RapidMind*, and the newer versions of SH are also known as RAPIDMIND. In this work, the name RAPIDMIND will be used for the current state of the system.

Basic Usage

RAPIDMIND is best described by example. Program III.8 specifies a sample program adding the contents of two streams (just like Program III.4). It is instructive to start its discussion with the function `mkkern`, which creates and returns a kernel.

Its use seems obvious from the syntax: A program is defined, enclosed in `RM_BEGIN` and `RM_END` markers, which takes two four-component-float elements as its input, one such element as its output, and adds the input elements into the output element. A program object representing this is created and returned.

But how is this host system code translated into a GPU program? After preprocessing, the code in Program III.9 is created, which sheds some light into how the code is compiled. At execution time of the function `mkkern`, `RM_BEGIN` is a call to a library function creating a new program object. The creation of the object `in_el1` of type `In<Value4f>` (the call of its C++ constructor) actually adds this object to the syntax table of the current program. The expression `in_el1+in_el2` is a call to the function `operator+()` of the object `in_el1` with argument `in_el2`, and this operator function then adds an addition expression to the syntax tree of the program. Likewise action happens for the call to

```
static Program mkkern(){
  Program kernel = RM_BEGIN {
    In<Value4f> in_el1, in_el2;
    Out<Value4f> out_el;
    out_el = in_el1+in_el2;
  } RM_END;
  return kernel;
}

int main(int,char**){
  init();

  Array<1,Value4f> in_stream1(256),in_stream2(256),out_stream(256);

  float* const in_array1 = in_stream1.write_data();
  float* const in_array2 = in_stream2.write_data();
  for(int i=0; i<4*256; ++i) in_array1[i] = 4-i, in_array2[i]=i;

  Program kernel = mkkern();
  out_stream = kernel(in_stream1, in_stream2);

  const float* const out_array = out_stream.read_data();
}
```

Program III.8: A simple RAPIDMIND program

```
Program kernel = ::rapidmind::begin_program("stream"); {{{{{ {
  In<Value4f> in_el1, in_el2;
  Out<Value4f> out_el;
  out_el = in_el1+in_el2;
} }}}}} ::rapidmind::end_program();;
```

Program III.9: Preprocessed code for the kernel from Program III.8

`operator=()`. Then, the program is ended; at this time, the program is completed and the library could compile it into target code.

This shows the main paradigm of RAPIDMIND: *"C++ is effectively a macro language for the RapidMind Development Platform"* [R06]. This peculiarity is responsible for the greatest assets of RAPIDMIND, and also for its problems.

Directing our attention now to the `main` function of Program III.8, we see how to use the kernel. The library is initialised, and then three 1-dimensional streams with `float4` elements are created. The function `write_data` returns a pointer to the contents of the stream which can be used to transfer data to a stream. In effect, the user gets a chunk of memory, and the RAPIDMIND system is then responsible to update the contents of the stream (which is stored, for example, on the GPU) with the user supplied data as soon as it is needed. The kernel is then executed by using it in an assignment operation.[10] The output data are then accessed by getting a pointer to them with a call to the function `read_data`.

[10] It is *not* executed by the expression `kernel(in_stream1, in_stream2)`; see the later explanation of partial evaluation.

Programming Kernels

The usage of C++ operators to create operations in the target program might lead to the wrong assumption that the same holds also for statements. But consider, for example, the statement `for(i=0; i<10; ++i) a=a+b;`. At runtime of the C++-program, this statement would result just in 10 calls to the functions `operator+()` and `operator=()`; in effect, to the library it is indistinguishable from the complete unrolling of the loop on the C++-level. The same holds for other control flow constructs: An `if`-statement is not performed per-kernel on the kernel-data, but once in the data space of the host program. Thus, to the kernel programs, C++ control flow statements are *static, not dynamic*. To express control flow in the kernels, special macros are used again. For example, Program III.10 shows how to express the statement above in terms of kernel-level control flow, and how this is actually preprocessed into functions manipulating the syntax tree accordingly.[11]

```
// Written by the user:
RM_FOR(i=0.0f, i<10.0f, i+=1) {
  a=a+in_el1;
} RM_ENDFOR;

// Actual calls at runtime (simplified):
internal_for(
  push_arg_queue() &&
  push_arg() &&
  process_arg(i=0.0f, 0) &&
  push_arg() &&
  process_arg(i<10.0f, 0) &&
  push_arg() &&
  process_arg(i+=1, 0)
);
{ a=a+in_el1; }
end_for();
```

Program III.10: Control flow in RAPIDMIND

For arithmetic computations, RAPIDMIND offers the usual assortment of types and operations, with familiar caveats; for example, `double` and `int` types are implemented as `float` on GPUs, and CPUs implement half-precision floats as single-precision floats. In addition to the familiar vector-types of up to four components, RAPIDMIND can operate likewise on vector-types of *arbitrary* length; such vectors are then internally split up into smaller vectors which can be handled by the hardware, and operations on such vectors are split accordingly. A library for arithmetical, trigonometrical and other helpful functions common to GPU programmers is also available. These functions are implemented by native instructions where possible and computed by more primitive instructions otherwise. Swizzling and masking are also supported, which have to use an idiosyncratic syntax to fit in the operator-overloading setup.

It is obvious from the examples III.8 and III.9, that inside a kernel also objects declared *outside* of a kernel can be used. This is in fact a meaningful operation: Objects defined outside of a kernel and used inside of it are passed as constant parameters to the ker-

[11] Only the actually executed function calls are displayed. The preprocessed code is larger to accommodate for immediate mode, which is explained later.

nel. In a similar way, random stream lookups can be performed (Program III.11). The runtime system takes care of transferring the input data to the GPU when necessary.

```
Array<1,Value4f> lookup(256);
Value1f constant(5); // Externally defined, thus constant input.
Program kernel = RM_BEGIN {
  In<Value1f> index;
  Out<Value4f> result;
  colour = lookup[index] * constant;
} RM_END;
```

Program III.11: Uniform parameters and random reads in RAPIDMIND

Indeed, the primitive types and operations need not be used solely in kernels. One can declare and use `Value4f` objects anywhere in a program: This is called *immediate mode* of execution, in contrast to the *retained mode* in kernel specifications. Used in this way, RAPIDMIND is a vector/matrix library.

Advanced Features

RAPIDMIND offers other interesting features. For example, it is easily possible to express strided or continuous sub-streams of streams, and the effect of access outside of array bounds can be specified in a way common to GPU programmers. But of particular interest are partial evaluation of programs and shader algebra, both of which operate on the program objects.

Looking back at Program III.8, we see that the kernel had been executed by the line `out_stream = kernel(in_stream1, in_stream2);`. As we recall, the expression `kernel(in_stream1, in_stream2)` is actually a call to the function `operator()()` of a program object. As it happens, this operator function does not actually execute the program, but just *binds a parameter*, returning another program with the bound parameter. This makes possible *partial evaluation*. For example, `kernel(in_stream1)` is a program in which the first parameter is bound to a particular stream. This program can be used in other contexts to bind various streams to the second parameter. Actual *execution* of the program takes place only when the result is actually used.[12]

Shader algebra [MTPCM04] provides a means to combine kernels. For example, suppose there are kernels p and q which use i_p, i_q input streams and write into o_p, o_q output streams. One can combine these two kernels into one kernel s, which performs both computations of p and q independently, reading from $i_p + i_q$ streams and writing into $o_p + o_q$ streams. If $o_p = i_q$, then one can also combine p and q by catenation into a kernel s stemming from the sequential execution of p and q, where q reads the output of q. It is important to note that because the programs are not actually compiled into target code at this point, the compiler might make use of *cross-kernel optimisations*. For example, if, p and q happen to have common code, classical common subexpression elimination can remove this.

RAPIDMIND supplies the user with a library of small kernels called *nibbles* to perform various simple, but useful tasks. For example, a nibble might duplicate an input or discard one of a number of inputs, or it might perform a simple arithmetical operation on its inputs. These nibbles can then be used to compose a larger program much in the same way primitive `NAND` or fan-out gates are used to create electronic circuits.

[12] The shown example of partial evaluation uses *tight binding*, where the stream is bound by value. Binding by reference, *loose binding*, is also possible.

As targets, RAPIDMIND supports GPUs via GLSLANG (Section III.2.c) and the Cell processor. It is possible to explicitly write shaders for one of the GPU's processors to facilitate RAPIDMIND's use in classical graphics processing. The runtime system can transparently chose the target on which to perform a computation, but it can also be specified by the user.

Evaluation

RAPIDMIND is supported by the company RAPIDMIND and accompanied by an excellent documentation.

The meta-programming nature of RAPIDMIND is its biggest asset. The ability to manipulate programs via shader algebras can make for a very elegant way of expressing algorithms. One is reminded of functional programming languages, where the data flow from one part of the algorithm to another is specified by plugging in functions as arguments to other functions and catenating functions in various ways. That the code is compiled after manipulation, offers additional opportunities for optimisation, as mentioned above. Another strong point of RAPIDMIND is the targeting of the Cell processor.

But the very nature that makes RAPIDMIND so expressive is actually quite different to what programmers are used to. Functional programming is certainly a very elegant way of programming, but nearly all programmers are much more familiar with imperative programming using functional abstractions. The embedding is a further hurdle for accessibility, because keeping track of whether one currently programs in immediate mode or in retained mode is not something that programmers are used to. A clearer separation of concerns would be helpful to programmers.

Also, because the programs are compiled during the run time of the application, the usual problems of *dynamic type checking* apply here, too—The C++ compiler can check only so much, and whether the streams which happen to be passed to the kernels have the correct type can in principle not be checked at compile time. Other problems, such as automatic backend selection choosing an incapable backend or some crashes, are almost certainly due to the tested beta version of RAPIDMIND. The full version of RAPIDMIND has just recently been released.

All in all, RAPIDMIND is a beautiful way of programming GPUs, but it is not for the faint of heart.

III.3.c Accelerator

The ACCELERATOR system [TPO06] is specifically designed as a GPGPU language. Technically, it bears most similarities to RAPIDMIND, in that GPU code is specified using operator overloading, this time in C#. The main difference is that the primitive object of operation is not the *element*, but the *stream*.

ACCELERATOR introduces four "parallel array" datatypes to C#: Arrays of floats, float4s, integers and bools. These are the only types allowed in ACCELERATOR algorithms: The other classical vector types such as float3 or multi-component bools are not supported. This holds likewise for struct types. An elementwise computation on these types is specified by using the appropriate operator on the streams. For example, if `A` and `B` are two parallel arrays, then `C=A+B;` creates a new parallel array `C`, which results from the elementwise addition performed on `A` and `B`. ACCELERATOR supports other operations such as reductions, replication or transposition on arrays, which are needed to perform advanced address calculations. For example, a gather operation on a stream `A` is performed by creating an index stream `I` and using a library function for gathering: After

executing `R=Gather(A,I);`, the array `R` is the elementwise result of the lookup operation with `I` in the fixed array `A`.

The code is quite succinct for small examples, but can get complicated in presence of address translations. For example, the code for a blur operation is presented in Program III.12 (slightly modified from [TPO06]). To access the positions adjacent to an element in one direction for the two dimensions[13], the array is shifted. The shifted array is then multiplied by a scalar, which translates to an elementwise multiplication by a constant. The resulting array is then subject to an addition operation with the array `accumulator`, which again is an element-wise operation.

```
FPA image; // The image to be blurred; a FloatParallelArray
FPA accumulator; // The resulting image.
float weights[]; // Some weights used in the computation.
...
for(int i=0; i<weights.Length; ++i){
  int[] shift = new int { 0,i };
  accumulator += PA.Shift(image,shift) * weights[i];
}
for(int i=0; i<weights.Length; ++i){
  int[] shift = new int { i,0 };
  accumulator += PA.Shift(image,shift) * weights[i];
}
```

Program III.12: A blur filter in ACCELERATOR

This is quite a different view of the GPUs. In most abstractions, kernels are specified in terms of elements, and operations are performed componentwise across all element components. In ACCELERATOR, kernels are specified in terms of streams, and operations are performed elementwise across all stream elements.

Technically, ACCELERATOR is compiled during the application's execution time just like RAPIDMIND. ACCELERATOR keeps track of the desired operations and arranges them into kernels which can be performed on a GPU. This compilation is performed whenever an operation needs to be done on a CPU or when data is read back from the parallel arrays into standard C# arrays to be operated upon in standard code.

All in all, ACCELERATOR does indeed completely abstract the GPU and as such is usable for GPGPU. It has some limitations, though. Its confinement to DIRECTX means that it is limited to the Windows operating system. That streams may consist only of a very limited number of primitive types is also unfortunate. Certainly, a 3-ints-stream is not commonly used, but it would not hurt to support it, when one already has float4-arrays. Furthermore, in contrast to RAPIDMIND, ACCELERATOR does not offer kernel level control flow statements. (As mentioned in Section III.3.b, host language level control flow statements are static with respect to the kernel code.) Because of these restrictions, ACCELERATOR is a less useful choice for GPGPU.

[13] The original code does indeed not use negative offsets.

III.4 Other Languages

This section deals with the hardware-specific systems CUDA by NVIDIA and CTM by ATI. They lie in stark contrast to every language considered so far, because they are *not an abstraction of the graphics pipeline*. Instead, they regard the GPU as that as what GPGPU programmers like to see it: Just some processing device. Thus, they form a radically different class of languages than both shading languages and the GPGPU languages investigated so far. However, they are very different from each other, too.

III.4.a CTM

ATI's CTM initiative [A06] follows a completely different approach than that promoted so far for GPGPU. Instead of *abstracting* the GPU, and thus enabling programmers to program the device *as though it were* just an ordinary co-processor, it lays bare the processing elements themselves, and thus presents the GPU *as it is.* Hence stems the name of the system: *Close To the Metal (*CTM*)*. CTM is the name of the main processing device; here, it is used interchangeably also for the language.

The reference manual [A06] describes the various processors of the CTM, the classes of instructions, the instructions themselves and the methods of scheduling and synchronisation. It is, in fact, an assembly language manual for that particular class of GPUs. This should not be confused with the assembly languages in OPENGL or DIRECTX (Section II.2.b): The API-assembly languages are themselves abstractions over the actual operations supported by the hardware. It is the responsibility of the graphics driver to translate the high-level assembly instructions into hardware-specific assembly instructions. For this section, and the rest of this thesis, "assembly language" always means API-assembly language.

Obviously, on this low a level, CTM is unusable as a GPGPU language. The better control over the target carries a large cost in the form of hindrances of lowest-level programming—"Close to the metal", just as advertised.

However, CTM is a very viable choice as a *target language* for high-level GPGPU languages; or at least as *one* of multiple possible target languages, because CTM is specific to ATI. Indeed, BROOK (Section III.3.a) supports CTM as one of its targets. Used in this way, there is to consider a trade-off between CTM and assembly languages: Assembly language programs need to pass through the additional layers of the graphics driver, yet are subject to probably good optimisers translating to CTM (or a language on a similar level); with CTM, the compiler writer has not only the opportunity, but also the *duty* to go down to the very lowest level. In fact, choosing between CTM and assembly language entails answering to mostly the same questions as for choosing between assembly language and a shading language (Section III.2.d).

Even when not used directly as a target language, reading the description of CTM can offer insight into the inner workings of the GPU; its use in higher-level instruction selections and scheduling is doubtful however, because the process of converting an assembly language program to CTM level is not known.

III.4.b CUDA

CUDA (*Compute Unified Device Architecture*) is NVIDIA's GPGPU programming environment [N07a]. Introduced in 2006 much to the surprise of everyone in the field, it also offers unfiltered access to the hardware. In contrast to CTM, it works on a higher

level of abstraction. The general level of programming is about the same as for BROOK (Section III.3.a).

To a programmer, the main difference of CUDA with respect to other GPGPU languages lies in the memory access; in terms of what is *possible*, of what needs to be *considered* and of how it can be *expressed*. The programmer now has knowledge about the memory hierarchy (including caches) and the scheduling of execution threads onto processing units, and he specifies their usages in a relatively high-level language.

Basic Usage

For simplicity, CUDA shall be introduced with a pure streaming example.

```
__global__ static void
 add(float* in_stream1, float* in_stream2, float* out_stream){
  const int pos = threadIdx.x;
  out_stream[pos] = in_stream1[pos]+in_stream2[pos];
}

int main(int,char**){
  float in_array1[256], in_array2[256], out_array[256];
  for(int i=0; i<256; ++i) in_array1[i] = 4-(in_array2[i]=2-i);

  const size_t size = 256*sizeof(float);
  float *in_stream1, *in_stream2, *out_stream;

  cudaMalloc((void**)&in_stream1,size);
  cudaMalloc((void**)&in_stream2,size);
  cudaMalloc((void**)&out_stream,size);
  cudaMemcpy(in_stream1,in_array1,size,cudaMemcpyHostToDevice);
  cudaMemcpy(in_stream2,in_array2,size,cudaMemcpyHostToDevice);

  add<<<1,256,0>>>(in_stream1,in_stream2,out_stream);

  cudaMemcpy(out_array,out_stream,size,cudaMemcpyDeviceToHost);

  cudaFree(in_stream1), cudaFree(in_stream2), cudaFree(out_stream);
}
```

Program III.13: A simple CUDA program

Program III.13 is the equivalent of Programs III.4 and III.8.[14] It does *not* use the advanced features of CUDA, but it serves to illustrate the basic programming model. It can be seen that, just like in BROOK (Section III.3.a), the kernel code is written in specially designated functions together with the main code. A compiler then separates those two parts, working on them independently. In CUDA, the identifier `__global__` designates a function as a GPU kernel, callable from the CPU.

The kernel `add` is written from the viewpoint of a single stream element, yet it does not see only a single element: *All* streams are accessible to it. In the primitive example at

[14] The underlying architecture natively works on scalars instead of vectors. Former architectures worked in a $3+1$ or $2+2$ model: Two instructions of up to the specified operand size could be executed in parallel, but not more instructions, even if they made use of less components in total. Such a consideration is not necessary on the G80. Therefore, the example here uses plain `float` as the basic type.

hand, a kernel needs just to fetch one particular element from each of two streams and write the result into the corresponding position in a third stream. This is being done by using the familiar array syntax with *the complete streams*, using the position as an index.

Looking now at the `main` function, we observe first a few functions to allocate memory on the GPU and to transfer data to and fro. GPU memory is represented in CUDA as C-pointers; it is, of course, not possible for the host program to access directly the memory behind those pointers.

The call to the `add` function passes in the three allocated streams; this is similar to the BROOK model. The call gets three other parameters, however. Here, the first two parameters specify the kernel to be run in $1 \cdot 256$ instances (*threads*). In this case, this corresponds to the size of the streams, but in general it is completely independent from it. Besides the raw number of threads, the parameters also specify a topology, and the third parameter specifies block-shared storage; both features are described below.

To stress the important difference: CUDA does not run multiple instances of a program on corresponding elements of streams—It runs multiple instance of a program on streams. Thus, the CUDA model is much more general than the streaming programming model, where access patterns not corresponding to the streaming model have to be expressed in an unnatural way. Scattering lies totally across the streaming model and is usually not expressible at all, yet there is no problem for CUDA.

Threads and Memory

The distribution of the threads can be viewed from a hardware perspective and from a software perspective. Fortunately, because CUDA has been designed for a specific hardware, these perspectives are not too far apart. Nevertheless, because this chapter is devoted to discussing the programmability as exposed to a programmer, the reader is referred to [N07a] for a detailed discussion about the mappings from the logical features onto the hardware components.

Each independently executing instance of a kernel is a thread. Threads constitute *blocks*. Threads within a common block have access to common *shared memory*, which will henceforth be called *block-shared memory*, to avoid confusion about where it is shared. Also, threads within a block can be synchronised with a barrier synchronisation. The blocks themselves constitute a *grid*. There is no synchronisation or shared memory between blocks in a grid. All threads inside a grid are required to execute the same program (statically speaking).

In Program III.13, the call `add<<<1,256,0>>>` specifies that the threads shall be executed in *1* block comprising *256* threads and *0* bytes of block-shared memory.[15] The predefined variable `threadIdx` used in the `add` kernel holds for each thread its id in its block. Each block is restricted to 512 threads[16]; the number of blocks per grid is restricted to $2^{16 \cdot 3}$, which is no real restriction for all practical purposes.

Program III.14 (taken from the CUDA SDK distribution) gives an example of shared memory usage. An array is held in global memory and sorted by bitonic merge sort [B68]. To remove the need to constantly access global memory, the array to be sorted is stored in block-shared memory. Each thread is responsible for copying the element

[15] This call specifies a linear arrangement of threads in blocks and blocks in grids; a two-dimensional or three-dimensional arrangement were also possible.

[16] For this reason, Program III.13 did not use the full 1024 threads which would have been necessary for full equivalence to the former examples.

```
__device__ inline void
swap(int& a, int&  b){ int tmp = a; a = b; b = tmp; }

__global__ static void bitonicSort(int* values){
  extern __shared__ int shared[]; // Block-shared memory.

  const int pos = threadIdx.x;
  shared[pos] = values[pos]; // Copy into shared memory.
  __syncthreads();

  // Go through all phases of bitonic-merge sort.
  for(int k = 2; k <= 512; k *= 2) {
    for(int j = k/2; j>0; j /= 2) {
      // Compare and maybe swap two elements in shared memory.
      int ixj = pos ^ j;
      if(ixj>pos) {
        if((tid & k) == 0){
          if(shared[pos] > shared[ixj])
            swap(shared[pos], shared[ixj]);
        } else {
          if (shared[pos] < shared[ixj])
            swap(shared[pos], shared[ixj]);
        }
      }
      __syncthreads();
    }
  }

  values[tid] = shared[tid]; // Copy from shared memory.
}
...
// Called by:
bitonicSort<<<1, 512, 512*sizeof(int)>>>(stream);
```

Program III.14: Bitonic merge sort in CUDA

at a particular position into block-shared memory at the beginning of the algorithm, and back to global memory at the end. To ensure that all elements are in place at the beginning and that all swaps of a particular phase have been performed before the next phase begins, a barrier synchronisation mechanism is employed.

The line `extern __shared__ int shared[];` declares the size of the block-shared memory space. The host is responsible for ensuring that enough space for all such arrays is available. Therefore the call specifies the amount of memory to be used by `shared`.[17]

A GPU-local function is declared by the qualifier `__device__`. Similar qualifiers can be used to specify accessibility and visibility of variables for the GPU or CPU.

[17] The amount is not specific to be used by a particular array but the amount is reserved for all. In fact, all arrays start at the *same* address in GPU memory (requiring a bit of address juggling to ensure they keep separated), so a separation could not be enforced anyway.

Kernel Programming

Inside a kernel, most features from C can be used. In particular, pointer juggling and looping are not restricted. Function calls are possible, but no recursion. Data can be declared as usual, but memory allocation and variables with static lifetime are disallowed.

CUDA supports `char`, `short`, `int` and `long` as signed and unsigned integral datatypes and `float`.[18] These types are supported in vectors of maximal length 4. Structs are possible as usual.

CUDA implements a number of the standard library functions of C, as far as they are concerned with arithmetic, and a few functions inherited from GPU usage.

Runtime Library

CUDA supports two API levels. The examples so far have used the CUDA *runtime* level. This is a level of auxiliary C functions using the lower level functions of the CUDA *driver* API. This is because the NVIDIA compiler which separates the GPU code from the CPU code and compiles the GPU code into binary images, outputs C code loading those images with the CUDA runtime API. Using the CUDA driver API, specifying execution configurations and loading program images is a bit more cumbersome, but one does not rely on the generated CPU code.

On a level on top of CUDA proper, NVIDIA provides two libraries for fast-fourier transformation and linear algebra. They are modelled after popular CPU libraries (FFTW and BLAS) for easier transitioning.

It is possible to map certain buffers from OpenGL and DirectX into the CUDA memory space to facilitate CUDA's use for visualisation.

Other Notable Features

Assembly. Since the release of the 1.0 version, the CUDA system is accessible also from an assembly level [N07d]. This is the same level that is the output of the CUDA compiler. For example, Program III.15 shows the assembly language that is visible when inspecting the intermediate files generated for Program III.13.

```
cvt.u32.u16     $r1, %tid.x;           # Compute
mul.lo.u32      $r2, $r1, 4;           #   offset.
ld.param.u32    $r3, %parm_in_stream1; # Fetch
add.u32         $r4, $r3, $r2;         #   first
ld.global.f32   $f1, [$r4+0];          #     element.
ld.param.u32    $r5, %parm_in_stream2; # Fetch
add.u32         $r6, $r5, $r2;         #   second
ld.global.f32   $f2, [$r6+0];          #     element.
add.f32         $f3, $f1, $f2;         # Add elements.
ld.param.u32    $r7, %parm_out_stream; # Store
add.u32         $r8, $r7, $r2;         #  output
st.global.f32   [$r8+0], $f3;          #    element.
```

Program III.15: Assembly program for the kernel from Program III.13

That assembly level is actually also a virtual machine. There is still an translation step from the PTX virtual machine level to the actual GPU instruction set. The PTX system

[18] Internally, the integer operations work on 32-Bit values.

is extremely new and hardly tested with arbitrary inputs, that is, with inputs different from the patterns created by the CUDA compiler.

Debugging. A debugging mode emulates the GPU for a CUDA program. Because debugging facilities on the device itself are not available, a CPU-based emulation with the usual debugging support from IDEs and access to host-base output channels is the best way to catch logical errors in the code.[19]

Programmability. Despite expressing program code on a higher level than the operations of the bare machine, a detailed knowledge of the architecture is necessary for high performance. This applies mostly to the memory distribution and access. The CUDA documentation details the partitioning of shared memory into banks and which access patterns introduce bank conflicts, and it mentions rules for how to make the most of the cache of constants. To get the best possible performance from the target, the memory access patterns of the threads and their arrangement into blocks and has to be taken into account early in the design of the algorithm. Section VI.2.a shows the effects of mapping threads onto blocks on the execution times.

Evaluation

CUDA is the best one can get for a truly general purpose languages for NVIDIA GPUs. It does not spare one from the hardware peculiarities of the target, but offers a high-level abstraction (as high-level as C, anyway). To enable compatibility with future generations of hardware, the assembly code files can be fed to the CUDA system instead of the binary images. Drivers for future GPUs might then translate the code into code for the then-current architecture. Thus, even though CUDA remains tied to the NVIDIA platform, there seems to be no particular need to recompile CUDA code whenever a new generation arrives. In this way, CUDA offers a usability common in CPU languages.

All in all, CUDA has to be considered the best system for GPGPU work on NVIDIA CPUs, because it offers the full flexibility of the hardware while not dragging down the user to the bare machine level.

III.5 Summary and Outlook

In this chapter, we have seen the different languages at the disposal of prospective GPGPU programmers, and we have investigated their peculiarities and usabilities. As was argued, programming in a shading language is not an alternative to be considered. Of the hardware specific languages, CUDA is the best alternative, for it offers more than CTM while demanding less. However, in both cases one is restricted to a particular brand of targets. The GPGPU languages RAPIDMIND, ACCELERATOR and BROOK are viable choices for programmers desiring a high level of abstraction and not being tied to a specific hardware vendor, though ACCELERATOR is tied to a particular operating system. RAPIDMIND is a system with many virtues but stranger than necessary to ordinary programmers. BROOK, however, offers an accessible programming model, and thus it deserves attention by the programmers.

As a language, BROOK leaves not much to be desired. The language CGIS, described in the next chapter, offers a very similar programming model and a similar level of abstraction. The differences between BROOK and CGIS either lie on a technical level or

[19] Obviously, not all kinds of errors can be caught: in presence of a race condition, an emulation need not deliver the same result as a native execution, for example.

are stemming from technical differences. In particular, that CGIS compiles its kernel language down to the assembly language level, but BROOK passes the kernels through to CG, is the origin of many a subtle or obvious difference, as shall be investigated in Chapter IV on CGIS and Chapter V on the CGIS compiler.

IV

CGiS

It is my firm belief that *all* successful languages are grown
and not merely designed from first principles.
B. STROUSTRUP, *The Design and Evolution of C++*, 1994

This chapter is concerned with an explanation of CGiS. This entails describing its syntax and semantics, the usage of the generated code and the runtime facilities. It also implies a discussion of the usability of CGiS, and the fitness for data-parallel GPU programming; the examination concerning this is interspersed with the description of the language, to achieve high spatial locality of the discussion.

Section IV.1 gives a high-level overview of the language, both regarding the design goals and the actual result. Sections IV.2–IV.4 describe the parts of the language with increasing scope, from a single kernel in Section IV.2 over data-parallel computations in Section IV.3 to interfacing with the outside in Section IV.4. Section IV.5 presents a larger program, so that the reader can more easily develop a feeling for the language. Section IV.6 subsumes the argumentation about whether or not CGiS attains its goals, and Section IV.7 sums up and concludes this chapter.

IV.1 Overall Design

CGiS is a high-level, data-parallel stream programming language in the sense of Section II.3.c, specifically designed for general-purpose computations on graphics hardware. The present section is concerned first with a high-level overview of CGiS' goals in Section IV.1.a and a high-level overview of CGiS programs in Section IV.1.b. After this section, the reader should have an idea about *how* CGiS programs look like and, in part, *why* they do so; both aspects are deepened in the next sections.

IV.1.a Design Ideas

I briefly sketch the ideas which have been underlying the design and evolution of CGiS. These ideas are in turn consequences of the grand goals of CGiS as mentioned in the introduction (Chapter I), the ambitions CGiS strives to fulfil in the end:

- CGIS shall be accessible to normal programmers.
- CGIS shall be executable on a wide range of targets.
- CGIS shall efficiently make use of the available resources.

These diffuse goals can be broken down into individual objectives as follows.

- *Familiarity:* The language must follow syntactic and semantic traditions from usual (i. e., imperative) CPU programming languages.
- *Readability:* A program's purpose and working must be understandable from the program text. Information has to be localised to confine effects, in particular in case of parallelism. Terseness has to be sacrificed for verbosity, if necessary.
- *Abstraction:* The target has to be virtualised as much as possible. The user does not need not to know on which of several possible targets a program eventually runs, neither at programming time nor at execution time.
- *Compatibility:* A program should be compilable to a range of targets. Implementation should rely on widely supported standards instead of confining the user to a certain platform.
- *Adaptability:* CGIS must be fit for the future. It must be possible to adapt the compiler to upcoming architectures as easily as reasonably possible.
- *Efficiency:* CGIS programs should run fast; in particular, there should be an incentive to program in CGIS instead of standard CPU code, for appropriate algorithms.
- *Controllability:* Advanced programmers should be able to exploit hardware capabilities by optional features not of relevance to the normal programmer. Also, advanced programmers should be able to direct the compiler's optimisation by providing user knowledge about the application to the compiler.
- *Visualisability:* For GPUs, an integrated visualisation mechanism and the ability to use computed results in other parts of the graphics pipeline leads to CGIS' usability in scientific visualisation.

Obviously, some of these objectives are connected. For example, compatibility and adaptability are two different subaspects of *retargetability*; Readability and familiarity are not the same, but go together; Abstraction is a necessary prerequisite for familiarity, but also plays a role for compatibility. Balancing those *user-oriented* aspects against the more *target-oriented* aspects of efficiency and controllability is an important part in the design of the CGIS system. For example, virtualisation is not always possible. Here, CGIS follows the general guideline that those aspects are virtualised where virtualisation might reasonably help implementing a program which would otherwise be unable to be implemented on a given target, but which the user would still want to use in a high-level language (e. g., relatively many outputs of GPU programs); but it does not provide features for which an emulation would be prohibitively expensive, so that nobody would use them anyway (e. g., recursion).

IV.1.b High-Level Overview

In this section, CGiS is described by an example. It shall be explained how to write a CGiS program and how to use it.

CGiS Programs

A CGiS source file is divided into three parts. Program IV.1 shows the stream addition example; refer to Chapter III for the same example in various other languages.

```
PROGRAM add;

INTERFACE
extern in float4 in_stream1<256>;
extern in float4 in_stream2<256>;
extern out float4 out_stream<256>;

CODE
procedure main(in float4 in1, in float4 in2, out float4 result){
  result = in1 + in2;
}

CONTROL
forall(in1 in in_stream1, in2 in in_stream2, result in out_stream)
  main(in1,in2,result);
```

Program IV.1: A simple CGiS program

Immediately, the three-fold sectioning becomes obvious. The purposes of these sections is as follows.

- The `INTERFACE` section declares the data which are operated upon. It also specifies the interface to the outside world, to the application.
- The `CODE` section declares all kernels.
- The `CONTROL` section declares a sequence of parallel executions, invoking the kernels from `CODE` with data from `INTERFACE`.

So let us investigate these sections in Program IV.1 one by one.

In the *`INTERFACE`* section, three streams are declared. Besides an element-type and a size, they have a *visibility specifier* `extern` and a *data-flow specifier* `in` or `out`.

In the *`CODE`* section, a single kernel called `main` is declared. It works on single elements, which are passed to and fro by parameters. In the body, the desired operation is specified in a standard, C-like way.

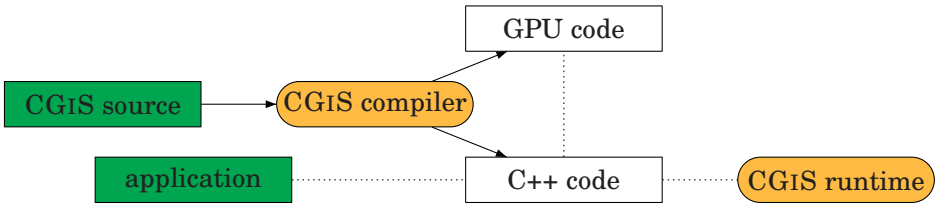
The *`CONTROL`* section consists of a `forall` loop, which specifies a parallel application of a kernel on all elements of a stream. In other words, it is a classical parallel *map* operation. The body of this loop specifies which kernel to execute with arguments of which streams, again by using the familiar call syntax of Pascal or C.

So we have seen how to specify a data-parallel computation on some data. Even before delving into the details, the basic structure of CGiS should be clear. Now we shall see whence come the data and how the computation is performed.

Using CGiS Programs

Figure IV.1 shows the general usage pattern of CGiS. Solid lines denote in- and output, dotted lines denote linkage. The filled rectangular nodes are user supplied code; Program IV.1 corresponds to the "CGiS source" node. The filled oval nodes are part of CGiS, and the other code components are generated by the CGiS compiler.

Figure IV.1 Using CGiS



As can be seen, the main application interacts only with the generated, abstract data interface in the C++ code. Indeed, it is completely invisible on which target the generated program runs. In particular, the application can switch between code compiled for the GPU and code compiled for another target (SIMD-CPUs, see Section V.6.a), or between codes for various generations of GPUs. For our example, interfacing with the generated code would be achieved as in Program IV.2.[1]

We observe the general usage pattern of CGiS:

(1) Initialise the system.

(2) Register the input data.

(3) Execute the program.

(4) Fetch the output data.

(5) Cleanup the system.

It is obvious that this pattern is completely abstracting away the target; in fact, for the SIMD-CPU target (Section V.6.a), exactly the same operations using *the same functions* have to be performed. The only piece of code indicating that the program is to be run on a GPU is during the initialisation process, where a symbolic constant `CGiS_GPU` is sent to the runtime; and this has to be done precisely *because* the rest of the code is oblivious to the target, and the CGiS runtime needs to know which of the possibly multiple targets to initialise.

Without going too much into detail about Programs IV.1 and IV.2, the reader probably agrees that this snippet does not offend the principles of familiarity, readability and abstraction: It is a verbose, but clear way to specify a streaming computation which happens to be performed on a piece of hardware designed to compute colours of triangles. Whether these principles are honoured in general, is a subject of the remaining sections of this chapter.

[1] Program IV.2 corresponds to a part of the "*application*" node in Figure IV.1; specifically to that part forming one end of the line connecting the node to the generated "*C++ code*" node.

```
int main(){
  float4 in_array1[256], in_array2[256], out_array[256];
  ... // Initialise input arrays.

  // Initialise system.
  CGiS_program* const prog = get_CGiS_program(CGiS_GPU);
  register_program_add(prog);
  prog->init_all();
  setup_init_add();

  // Data interface.
  set_data_add(CGiS_add_in_stream1,in_array1);
  set_data_add(CGiS_add_in_stream2,in_array2);
  set_data_add(CGiS_add_out_stream,out_array);
  setup_data_add();

  // Perform computation.
  execute_add();
  get_data_add(CGiS_add_out_stream);

  // Cleanup.
  cleanup_stream();
  delete prog;
}
```

Program IV.2: Usage of the code generated for Program IV.1

Notation

In the following discussion, we shall adopt the following notation.

Fragments of CGiS' grammar describe various portions of the CGiS syntax. To this end, we employ a BNF notation, using the following convention:

- A grammar rule always has the form *NAME* ::= α where *NAME* is a non-terminal symbol and α is formed as a sequence of terminal and non-terminal symbols with a few operations, as explained in this list.
- Nonterminal symbols are *WRITTEN LIKE THIS*, terminal symbols are **written like this**.
- The operators used are:
  - The operator $[\cdot]^?$, specifying that its content is optional.
  - The operator $[\cdot]^+_{\$}$, specifying that its content occurs at least once, but possibly multiple times. Multiple occurrences are separated by a **$**, if this symbol is specified. (For example, $[\alpha]^+_,$ is a comma separated list of α.)
  - The operator $[\cdot]^*_{\$}$, specifying that its content may occur, and if it occurs, it may do so multiple times. Multiple occurrences are separated by a **$**, if this symbol is specified. ($[\cdot]^*$ is a shorthand notation for $[[\cdot]^+]^?$.)
  - The operator $[\cdot_1|\cdot_2]$, which specifies that exactly one of its cases $\cdot_1$ or $\cdot_2$ has to be used. It may have more than two cases: $[\cdot_1|\cdot_2|\ldots|\cdot_n]$ is a shorthand notation

for $[\cdot_1|[\cdot_2|[\ldots|\cdot_n]\ldots]]$. When a complete right side is a list of alternatives, the enclosing [] brackets may be left out.

Note that the grammar serves only to illustrate the syntax of CGiS. In some cases, it has been simplified for ease of exposition, and it includes at times forward references. It is also by far not the actual LALR(1) grammar used in the parser.

In some examples, we shall make use of variables called `vec2` or `vec3a` or the like; these are always variables of basic type `float` and the specified width.

As mentioned before, rationalisation of CGiS' design is interspersed with its description. Such paragraphs are specially marked...

Rationale: ... like this.

IV.2 Sequentialism: Kernels

In this section we investigate the syntax and semantics of kernels. We do so in a purely sequential way, that is, a kernel is considered *in isolation*. It receives inputs from somewhere and produces outputs which go somewhere; this in-/output facility, that is, the interaction with the global data and thereby with other kernels, is covered in Section IV.3. Thus, a kernel operates solely on stream elements. These objects shall henceforth be known as *scalars*: A scalar is anything which is not a stream. (Thus, vectorial values such as `float4` are scalars, whereas a stream consisting of two `float` elements is not.)

In the end of this section, the operations `gather`, `lookup` and `writeback` are considered, in preparation of Section IV.2.d; these are the only statements considered here which can introduce cross-kernel dependencies.

For the most part, the syntax and semantics are very similar to that of C. Thus, there is no formal definition of the semantics. Instead, I list the rules of the language in English prose, appealing to the reader's familiarity with imperative languages in general and C in particular. The differences of CGiS to C will receive special attention, of course, and it is these differences which will account for most of the explanations in the rationale sections.

IV.2.a Types

Predefined Types

CGiS offers floating point, boolean and signed and unsigned integer types in widths of 1–4, as well as user-defined types:

TYPE ::= *ID* | **`int`** | **`int2`** | **`int3`** | **`int4`** | **`uint`** | **`uint2`** | **`uint3`** | **`uint4`**
| **`float`** | **`float2`** | **`float3`** | **`float4`** | **`bool`** | **`bool2`** | **`bool3`** | **`bool4`**

All values of such a type are called scalars, as mentioned before. Values of the types `int`, `uint`, `bool` or `float` are called *primitive (scalars)*, values of the multiple-component integral or floating point types are called *vectorial scalars* or *vectors*. The component type is always called the *base type*: `float` is the base type of `float2`.

Rationale: CGiS does not support pointer types. For lack of dynamically allocated data, memory chunks are always represented as arrays and accessed with numerical indices. It is also not necessary to use pointers to get procedure outputs, because CGiS supports call-by-value-result semantics [S99] (Section IV.2.c).

User-defined Types

Record types can be defined in the `INTERFACE` section with C-like notation:

```
TYPEDEF  ::=  struct { [COMPDEF]+, } ID ;
COMPDEF  ::=  TYPE [ID ]+, ;
```

Rationale: Structs are an extremely useful way of structuring information, in particular when they are used for interfacing with the application. Because CGIS performs texture packing on its own (Section V.5.a), structs can indeed be used in this way, and thus it is worthwhile to include them.[2] Were structs only be used *inside* CGIS programs, their usage would be extremely limited. Indeed, in this case CGIS might not even had supported structs.

Representation

CGIS does not specify a precision or a width for the scalar data types, nor does it specify a relation between the sizes of the types. The following requirements and notes apply, though:

- Integral types can be emulated by floating point types. If so, operations are made on values as though they were floating point values, and afterwards the value is rounded towards $-\infty$. In presence of such an emulation, bit-operations are undefined. If they are not emulated, then signed integral types are represented in a two's complement notation.
- There is no specification about internal precision of floating point types. Also, adherence to the full IEEE standard [I85] or presence of denormalised values or infinities are not required.

In short, CGIS stays close to the hardware's capabilities in representing arithmetical types. Except for the emulation as noted, the values are always represented by their natural counterparts in hardware. The user has to consider this in particular in the case of bit-manipulations (which may not be possible on some targets) and of floating-point computations resulting in values which are not representable (which may result in special values (e. g., NaN or ∞) or may lead to undefined results).

Rationale:

- It is not possible to give more precise statements about representation or accuracy, because the capabilities of the hardware are too diverse. By specifying what is emulated and the manner in which it is, and that everything else is mapped onto their respective hardware counterparts, the programmer however can deduct what to expect from the hardware specification.
- Emulation of bit-operations on integral types with floating point types is somewhere between cumbersome and impossible. In particular, when the precise range of representable values is not known, operations such as rotation could not be implemented at all. That emulation of integral values is present at all is a matter of convenience: It saves the programmer from performing manual truncation operations when he would perform the emulation manually for such values stored in a CGIS`float` variable.

[2] OPENGL allows only textures of primitive types.

Conversion

There are two kinds of implicit conversion between types. On the one hand, primitive values can take on the role of vectorial values in many operations, as defined in Section IV.2.c, by being treated as a vector with the primitive value in all its components. For example, if `vec1` contains a 4 and `vec2` a $[-2, 0]$, then `vec1+vec2` equals $[2, 4]$. This is called *elemental intrinsic overloading* [TL00]. On the other hand, the usual conversions among signed and unsigned integer types and between integral and floating point types are performed, along this line:

$$\texttt{uint} \longrightarrow \texttt{int} \longrightarrow \texttt{float}$$

Such a conversion is always performed with available hardware instructions.[3] Conversion to a specific type can be done by assigning an expression to a variable of that type, as usual. Conversion from `float` towards an integral type rounds towards $-\infty$.

Rationale:

- Arithmetical values are considered different from boolean values. This is in contrast to C/C++: There, an arithmetical value can be used in place of a boolean value (`bool` or `_bool`), where the truth value is determined as $\cdot \neq 0$; and a boolean value can be cast to arithmetical values, where `true` means 1 and `false` means 0. The former convention merely saves very few key-strokes, and the latter is used only very sparingly; both, however, can lead to confusion on the reader's part. Thus, this conversion has been sacrificed for increased readability.
- There are no explicit type conversions. This is simply not necessary: Because the only conversions are among arithmetical types, and because there is no overloading, the necessary conversions are always clear. In particular, there is no need of pointer castings, because CGiS has no pointers. In contrast to the different worlds of boolean and of arithmetical values, a conversion among different arithmetical types cannot lead to confusion, so it is no problem to have it performed in an implicit way.

IV.2.b Scalars

Constants

CGiS supports constants of all predefined types. Vectorial constants are declared by an array-like notation using `[ ]`. A one-component vector $[n]$ is equivalent to n.

```
CONSTANT  ::= BOOLCONST | CONNUM | VECCONST
BOOLCONST ::= true | false
VECCONST  ::= [CONNUM ] |
              [CONNUM ,CONNUM ] |
              [CONNUM ,CONNUM ,CONNUM ] |
              [CONNUM ,CONNUM ,CONNUM ,CONNUM ]
```

[3] The official specification of the G80 architecture does not allow a type-correct transfer between signed and unsigned integral registers [N07c]. However, the same document states that *"in practice"* such a conversion can be done losslessly for representable values (see Section V.3.a). Although the correct functioning is not *guaranteed* by the specification text itself, it can be regarded as a secure conversion.

CONNUM is a constant number, specified as a compile-time constant expression over constant numerals. The numbers are specified as decimal values (in which case they have type `int`), in hexadecimal notation (type `uint`) or in floating point notation (type `float`). The usual type conversion rules are applied. For example, `[1+3,2.0]` is a constant of type `float2`.

Variables

Variables are declared in the usual way with optional initialisers for primitive values or initialiser list for structs.

$$\begin{array}{rcl} \mathit{VARDECL} & ::= & \mathit{TYPE}\ [\mathit{VARDECL1}]^{+}_{,}\ \texttt{;} \\ \mathit{VARDECL1} & ::= & \mathit{ID}\ [\texttt{=}\ \mathit{VARINIT}]^{?} \\ \mathit{VARINIT} & ::= & \mathit{EXPR} \mid \texttt{\{}\ [\mathit{EXPR}\ ]^{+}_{,}\ \texttt{\}} \end{array}$$

CGiS' scoping rules are familiar to C programmers:

- Variable declarations can be placed anywhere a statement could stand.
- The scope of a variable spans from the point of its declaration until the end of the block in which it is declared, where blocks are, as usual, delimited by `{ }` (Section IV.2.d).
- Variables declared in an outer scope are visible in all inner scopes. Declaring a variable with the same name as a visible variable is an error. Thus, shadowing is not possible.

Note that on this level, CGiS has no concept of *global data*. When a programmer desires a kernel to access global data, the data has to be passed explicitly to the kernel as a parameter, effectively creating a local name for it.

Rationale: A main design principle of CGiS is that the scope of a single kernel is limited, and that the interaction with other kernels or parallel executions of the same kernel have to be made explicit. Thus, the need to explicitly create a local name for global data makes it clear to the programmer what side-effects a kernel could have (or its absence of side-effects), and what dependencies would be introduced by the execution of different kernels (or the absence of such dependencies).

Variables with a lifetime longer than the execution time of a kernel (variables declared with the `static` modifier in C) are not supported in CGiS.

Rationale: If a kernel would run only on a particular stream input, static variables could be implemented as a stream of data aligned with the given input. But a problem arises if a kernel is run on streams of different dimensions or size: If a variable has a lifetime spanning from one execution to another, there would have to be a way to create a mapping from kernel executions in one parallel map operation onto kernel executions in another parallel map operation. A sensible way to define this when the shape of the streams is different does not present itself. Irrespective of the implementation, it were a case of local computations with global effects.

IV.2.c Expressions

CGIS features a variety of arithmetical operations. Some of them are represented by operator symbols, others by operator keywords, and others as function symbols. Swizzles are among CGIS' peculiarities.

```
   EXPR  ::=  UNEXP | BINEXP | TEREXP | FUNCEXP | PRIMEXP | ( EXPR )
  UNEXP  ::=  [~| - | not] EXPR
 BINEXP  ::=  EXPR BINOP EXPR
  BINOP  ::=  AROP | LOGOP | RELOP
   AROP  ::=  + | - | * | / | # | \| | ^| & | max | min | << | >> | <<< | >>>
  LOGOP  ::=  and | or
  RELOP  ::=  == | != | < | > | <= | >=
 TEREXP  ::=  EXPR ? EXPR : EXPR
FUNCEXP   ::  PREFUNC ( EXPR )
 PREFUNC  ::=  COMPFUNC | HORFUNC
COMPFUNC  ::=  sin | cos | tan | exp | ln | exp2 | ld | exp | sqrt | abs | flr | frc
 HORFUNC  ::=  hand | hor
 PRIMEXP  ::=  CONSTANT | ID | ID . SWIZZLE
```

An identifier is either a variable name or a structure name using the `.` syntax to access structure components.

Rationale: Function calls are not expressions, but statements (Section IV.2.d). This is because user-defined CGIS procedures do not return values in the way of mathematical functions or functions in certain programming languages, but pass values solely in reference parameters.

Table IV.1 shows the precedence of the operators.

Table IV.1 Precedences in CGIS

(highest)

Swizzles, functions, unary -

```
* / # \
  + -
<< >> <<< >>>
  < <= => >
   == !=
  max min
   not ~
     &
     ^
     |
   and
    or
    ?:
```

(lowest)

Operators

Table IV.2 CGIS operators

Operator	Source	Comp.?	Target
`+ - * / max min`	arith	✓	arith
`| & ^~`	integral	✓	integral
`#`	`float3`		`float3`
`\`	arith		arith1
`<< >> <<< >>>`	integral + uint	✓	integral
`and or not`	logical	✓	logical
`< > != == <= >=`	arith	✓	logical
`?:`	`bool` +any		any

Most operations work on values of all possible width, and their result is the component-wise application. For example, `[1.0,2.0] / [4.0, 0.5]` would produce the value `[0.25,4.0]`. This holds for those operators which are marked with a ✓ in the column "Comp.?" in Table IV.2, which also specifies the operand and target types. The meaning of the type specifiers is given in Table IV.3.

Table IV.3 Legend for Tables IV.2 and IV.4

arith	Integral and floating point types of all widths are allowed, or the same type is produced.
integral	Integral types of all widths are allowed, or the same type is produced.
floating	Floating point types of all widths are allowed, or the same type is produced.
logical	Logical types of all widths are allowed, or a logical type with the same width as the input type is produced.
arith1 (\ only)	The dot product produces an arithmetical type of the same kind as the input and with width 1.
`bool` +any, any (`?:` only)	The ternary conditional operator gets a boolean condition as an input and any kind of operands as the choice values (using standard type conversion). The result is one of the choice values.
integral+uint	The binary operators get one integral type of any width and one `uint` as inputs.
unsigned arith (`abs` only)	The same type as the input type is produced, except that integers become unsigned integers.
boolean	Any boolean value is allowed as input.

Most operands are familiar from mathematical notation or from standard programming languages, though a few deserve explanation.

- The right-shift operator >> performs an arithmetic shift, that is, for signed integer types it shifts in copies of the left-most bit. In addition to the shift operators, CGIS supports bit rotation operators <<< and >>>.
- # performs an arithmetic cross product between vectors:

$$[a_x, a_y, a_z]\#[b_x, b_y, b_z] = [a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x].$$

\ performs an inner product between vectors, e. g:

$$[a_x, a_y, a_z] \backslash [b_x, b_y, b_z] = a_x b_x + a_y b_y + a_z b_z.$$

It degenerates to a simple multiplication for values of width 1.

Rationale:

- CGiS uses the names `and` and `or` instead of && and || more familiar to C programmers to signify that they may be evaluated fully. This does not make a difference in the *semantics*, because the operands of such operators cannot have side-effects (all side-effects are confined to statements in CGiS). For uniformity, the logical not-operator has then also be named `not` instead of `!`.
- Cross product and dot product are useful in geometric transformations and therefore familiar to GPU programmers. They are also, in part, supported in hardware. Rotation operators are supported on some CPU targets [F06] and are useful for certain algorithms (see the RC5 example in Section VI.2.f). Their exposure in CGiS alleviates the programmer from simulating them with shifts and bit-masks, and alleviates the compiler from reconstructing the intent from that code [FLW07].

Predefined Functions

For certain arithmetic operations, CGiS features predefined operators which are used in a way similar to function calls in C. Table IV.4 shows their types; see Table IV.3 for an explanation of the types. Again, most functions are familiar, but a few deserve explanation.

- The functions `hand` and `hor` are *horizontal* boolean operations, computing a sort of inner product with $\wedge$ respectively $\vee$.
- The functions `exp2` and `ld` compute exponentiation of respectively logarithm to base 2.

Table IV.4 CGiS functions

Operator	Source	Comp.?	Target
`sin cos tan exp exp2 ln ld sqrt flr frc`	floating	✓	floating
`abs`	arith	✓	unsigned arith
`hand hor`	boolean		`bool`

Rationale:

- One could add a plethora of trigonometric functions as reserved words to CGiS; it was a conscious design choice *not* to do so. The reason is that GPUs do not offer other functions as native operations. Functions have to be approximated, for example by the first few terms of the Taylor series. But the desired precision of the approximation depends highly on the application; therefore, it is likely that for different applications with different ranges of input the user would desire different implementations, such as an approximation with more or less summands or a lookup table. Therefore, sample implementations are better provided as a CGiS library, where they *can* be used, but need not, instead of as predefined names cluttering up the namespace.

- Exponentiation and logarithm to base 2 are available because all GPU generations support this. GPUs do not support `exp` or `ln` functions, but these come up too often and are derived too easily from `ld` and `exp2` ($\ln a = \log_2 a \cdot \ln 2$, $e^a = 2^{a/\ln 2}$) not to include them as keywords.

Swizzles

CGIS supports the *masks* and *swizzles* from the GPU world (Section II.3.a). Their semantics are similar to that of GPUs, but not equal. Masks are considered in Section IV.2.d.

Swizzling allows the user to construct a new value from the components of an existing value. The components are specified using either the `xyzw`- or the `rgba`-syntax. The resulting values have the same base type as the operand, and the width as specified by the components. For example, suppose the contents of `vec3` are $[1, 2, 3]$.

- `vec3.x` is a `float` with value 1.
- `vec3.yy` is a `float2` with value $[2, 2]$.
- `vec3.w` is not valid, because `vec3` does not have a `w`-component.
- `[0,4].y` is not valid, because constants cannot be swizzled.
- `vec3.rggb` is a `float4` with value $[1, 2, 2, 3]$.
- `vec3.rx` is not valid, because the component notations have been mixed.
- `vec3.xyzzy` is not valid, because CGIS does not support vectors with a width larger than 4.
- `vec1.x` is not valid, because `vec1` is a primitive scalar.

Rationale: This syntax lies in contrast to GPUs, which always require a swizzle to have width 1 or width 4. This is unintuitive; thus, CGIS models swizzling as a general way for arbitrary selection, rearrangement and replication. The GPU programmer has to take special care only in the case of a swizzle with width 2 or 3, because for width 1 and 4 the semantics happen to be the same as in GPUs. Constants cannot be swizzled because the programmer can as well write them in the correct order.

IV.2.d Statements

CGIS supports the following statements. For the most part, their semantics are as usual; explanations will concentrate solely on CGIS' idiosyncracies.

```
   STMT  :=  CALL | COND | LOOP | BLOCK | COMMASTMT | DATAACCESS
   CALL  ::= ID ( [ ID ]+, ) ;
   COND  ::= if ( EXPR ) STMT [ else STMT ]?
   LOOP  ::= FOR | WHILE | DO
    FOR  ::= for ( EXPR ; EXPR ; EXPR ) STMT
  WHILE  ::= while ( EXPR ) STMT
     DO  ::= do STMT while ( EXPR )
  BLOCK  ::= { [ STMT ]* }
COMMASTMT ::= [ COMMASTMT , ASSIGN | ASSIGN ] ;
 ASSIGN  ::= ID [[.|:] MASK]? [= | += | -= | *= | /=] EXPR
```

Masking

Masking allows the user to guard some components of an assignment target. This can be done either statically, using a component mask, or dynamically, using a boolean value. These operations have slightly different semantics.

First, we turn to *statical* masking. This is specified in the same way as swizzling, but components must not be present multiple times, and the order must be preserved. The effect of a statical mask is to change the lvalue and its type to a subvalue.

- `vec3.xz = vec2;` is a valid assignment, updating the `x`- and `z`-components of `vec3` while leaving invariant the `y`-component.
- `vec2.x = vec3;` is an invalid assignment, because a `float3` cannot be assigned to a `float`.
- `vec3.xy = f;` is a valid assignment, because a scalar gets automatically promoted to a vector of the necessary size (here, width 2).
- `vec3.zz = vec2;` is an invalid assignment, because replications are not allowed in masks.
- `vec3.zx = vec2;` is an invalid assignment, because reordering is not allowed in masks.
- `vec4.xy = vec4.yx;` exchanges the first two components of `vec4`.
- `vec1a.x = vec1b;` is not valid, because `vec1a` is a primitive scalar.

Rationale:

- Masking uses nearly the same syntax as swizzling, because this makes the concept easier to understand. On GPUs, masking is not an operation on a value, but a property of the assignment: The type of the left-hand side stays the same, just some components are blocked in being written (Section II.3.a). The change was done to make masking easier to read: The length of the mask locally determines the width of the type of the lvalue and thus of the possible rvalues.
- Masking does not allow replication for the obvious reason that it would be unclear which value gets written eventually. Masking does not allow reordering, because this might confuse the user, in particular those used to the GPU assembly semantics; and in contrast to the change in typing to the GPU world, reordering can easily be recreated by swizzling the right-hand side.

Dynamical masking is a form of guarded assignment. Instead of a fixed mask, a boolean vector (a *guard*) is appended to the target variable. A result is passed through exactly for those components for which the component in the guard is `true`. This form of masking does *not* change the type of the target, because it is dynamic: Dynamic masking is an *attribute of an operation*. For this reason, it uses a very similar, but not exactly equal syntax as static masking. Again, let us turn to a few examples, using `b3`, a vector of type `bool3` with content [`true`,`false`,`true`].

- `vec3a:b3 = vec3b;` is a valid assignment, updating the `x`- and `z`-components of `vec3a` by the `x`- and `z`-components of `vec3b`.

- `vec3:b3 = vec2;` is not a valid assignment, because the static types of `vec3` and `vec2` do not match.[4]
- `vec2:b3 = vec2;` is not a valid assignment, because the static types of `vec2` and `b3` do not match.
- `vec3:b3 = vec2.xyx;` is a valid assignment, updating `x`- and `z`-components of `vec3` with the value of `vec2.x`.
- `vec1a:b = vec1b;` for a `boolb` is a valid assignment, because this is simply `if(b) vec1a = vec1b;`.[5]

Both cases of masking are only allowed in =-assignments.

Rationale:

- In the case of dynamical masking, the semantics of this in an operator-assignment such as += would not be obvious; thus, it is not allowed. In case of a static mask, it is also interpreted as a swizzle, i. e., `a.xy += vec2;` is `a.xy = a.xy+vec2;`.
- Three different mechanisms use a very similar syntax: Component selection in structs, swizzling and masking. For static masks, swizzling and component selection, the `.`-syntax has long since been established. Except in the rare cases of a user declaring a struct with a component `x`, where a `.x` modifier may look like struct component selection or swizzling/masking, it is obvious what kind of operation is specified; therefore, there is no harm in using the established syntax (albeit, as mentioned before, the semantics are slightly different).

  Things are not so clear for the dynamic masks, and I have pondered several other possibilities. As it is a property of the assignment, the mask could be appended to the assignment operator: `vec3a =.b3 vec3b;`. This, however, looks very ugly, and it is just a strange syntactic feature for a language to have subscripts on operator symbols. Thus, this method was discarded. CG uses the `?:`-operator for component-wise selection: `vec3a = (b3)?vec3b:vec3a;`. In CGIS, the semantics of the `?:`-operator is more like that of the `if`-statement, which takes only a single boolean value for a *common* choice.

  A more viable way of expressing dynamic masking would be a guard syntax on the whole assignment: `[b3] vec3a = vec3b;`. However, this conflicts with another syntactic feature (in this case, vector constants also using square brackets), and whatever symbol one might use for guarding, confusion might not be much less than when using the masking syntax. The `.`-syntax of static masking might be even worse, for the semantics are different for equal syntax. Also, were the `.`-syntax used, a boolean vector called `xyz` or the like would introduce ambiguities in the grammar. So, in the end I decided on the `:`-syntax, to emphasise the parallels to static masking and yet contrast it. This is also exemplified by the name “dynamic masking”, in contrast to “guarding”.

Control Flow

The control flow statements of CGIS are pretty familiar to anyone used to C. The `for` loop also allows variable declarations in the initialiser part, which are regarded as in the scope of the loop. The only peculiarity of control flow in CGIS is that procedure calls

[4] Compare with the valid `vec3.xz = vec2;`.
[5] Compare with the invalid `vec1a.x = vec1b;`.

are only possible as statements, not as expressions. Also, because of lack of hardware support, CGiS does not support recursive calls.

Section V.4.c explains how the CGiS compiler copes with control flow in CGiS programs on hardware not supporting such.

Random Reads and Writes

CGiS supports the following explicit data access statements inside kernels. Recall that we assume for now to already have declared the kernel parameters.

```
DATACCESS  ::=  GATHER | LOOKUP | WRITEBACK
   GATHER  ::=  gather ID : [RELPOS]+, ;
   LOOKUP  ::=  lookup ID : [ABSPOS]+, ;
WRITEBACK  ::=  writeback ID : [ABSPOS]+, ;
   RELPOS  ::=  ID < INTCONST [, INTCONST]? >
   ABSPOS  ::=  ID < ID [, ID ]? >
```

They are subject to two placement constraints. First, the statements inside a procedure always have to follow a specific order: `gather` before `lookup` before other statements before `writeback`. Second, `gather` can only occur in procedures called from the `CONTROL` section, not in procedures called from other procedures.

Their semantics is as follows.

- Random reads can be performed with `lookup` statements. A statement of the form `lookup S: a<p1>, b<p2>;` performs two lookups into stream `S` at positions `p1` and `p2` and stores the results in variables `a` respectively `b`. `S`, `p1` and `p2` have to be procedure parameters, because no other values could have been computed by then.
- Random writes can be performed with `writeback` statements. Their semantics is completely dual to `lookup` statements, except for the fact that the positions need not be procedure parameters. Thus, the effect of a statement of the form `writeback S: a<p1>, b<p2>;` is to write the values of `a` and `b` into stream `S` at positions `p1` respectively `p2`. If `p1` equals `p2`, the behaviour is undefined.
- The semantics of `gather` will be explained in Section IV.3.a, for it is necessary to know the iteration paradigm of CGiS to understand `gather`.

Section IV.3 also explains how random accesses can be allowed in the presence of parallel execution.

Rationale:

- Free gathers (reads from arbitrary locations) and scatters (writes into arbitrary locations) are very useful operations in many algorithms. Unfortunately, GPUs cannot perform scatters at the level targeted by CGiS. However, the CUDA approach shows that GPUs can, in principle, support scatters; thus, the language CGiS provides the `writeback` statement. Also, SIMD-CPUs, the other target of CGiS, do not have a problem with scatters.
- The requirement on the ordering is merely a help to the user to structure the algorithm more along the streaming paradigm. Internally, the `lookup` and `writeback` operations of called procedures are inlined, and do end up among other statements.

IV.3 Parallelism: Maps

Section IV.2 described the kernels. In this section, we regard the kernels as given and consider the computation of a multitude of kernels. This entails specifying the sections INTERFACE and CONTROL, and the kernel declaration syntax inside the CODE section. Again, the semantics is given, for the most part, in plain prose. But specifying exactly when reads into streams occur and when contents are considered to be updated might be confusing, especially to a programmer not used to parallel computations with concurring accesses to common data. Thus, a formal definition of data accesses by an implicit sequentialisation is essential for a well-defined and understandable parallel semantics, and the end of this section presents such a semantics.

IV.3.a Streaming Computations

At first, we concentrate on plain streaming computations. Recall from Program IV.1 that streams are declared in the INTERFACE section and that a `forall` loop in the CONTROL section specifies the parallel operation on elements of streams. The following grammar specifies the INTERFACE section. A few symbols are not expanded into their productions here, because they are not used until later parts of this section.

```
INTERFACE  ::= INTERFACE [GLOBAL]+
    GLOBAL ::= GSTREAM | GSCALAR | TYPEDEF
   GSTREAM ::= ACCESS TYPE ID STREAMSIZE [STREAMSPEC]? ;
STREAMSIZE ::= < [ID |INTCONST] [, [ID |INTCONST]]? >
   GSCALAR ::= ACCESS TYPE ID ;
    ACCESS ::= intern | extern FLOWSPEC
  FLOWSPEC ::= in | out | inout
```

The following grammar shows how to use these global data in the CONTROL section; again, some symbols are expanded later.

```
   CONTROL ::= CONTROL [CONTSTMT]+
  CONTSTMT ::= FORALL | SHOW | INTRINSIC
    FORALL ::= forall ( [ID in ID ]+, ) PARBODY
   PARBODY ::= CALL | { [CALL]+ }
```

And finally, we turn our attention to the declaration of kernels in the CODE section.

```
      CODE ::= CODE [[ PROC | TEMPLATE | INSTANCE ]]*
      PROC ::= procedure ID ( [PARDECL]+, ) { [STMT ]* }
   PARDECL ::= REDFLOW TYPE ID [STREAMPAR]?
   REDFLOW ::= FLOWSPEC | reduce
 STREAMPAR ::= < _ [, _]? >
```

The general usage of CGiS for streaming computations is as follows:

(1) Declare the streams in the INTERFACE section.

(2) Write a `forall` loop in the CONTROL section, declaring *iterators* on those streams.

(3) Write a procedure call to a procedure in the CODE section, passing it those iterators.

The net effect of this is that the procedure specified in the call can be executed in parallel on all elements of the corresponding streams. Armed with this grammar fragment, Program IV.1 becomes even more clear.

Let us look at the parts of the specification one-by-one, starting from the declarations of the global data. Data are either `extern` or `intern`. This concerns their *visibility* to the application, not to the CGiS program: `intern` data are completely local to the CGiS program, `extern` data are somehow also used by the application. The exact manner of this usage depends on the *flow* specifier: Data with `in` flow is set by the application, data with `out` flow is read by the application, for `inout`, both holds.

Notation: Unless explicitly stated otherwise, when this text speaks of "*in/out flow*", it means any flow comprising the `in`/`out` flow; that is, anything mentioned about `in`/`out` flow also holds for `inout` flow.

The flow modifier on global data also governs accessibility of the CGiS program: Data without `out` flow cannot be written, data without `in` flow cannot be read. Data with visibility `intern` can be written and read. When passing an iterator to a procedure, the flows must match: An iterator can be passed to a parameter with `in` flow only if the stream itself has `in` flow.

One can pass components of streams of structs again with the `.` component selectors. This allows the programmer to switch freely between a stream of structs and a struct of streams. Program IV.3 shows an example for this: The stream `Points` has elements of type `point_t`. To the procedure `main`, only the `coord`-components of these elements get passed. Thus, `Points` is a stream of structs as visible to the outside and the `CONTROL` section, but it can be accessed as though it were a struct of streams.

```
PROGRAM struct_test;

INTERFACE
struct { float cx, cy; } coord_t;
struct { coord_t coord; float3 value; } point_t;

extern in point_t unipoint;
extern in point_t Points<16>;
extern out float floats<16>;

CODE
procedure main(in coord_t c1, in coord_t c2, out float f){
  f = c1.cx*(c2.cy+1.0+c2.cx)+c1.cy;
}

CONTROL
forall(a in Points, b in floats)
  main(a.coord, unipoint.coord, b);
```

Program IV.3: Stream of structs and struct of streams in CGiS

Iterators are the most natural parameter passing mechanism to streaming kernels; in fact, it is the *defining characteristic*. But in addition to iterators, other values can be passed to the procedures in CGiS. Let us look first at scalar parameters.

- The *index* operators can be specified on iterators in a call. They pass to the kernels the index of the current element, starting from 0. That is, if a stream with size 256

gets iterated upon by iterator `i`, the parameter `indexX(i)` passes to the kernel responsible for computing at position ι a ι, for $\iota \in \{0, \ldots, 255\}$. The equivalently defined operators `indexY` and `indexXY` are only allowed for iterators iterating over two-dimensional streams.

- Scalar values from the `INTERFACE` section can also be passed to kernels. This mechanism, which amounts to *uniform* parameters in some other GPU languages, can be used for parameterisation, and in general whenever some input data is constant across all stream elements. Like in streams, the programmer can pass selected components of structs (Program IV.3).

In both cases, the procedure parameter must not have `out` flow.

Rationale:

- The iterators are syntactic sugar: One could equally well just use the stream as the parameter of a call as in BROOK (Program III.4). From the parameter of the called procedure it can be inferred whether a stream shall be passed or an iteration is desired. But it has to be *inferred*, and that goes against the principle of readability. With the explicit iterator syntax, it becomes *obvious*.
- Global scalar parameters have to be explicitly passed to the kernels to localise a computation's input state, again.

With the iterator mechanism covered, the semantics of the `gather` operations can be explained easily. Let `i` be an iterator to stream `S` passed to parameter `p` of a kernel. Then, the statement `gather p: shifted<x,y>;` declares the input `shifted` to be read from the stream `S`. The position is determined by the position of `i` (the result that would be returned by the index operators) with offsets `x` and `y`. Thus, gathering with offset 0 or $(0, 0)$ is disallowed. (The effects of gathers to out-of-bounds indices are explained later.)

For example, in Program IV.1, the procedure `main` receives a parameter `in1` which is passed from an iterator `in1` iterating over `in_stream1`. Two `gather` statements with offsets `-1` and `1` would declare fetches to the neighbouring two elements of `in_stream1`.

Note that at the declaration of the procedure, scalar parameters are just some parameters with a flow specification; it matters not whether they come from an iterator, a global scalar or an index operator. In fact, the same procedure can be called multiple times with different kinds of values passed to the same parameters. The only restriction is that `gather` can be performed only on parameters being called with iterators.

Rationale: CGIS uses a uniform syntax for vastly different kinds of inputs (uniform scalars, stream elements and index parameters) because, to the *kernel*, these elements actually are the same: Scalar parameters with a certain type and a certain data-flow specifier. The only difference is that only stream element parameters can be subject to `gather` operations. It is the compiler's duty to take care of the different meanings in different contexts.

Kernels can also be passed streams. These are used for table lookups and for writeback operations. The stream parameters have no explicit size, just an explicit dimension. They can get passed any stream of the correct dimension. Program IV.4 shows this in a small example. Regarding the flow, the same conditions as for iterators apply: `lookup` can be performed only on streams with `in` flow, `writeback` only on streams with `out` flow.

```
PROGRAM table;

INTERFACE
extern in float values<256>;
extern in float indices<16>;
extern out float results<16>;

CODE
procedure table_lookup(in float whole<_>,
                       in float position, out float value){
  lookup whole: desired<position>;
  value = abs(desired);
}

CONTROL
forall(s in indices, r in results)
  table_lookup(values,s,r);
```

Program IV.4: Lookup in CGiS

To ensure a correct semantics of kernels in presence of multiple kinds of reads and writes, the following requirements and specifications are made.

- A stream must not be iterated upon more than once in one `forall` loop. It also must not get passed more than once as a stream parameter in the same procedure call.
- A stream which is iterated upon can be passed as a stream parameter. If so, all reads from the stream via `gather`, `lookup` or through `in` iterators occur before all writes via `writeback` or `out` iterators. A stream which gets passed as an `out` stream, that is, which might be the subject of a `writeback`, must not be iterated upon with an iterator that gets passed to an `out` parameter.
- When a stream does not get passed as a stream, all reads via the iterator or `gather` occur before all writes via the iterator.
- Multiple writes to the same location of a stream with `writeback` result in undefined behaviour.

Rationale: One could adopt the convention that the result of multiple iterations or other practicalities resulting in the potential of multiple writes is *implementation-dependent* or *undefined*, understanding that a programmer might know how on a particular target the accesses happen to get scheduled. This would not be useful in the long run, however, because one has no control over the sequences of data access and the scheduling of the GPU kernels in general. It might be the case that on a certain configuration of hardware and driver race conditions resolve in a predictable way, but in the very next driver version the situation may be different again. Thus, CGiS does not allow this at all.

Some other notes about calling streaming kernels:

- There is no type conversion at procedure calls: The types of the variables in the call and the parameters have to match exactly.

- To enable correct correlation between elements of various kernels, the sizes and dimensions of all streams used in the same `foralll` loop must be equal. If the sizes are not statically specified, then the runtime systems checks for this (Section IV.4).
- It is possible to have multiple calls in the body of a `foralll` loop. This is equivalent to a sequence of `foralll` loops, each having one of the calls.
- Reads outside the bounds of streams are allowed and result in clamping to the borders. For example, consider a stream with dimension 1 and size 256. A `lookup` at position $p < 0$ would always return the value at position 0, and a lookup at $p > 255$ would return the value at position 255. The same holds for a `gather` with out-of-bound accesses.

Rationale: Having a fixed semantics for out-of-bounds accesses is highly desirable. In particular, whereas for `lookup` operations the programmer can check dynamically in the program whether an access would lie outside the range of the stream, this is not possible for `gather`. In fact, *any* `gather` operation specifies an out-of-bounds access for at least one invocation of the kernel. Clamping mode provides the most easily understandable such specification. It conceptually comes for free on GPUs:[6] This behaviour is one possible behaviour of specifying out-of-bounds accesses to textures.[7] Another alternative would be *wrapping*: For a stream with size 256, an access to position p would be performed at position $p \bmod 256$. I decided against this semantics: By the very nature of gathering operations, the user wants to access the *neighbourhood* of an element; a model which is better covered by the clamping semantics. To get the wrapping behaviour in a `lookup`, the operation mod can easily be performed by the user himself.

IV.3.b Reduction

A *reduction* operation performs a specific binary operation on all elements of a stream. Thus, a stream gets *reduced* to a scalar; hence the name. Reduction corresponds to fold operations in functional languages [T99]. For example, Program IV.5 shows a reduction operation: All elements of a stream are reduced by the add operation into a scalar, computing the sum of all elements.

That a kernel specifies a reduction operation, is specified by the presence of a parameter declared with `reduce` instead of a normal flow specification. This is a parameter representing the temporary value of the sum. The call to the reduction procedure uses the familiar iterator syntax to pass iterators of the stream which is to be reduced to the reduction procedure. The procedure itself works only on the particular element, that is, each kernel conceptually is responsible only for updating the part of the reduction corresponding to its particular element.

Reduction is implemented in a parallel divide-and-conquer strategy (see also Figure V.2 in Section V.1.c). For example, to sum up the elements in a stream of 16 elements, one first performs 8 parallel reductions, each computing the sum of 2 elements. This results in a stream of 8 elements, which is subjected to 4 parallel reductions. The next step

[6] The clamping has to be performed *somewhere* inside the system, so it certainly adds some computational burden. It is unclear where and how the clamping is performed, and it can be assumed that the implementation is specialised and efficient. The CUDA backend (Section III.4.b) does not benefit from the automatic clamping and therefore has to create appropriate instructions in the kernel for itself.

[7] It does, of course, *not* come free on CPUs; in fact, a naïve implementation which performs security checks for *every* access can make the performance go downhill pretty fast. But to not look out for out-of-bounds accesses would lead to immediate disaster. And *if* one does a check anyway, it is not much more expensive to provide a graceful, specified semantics instead of securely returning an arbitrary value.

```
PROGRAM compute_sum;

INTERFACE
extern in float to_reduce2d<SIZEX,SIZEY>;
extern out float reduced;

CODE
procedure sum_up(in float data, reduce float s){
  s = s+data;
}

CONTROL
forall(source in to_reduce2d)
  sum_up(source,reduced);
```

Program IV.5: Reduction in CGiS

performs two reductions, and the last step returns the final value. For this operation to produce the same value as a sequential fold, the reduction procedure has to be associative. It is impossible to check this in a compiler; the user is solely responsible for assuring associativity.

Rationale: Reduction procedures and streaming procedures specify two vastly differing models of computation, yet they are programmed using nearly the same syntax. This is possible and useful, because the *scope* of each kernel is the same in each case, namely one element.

Reduction procedures are severely restricted: They get passed only the iterator to the reduction stream and the reduction parameter, and they can perform only basic operations on the data. The programmer has to ensure that the operation is associative and that there are no persisting manipulations on the iterator value.

For example, a reduction as in Program IV.6 is allowed: In this example, a `float4` is interpreted as a 2×2-matrix, and the reduction computes a matrix multiplication.

IV.3.c Special Directives

CGiS supports *intrinsics*, which are special functions used *on streams*, that is, in the `CONTROL` section. On the one hand, a `show` statement allows for easy visualisation. On the other hand, special statements for multiplication of matrices with matrices or vectors cater to this need, which comes up quite often in numerical application

The following grammar shows the productions for the up to now unspecified nonterminal symbols.

```
    SHOW  ::=  show ( ID ) ;
INTRINSIC ::=  [ matvecmul | matmatmul ] ( ID , ID , ID )
STREAMSPEC ::= : [ POSSPEC | MVSPEC ]
 POSSPEC  ::=  [packing ( INTCONST )]? POSITION
  MVSPEC  ::=  matrix | vector
```

```
PROGRAM red_matmul;

INTERFACE
extern in float4 to_reduce<16>;
extern out float4 reduced;

CODE
procedure matmul(in float4 data, reduce float4 s){
  float2 line1 = data.xy, line2 = data.zw;
  float2 column1 = s.xz, column2 = s.yw;

  float4 new_matrix;
  new_matrix.x = line1 \ column1, new_matrix.y = line1 \ column2;
  new_matrix.z = line2 \ column1, new_matrix.w = line2 \ column2;
  s = new_matrix;
}

CONTROL
forall(I in to_reduce) matmul(I,reduced);
```

Program IV.6: Another reduction in CGIS

Visualisation

With the statement `show(stream);`, the contents of a stream can be displayed on the screen. The elements of the stream are treated as pixel colours for this operation and are displayed on a black background. For example, consider a simulation of the game of life [G70] (see also Section VI.2.b): Program IV.7. In a `forall` loop, the next simulation state is computed in parallel across all cells. The state is held in a numerical stream, where each cell is represented as a 1 (living) or 0 (dead). The procedure `iterate` (not shown in the example) computes for each cell the number of its neighbours and whether it itself should be alive in the next state (be born/stay alive) or dead (die/remain dead).

```
PROGRAM life;

INTERFACE
extern inout float field<512,1024> : B;

CODE
procedure iterate(inout float element, in float2 index)
  ... // Body omitted.

CONTROL
forall(float e in field) iterate(e,indexXY(e));
show(field);
```

Program IV.7: Game of Life in CGIS

Note that the declaration of the state stream `field` includes a *packing specifier* (the *POSITION* in the grammar). For visualisation, CGIS needs to know which colours to use. In this case, the specification states that the state should be visualised with blue colour; the colour specification follows the general `RGB` pattern for red, green and blue.

The background colour is fixed at black. There may be only one `show` statement in a single program.

Matrix Algebra

CGiS also supports intrinsic functions for matrix-matrix multiplication and matrix-vector-multiplication. These intrinsics, `matmatmul` and `matvecmul`, are specified in place of a `forall` loop. Program IV.8 shows a matrix-vector-multiplication. Note that these operations require the matrices and vectors to use a specific format for data packing. To this end, they are declared with the special packing modifiers `matrix` and `vector` in the `INTERFACE` section. This does not preclude their usage as streams in any other parts of CGiS: A program might fill a vector with normal streaming operations, let it then partake in a matrix multiplication, and perform a reduction on its components afterwards.

```
PROGRAM Matrix;

INTERFACE
extern in float A<Ax,Ay> : matrix;
extern in float Vin<Ax> : vector;
extern out float Vout<Ay> : vector;

CODE

CONTROL
matvecmul(A,Vin,Vout);
```

Program IV.8: Matrix algebra in CGiS

IV.3.d Semantics

Now it is time to formally specify the sequentialisation of data accesses and updates. To this end, we define the basic objects of our consideration as follows.

$\mathbb{V}$ is the space of variable names. $\mathbb{D}$ is the domain of primitive values. $\mathbb{D}$ is not specified further; it consists simply of all possible values of the primitive types such as `float4` or `uint` and of the user-defined structs. A *state* consists of an evaluation of the scalar variables, which is a function $\mathbb{V} \to \mathbb{D}$, and an evaluation of the indexed streams, which is a function $\mathbb{V} \to \mathbb{Z} \to \mathbb{D}$ (stream to element index to element value). The state space, then, is defined as

$$[\mathbb{V} \to \mathbb{D}] \times [\mathbb{V} \to \mathbb{Z} \to \mathbb{D}].$$

Note that we consider only one-dimensional streams for simplicity; this is no loss of generality, because a two-dimensional stream of size $X \times Y$ is isomorphic to a one-dimensional stream of size XY, with appropriate translation of indexes and index shifts; the extension of the semantics is straight-forward.

A `CONTROL` section consists of multiple `forall` loops or intrinsics. The effect of `show` statements and matrix intrinsics on the state is obvious, so we concentrate on `forall` loops. As mentioned before, a `forall` loop with multiple calls in its body is syntactic sugar on multiple forall loops. Therefore, without loss of generality, we consider only a single `forall` loop with a single call.

We regard a kernel as a function φ taking scalars and streams as an input and providing values of scalars and of streams at specific positions as an output:

$$\underbrace{\mathbb{D}\times\ldots\times\mathbb{D}}_{\text{scalar inputs}}\times\underbrace{[\mathbb{Z}\to\mathbb{D}]\times\ldots\times[\mathbb{Z}\to\mathbb{D}]}_{\text{stream inputs}}\ \to\ \underbrace{[\mathbb{V}\to\mathbb{D}]}_{\text{scalar outputs}}\ \times\underbrace{[\mathbb{V}\to(\mathbb{Z}\times\mathbb{D})]}_{\text{stream outputs}}.$$

The scalar inputs stand for all scalar `in` parameters, whether they come from iterators, global scalars, index operators or shifts. The stream inputs are all streams passed with `in` flow; a kernel operates on them with `lookup` operations. The output is the effect on iterators or reduction parameters and the effect on streams written to with a `writeback` operation.

Now let us consider the invocation of a streaming kernel `f` with its semantical function φ. The function call shall be

```
f(i1,...,in, c1,...,cp, index(i), U1,...,Uq, o1,...,om, V1,...,Vr),
```

where

- the `i` are iterators to streams $I_1,\ldots,I_n$ passed to `in` parameters;
- the `o` are iterators to streams $O_1,\ldots,O_m$ passed to `out` parameters;
- the `c` are scalars $c_1,\ldots,c_p$;
- the `U` are streams $U_1,\ldots,U_q$ passed to `in` parameters;
- the `V` are streams $V_1,\ldots,V_r$ passed to `out` parameters;
- `f` has shifted parameters, which are computed from the input iterators $i_{\sigma_1},\ldots,i_{\sigma_d}$ with offsets $\delta_1,\ldots,\delta_d$.

Note that because all streams are one-dimensional, we can, without loss of generality, assume that there is exactly one index operator, and that the shift displacement can be specified as a single scalar. Also note that we have split `inout` parameters in two parameters having solely `in` respectively `out` flow to ease the exposition.

The effect of a kernel on the streams at a particular point ι with current state (S_1,S_2) can be expressed by the function

$$\begin{aligned}\varphi' \quad = \lambda\iota.\varphi(&S_2(I_1)(\iota),\ldots,S_2(I_n)(\iota),\ S_1(c_1),\ldots,S_1(c_p),\ \iota,\\ &S_2(U_1),\ldots,S_2(U_q),\ S_2(I_{\sigma_1})(\iota+\delta_1),\ldots,S_2(I_{\sigma_d})(\iota+\delta_d)).\end{aligned}$$

Then, a call transforms the state space $(S_1,S_2)\mapsto(S_1,S_2')$, where S_2' is defined as follows:

$$S_2' \quad = \quad \lambda S.\begin{cases}\lambda n.\pi_1(\varphi'(n))(S) & S\in\{O_1,\ldots,O_m\}\\ \lambda n.\pi_2(\pi_2(\varphi'(n'))(S)) & S\in\{V_1,\ldots,V_r\}\\ \lambda n.S_2(S)(n) & \text{otherwise,}\end{cases}$$
$$\text{where } n' \text{ is the solution of } \pi_1(\pi_2(\varphi'(\cdot))(S))=n.$$

π_ρ is the projection on the ρth component. When n' is used (in presence of output streams) but not well-defined, then S_2' is not defined. The global scalars (S_1) are not affected.

The second method of calling kernels is per reduction. This is actually much easier if we have the function φ, as per our assumption. So, consider `f` being called with an iterator to stream I and scalar value s. Then, the call transforms $(S_1, S_2) \mapsto (S_1', S_2)$, where

$$S_1' = \lambda v. \begin{cases} \varphi(I) & v = s \\ S_1(v) & \text{otherwise.} \end{cases}$$

This section should have made precise that fluffy prose describing the data access semantics of CGiS: All inputs are evaluated, the kernel gets only scalar values and streams, and in the end the kernel outputs get fed one-by-one into the new streams. The implementation notes in Section V.1 describe how this is assured.

IV.4 Interfacing with the Outside

We have seen in the preceding sections how a single CGiS program works on streams. But CGiS programs do not stand in isolation: The whole reason for CGiS is to be used in conjunction with a larger application. So this is one kind of interfacing with which we shall deal here, in Section IV.4.a. Additionally, the principles of modular design apply to CGiS programs in the same way they do for any other program. Therefore, a way to structure larger CGiS program into smaller programs and libraries is needed. Section IV.4.b discusses CGiS' facilities for interaction of multiple CGiS programs.

IV.4.a Interfacing with the Application

Recall Program IV.2 and the subsequent discussion. Five tasks are necessary for interfacing with a generated CGiS program:

(1) Initialise the system.
   (a) Decide on a target.
   (b) Setup variant sizes.
   (c) Register all programs with the runtime.
   (d) Initialise the runtime.
   (e) Initialise the programs.

(2) Register the input data.
   (a) Setup pointers to the streams.
   (b) Setup data for the streams.

(3) Execute the program.

(4) Fetch the output data.

(5) Cleanup the system.

Not all these steps are needed for a single computation. Initialisation and cleanup need to be done only once, and there can be multiple executions per data transfer.

We cover these steps one by one. All procedures used are declared in a header file which has to be included in the application. The contents of the header for Program IV.1 are displayed in Program IV.9.

```
#ifndef HEADER_add
#define HEADER_add
#include "cgis_main.h" // A header defining CGiS_program.
typedef enum { CGiS_alldata_add,
  CGiS_add_in_stream1,CGiS_add_in_stream2,CGiS_add_out_stream
} CGiS_data_add;

bool register_program_add(CGiS_program* program);
bool setup_init_add();
typedef enum { } CGiS_size_add;
bool setup_size_add(CGiS_size_add type, size_t size);
bool set_data_add(CGiS_data_add tex, void* data);
bool get_data_add(CGiS_data_add kind);
bool setup_data_add();
bool execute_add();
bool run_add();
bool set_texture_add(CGiS_data_add tex, unsigned int texname);
bool get_texture_add(CGiS_data_add what, unsigned int* retval);
bool cleanup_add();
#endif
```

Program IV.9: The header generated for Program IV.1

(1) CGiS has the concept of *program objects* in the application which hold state pertaining to multiple CGiS programs. For example, the runtime has to allocate enough space for all CGiS programs taking part in an application to avoid costly reallocation or data transfer during the execution time. (This description is intentionally vague to spare the reader, for now, the details of the OpenGL implementation. The following discussion in this section also does not go too deeply into the details for the same reason. The exact purpose of the program object and the exact actions of the other procedures will become clear in Section V.1.) To this end, the programmer has to code the following steps.

   (a) Initialise a program object by calling the library function `get_CGiS_program` with a target identifier of `CGiS_SIMD` or `CGiS_GPU`. The SIMD identifier is for programs running on the SSE or AltiVec extensions of CPUs; because processors do not support both, a single identifier poses no disambiguation problems.[8]

   The return value of this function is a pointer to an interface object. This object has a few virtual functions which give information about the current target or can aid in debugging.

   (b) The sizes of the variant streams have to be specified. This is not necessary for the example program, because it used fixed size streams. In other cases, the function `setup_size_`*`NAME`* has to be called with an enumerant for the size to be specified and the actual size.

   (c) All CGiS programs have to register themselves with the program object. This is done by calling the function `register_program_`*`NAME`* with the program object.

   (d) The method `init_all` of the program object initialises the runtime. For example, the GPU version allocates space, and it opens a window for programs using

[8] The SIMD target is out of the scope of this work. It is described in detail, together with the associated work on the language and the compiler, in [F08].

the `show` statement. This function also checks that streams which have been inferred by the compiler to have the same size actually do get assigned the same size.

(e) The function `setup_init_`*`NAME`* then has to be called for all CGiS programs. One of the purposes of this function is to assemble the GPU programs and to transfer them to the GPU.

(2) Then, the data has to be specified. First, pointers to all external data have to be presented to the programs with calls to the function `set_data_`*`NAME`*. Second, a separate call to the function `setup_data_`*`NAME`* ends this process and declares the data as ready for transfer to the GPU. The other function `set_texture_`*`NAME`* will be discussed in Section IV.4.b, which covers the interaction of multiple CGiS programs

(3) The execution takes place with a simple call to the function `execute_`*`NAME`*. All changes of kernels, upload and download of data and setting of graphics state are performed in this generated function.

(4) Data is downloaded onto the CPU with a call to `get_data_`*`NAME`*. This works similar to data setup with `set_data_`*`NAME`*. The download function does not need a pointer, though, because it has been specified in the `set`-function. Likewise, the function `get_texture_`*`NAME`* is covered in Section IV.4.b.

(5) The internal data of the runtime system and the handles to the OpenGL context are freed by calling the procedure `cleanup_`*`NAME`*.

To further ease the interfacing, the programmer can download all output data by passing the constant `CGiS_alldata_`*`NAME`* to the function `get_data_`*`NAME`*, and anything from `setup_data_`*`NAME`* through to `get_data_`*`NAME`* is subsumed by `run_`*`NAME`*.

We observe a couple of points. First of all, the target is completely hidden. This is one of the main objectives of CGiS: To hide from the programmer where the data-parallel code is executed. Second, the generated code is easily usable. The programmer can interact with the generated code by a few, easily learnable functions. The initialisation steps might seem cumbersome at first, but the various calls are necessary to pass information to and fro among multiple CGiS programs. The main code, however, comprises only pointer store- and fetch-functions and the triggering of the execution. Interfacing with the GPU is as easy as interfacing with any other kind of library facility.

IV.4.b Interfacing with Other CGiS Programs

There are various reasons for a larger project to be split into a multitude of CGiS programs. For one, a natural division might present itself; one program might be responsible for computing a simulation, and another one for computing and displaying an appropriate visualisation. Also, a larger task might have to be split into multiple CGiS programs because CPU interaction is required at certain steps. For example, an iterative procedure has to be applied until some error condition falls under a threshold, and then the result is subject to further computations. This cannot be expressed in pure CGiS, for it entails looping on a yet larger scope: a sequential, data-dependent loop over `forall` loops. Thus, the task has to be split into one program for the iteration and one for the subsequent work. In the end, in both cases a way to transfer data from one CGiS program to another is needed.

Also, filtering out common code into libraries is a well-established principle of program design. An interface between CGiS programs provides library support on streams; CGiS also provides library support on scalars, that is, libraries of `CODE` functions.

These two issues require syntactic support in the CGIS language. Because syntactically kernel libraries need a strict subset of the features for cross-program data transfer, this shall be discussed first.

CGIS also provides a way to use the data computed in a CGIS program in other graphics applications. As it happens, this uses the same features as shared streams, and therefore it is explained last.

Textual Inclusion

With the directive `import "FILE";` the user specifies that a certain chunk of text has to be textually included at this position in the source file. That is, the contents of the specified file are parsed as though they were present at the position of the `import` statement. Thus, all names defined in the file can be used as though they were declared in the importing file.

The `import` statement can be placed in the `CODE` section (on the same level as procedure declarations) or in the `INTERFACE` section (on the same level as data declarations). In the first case, it provides a way to include library kernel procedures into a CGIS program. For example, a library of trigonometric functions might be specified in such a file and provide its functions to all files including it. In the second case, common streams can be declared in an imported file which then gets used by all CGIS programs working on these streams. Although it is not strictly *necessary* to use `import` in these cases, it makes life much simpler; see below.

Rationale: Using `#include` would have been more familiar to the C-programmer, but this preprocessor notation would stick out in CGIS. The `import` statement is more in line with the rest of the CGIS syntax. It is not unfamiliar to most programmers, because statements such as `import` or a similar `use` are well-established in other programming languages [CT98, T99, E04].

Templates

CGIS also features a template mechanism for generic procedures. The hitherto unspecified parts of the grammar are as follows.

$$
\begin{array}{rcl}
\mathit{TEMPLATE} & ::= & \texttt{template}\ \mathit{TYPETEMPS}\ \texttt{(}\ [\mathit{PARDECL}]^{+}_{,}\ \texttt{)}\ \texttt{\{}\ [\mathit{STMT}\ ]^{*}\ \texttt{\}} \\
\mathit{INSTANCE} & ::= & \texttt{instance}\ \mathit{ID}\ \mathit{TYPEINSTS}\ \mathit{ID}\ \texttt{;} \\
\mathit{TYPETEMPS} & ::= & \texttt{<}\ \mathit{ID}\ [\mathit{ID}\ ]^{+}_{,}\ \texttt{>} \\
\mathit{TYPEINSTST} & ::= & \texttt{<}\ \mathit{TYPE}\ [\mathit{TYPE}\ ]^{+}_{,}\ \texttt{>}
\end{array}
$$

The semantics of the templates and instantiations is pretty straightforward. A *template declaration* is exactly the same as a procedure declaration, only that some types are unspecified. These types are declared as *type variables* in the template declaration. A *template instantiation* is performed by declaring a procedure to be a particular instance of a template, assigning actual types to the formal types. The resulting procedure is then usable in the same way as when it would have been declared normally.

Program IV.10 shows an example for this. It starts with a template of a procedure with one type variable, `T`. The template is called `get_something`, and it is, in effect, a generic stream lookup procedure. The type variable `T` is used in the code in place of actual types. In the instantiations below the template, the type variable `T` gets assigned the type `float` for `get_float` and `int3` for `get_int3`. At this point, the template gets

```
template<T> get_something(in float coord, in T A<_>, out T o){
  lookup A: retval<coord>;
  o = retval;
}

instance get_something<float> get_float;
instance get_something<int3> get_int3;
```

Program IV.10: Function templates and instantiations in CGiS

```
procedure get_float(in float coord, in float A<_>, out float o){
  lookup A: retval<coord>;
  o = retval;
}

procedure get_int3(in float coord, in int3 A<_>, out int3 o){
  lookup A: retval<coord>;
  o = retval;
}
```

Program IV.11: Function declarations equivalent to Program IV.10

instantiated with the types and the procedure is checked with respect to the assigned types. The net result is the same as would result from Program IV.11.

Program IV.11 is more pure in the sense that all data accesses become explicit, whereas Program IV.10 alleviates the programmer from the burden to write essentially the same code over and over again.

Rationale:

- Genericity is in general a worthwhile feature. With procedure templates, the programmer can increase the reusability of a library, and in general factor duplicate code into common declarations. The only question is whether to use a type inference mechanism [T99] or explicit instantiation. Type inference would be very easy in CGiS, because all types are known statically and CGiS requires exact type matching in procedure calls. But in line with the rest of the language, CGiS requires explicit instantiation, to make the code more readable: Again, all information to understand the instance is in one place.
- The angular brackets for templates were modelled after C++ templates [S97b]. Confusion with the stream size use of angular bracket seems unlikely.

Shared Streams

Different programs sharing streams poses the problem of transferring information about data location and alignment between programs. For example, a raycaster implemented in CGiS consists of several programs which all work on a particular action to be performed on rays: One is responsible for tracing a ray through a scene, another one for computing intersections with objects, another one for computing the final colours, and so on; Section VI.2.g presents the raycaster in more detail. But all these programs need common streams. Data has to be passed to and fro between these programs. It would be prohibitively expensive to pass the actual data over the CPU, that is, to download the

data from one program and pass it over to the next program: Data transfer times would dwarf all possible performance gains. In terms of CPU programming languages, what one desires is to pass a *reference* or a *pointer* to the data.

For GPU programs, the streams are stored in textures. Thus, passing a pointer entails passing some identifier to the texture in which the stream happens to reside. And this is exactly what CGIS does: The function `get_texture_`*`NAME`* gets such an id from one program and the function `set_texture_`*`NAME`* passes this id to another program. However, the different programs also need to use the same texture packing. To ensure this, the position specifiers (Section IV.3.c) are employed again, but this time with an additional *packing id*.

```
extern inout float3 directions<PICX,PICY> : packing(1) RGB;
extern inout float states<PICX,PICY> : packing(1) A;
extern inout float current_tri<PICX,PICY> : packing(2) R;
extern inout float2 current_t<PICX,PICY> : packing(2) GB;
```

Program IV.12: Declaration of common streams in CGIS

For example, consider the fragment of common stream declaration for the raycaster in Program IV.12. It declares a variety of streams with an *extended packing specification*: In addition to the position specifiers such as `RGB` or `A` (Section IV.3.a), a packing id is used. This is a numerical id which simply identifies streams that shall be packed together in the same texture.[9] In this example, the streams `directions` and `states` are specified to reside in one texture, and streams `current_tri` and `current_t` are specified to reside together in the a different texture. All programs using these streams include them with the `import` directive: `import "streams.cgish";` in the `INTERFACE` section.

The contents of the streams are specified in the usual way by one program; for our example, let this be the program `init`. Then, the GPU pointer (the texture) is passed to other programs by the `[s|g]et_texture_`*`NAME`* functions:

```
unsigned int initdirs;
get_texture_init(CGiS_init_directions, &initdirs);
set_texture_raycaster(CGiS_raycaster_directions, initdirs);
set_texture_traverse(CGiS_traverse_directions, initdirs);
```

Now the texture is also known in the programs `raycaster` and `traverse`, and so these programs can exchange data through those streams without holding it in separate, local copies or using the CPU.

Rationale: It might concern the reader that suddenly a programmer seems to be forced to deal with *textures*, although CGIS should abstract from the hardware. But in reality, the user does nothing more than specifying a *data layout*; the `packing` specifier plays the role of padding or alignment directives in CPU programming languages. This, in turn, always must be specified in situations where a consumer of data might expect the data in another layout than the producer of data. The reason why this is the case in CGIS is an optimisation process explained in Section V.5.a which performs automatic data layout specific to a single CGIS program.

[9] This is *not* the texture id. The scope of the packing specifier is a CGIS file: For the same stream different packing specifiers can be used in different files without harm. For correct data exchange, the only thing that matters is that streams which have equal packing specifiers in one file also have equal packing specifiers in any other file. Obviously, to minimise confusion it is best to use the *same* specifier in all files; and this, in turn, is guaranteed by specifying the streams in a separate file and including this file with the `import` statement.

External Visualisation

The identifier one gets when performing the `get_texture_`*`NAME`* function is actually a standard OPENGL texture id. The CGIS implementation makes sure that, when it is requested by the application, it holds the desired data and is free to be bound as an OPENGL texture in arbitrary applications. Thus, the programmer can use the textures in more advanced visualisations, for example, to use the image in a larger scene or to distort it and map it on a surface. Obviously, it is forbidden to *modify* the contents of a texture.

This feature enables CGIS programs to be used as *procedural textures*. A procedural texture (a texture which is not prespecified, but generated at run time) is a common use of fragment programs in graphics applications. CGIS programs can be used for this, but also as a plug-in in scientific visualisations, where the interfacing requires more than just a plain 2d-visualisation as is possible with the `show` statement.

IV.5 Example Program

As an example of CGIS code, we finally look at a larger program which uses most of CGIS' features. This way, we can see CGIS in the context of a larger example, which will certainly make its constructs clearer. Also, it serves as a general example of how CGIS programs look like. For a more detailed explanation, consult [G05].

The task to implement is the following: Let there be a container filled with a liquid and with an image on the bottom. The liquid gets distorted and waves are engendered. These waves propagate over the surface, interacting which each other, getting reflected by the container's brink and withering with time. Compute those propagation, the refraction of the image and display it. For example, Figure IV.2 shows an image with distortion (the caustics are part of the image).

Program IV.13 shows the data used in this program and the general sequence of computation for one time step of the simulation. The kernels are listed in Program IV.14. We observe the following sequence of computations taking place:

(1) First, the program computes the propagation of waves. To this end, it takes the last state and computes from them the general propagation of the waves. For this, it takes into account a damping factor which specifies how fast the waves should whither. For computing the reflection at the borders, the kernel gets as inputs also the position of its element (the index operators) and the dimension of the container (`WIDTH`, `HEIGHT`).

(2) Then, given the then current state of simulation, the refraction of light is computed, by two separate kernels for each dimension.[10] As input, they get the refractive index of the liquid.

(3) Then, the final image is computed from the distortion and the original image supposed to lie on the bottom of the container. A `show` statement visualises this on the screen.

[10] In reality, it is one parameterisable kernel, see Program IV.14.

Figure IV.2 Refraction

In this sequence, we see several features of CGiS: Sequences of parallel streaming computations; passing of stream iterators, global scalars and index operators; passing of streams for lookup purposes; visualisation. We also see in the `INTERFACE` section the declaration of all global data. The streams have not a specified size: The application sets `SIZEX` and `SIZEY` at runtime. The streams `LAST` and `CURRENT` have a packing specifier, because the application needs to switch those two streams after each time step.

Now we briefly consider the kernels, listed in Program IV.14. The file `util.cgish`, which is imported by Program IV.14, contains the procedure `clamp(inout float3 x)`, clamping the input to the range $[0,1]$ (`0 max (1 min x)`), that is, computing

$$\lambda x. \begin{cases} 1 & x > 1 \\ 0 & x < 0 \\ x & \text{otherwise} \end{cases}$$

For a detailed explanation of the `CODE` section, the reader should consult [G05]. Here, upon inspection we can see the following features in action: Multiple streaming kernels called from different contexts; lookups with relative addresses via `gather` on iterators, to get access to the height field around a specific point; lookups with absolute addresses via `lookup` on parameter streams, to compute the distorted image from the original image; passing of element indexes; control flow; procedure calls inside of `CODE` (`refractionX` and `refractionX` to `refractionXY`); library support via `import`.

The reader can observe that the subtasks of the simulation naturally map to CGiS constructs. The abstraction of CGiS allows to express the algorithm in a concise, yet readable way. As we shall see in Section VI.2.d, the generated code is also fast. Thus, this wave propagation example can serve as a convenient example to illustrate CGiS' strengths.

```
PROGRAM viswave;

INTERFACE
extern inout float LAST<SIZEX,SIZEY> : packing (1) A;
extern in float CURRENT<SIZEX,SIZEY> : packing (2) A;
extern in float RINDEX, DAMPING, WIDTH, HEIGHT;
intern float X<SIZEX,SIZEY> : packing(4) R;
intern float Y<SIZEX,SIZEY> : packing(4) G;
extern in float3 TEXTURE<SIZEX,SIZEY>;
extern out float3 IMAGE<SIZEX,SIZEY> : RGB;

CODE
// See Program IV.14.

CONTROL
forall(last in LAST, current in CURRENT)
  propagate(last,current, indexX(last),indexY(last), DAMPING,WIDTH,HEIGHT);

forall(x in X, y in Y, height in CURRENT){
  refractionX(RINDEX, x, height, indexX(height), WIDTH);
  refractionY(RINDEX, y, height, indexY(height), HEIGHT);
}

forall(pixel in IMAGE, height in CURRENT, x in X, y in Y)
  render(TEXTURE, pixel, height, x, y);

show(IMAGE);
```

Program IV.13: Wave propagation in CGiS: `INTERFACE` and `CONTROL`

IV.6 Fitness for the Purpose

During the course of this chapter, rationales alluded to various causes for the choice of language features. After having seen these specifics, let us now review the objectives stated in Section IV.1 of CGiS as a whole. A couple of these objectives can be evaluated from the description of the language alone, but for some the evaluation has to be deferred to future chapters. *Adaptability* and *compatibility* are features of the inner workings of the compiler, and thus they are the topic of Chapter V. *Controllability* also happens to be more at home in that chapter, for it is there that the additional possibilities for giving information to the compiler (*hints*, in CGiS' terminology) will be covered. *Efficiency* is the topic of Chapter VI, where several applications will be presented.

The objective of *abstraction* surely is achieved. The target of the execution of a CGiS program is completely hidden, both when writing the program and when executing it. The user has at his disposal a truly *general purpose, data-parallel language*, which can be used to implement *general purpose, data-parallel algorithms*. However, to achieve the objective of *visualisation*, CGiS offers the `show` statement and the opportunity to read stream contents as standard OpenGL textures. Thus, both those programmers who want just a general data-parallel language and those who regard CGiS as a language aiding in a greater visualisation can supply their wants in CGiS.

Familiarity is achieved by basing CGiS on standard imperative languages, borrowing features from C and PASCAL. Familiarity has been sacrificed in a few cases because of hardware limitations. Recursion or dynamic memory allocation would not be possible to implement on the hardware. Likewise, a very precise statement regarding data type

```
procedure propagate(inout float last, in float cur, in float x, in float y,
                    in float damping, in float width, in float height){
  gather cur: t<0,-1>, l<-1,0>, r<1,0>, b<0,1>,  tl<-1,-1>, tr<1,-1>,
              bl<-1,1>, br<1,1>, tt<0,-2>, ll<-2,0>, rr<2,0>, bb<0,2>;
  if((x<2.0) or (y<2.0) or (x>=width-2.0) or (y>=height-2.0)) last = 0;
  else last = damping*(((t+l+b+r+tl+tr+bl+br+tt+ll+rr+bb)*0.1667)-last);
}

procedure refractionXY(inout float wpos,in float w,in float wh,in float th,
                       in float rindex, in float current){
  ... // Compute refraction at specified point.
}

procedure refractionX(in float rindex, inout float xpos, in float current,
                      in float x, in float width){
  gather current: left<-1,0>;
  refractionXY(xpos,x,width,left,rindex,current);
}

procedure refractionY(in float rindex, inout float ypos, in float current,
                      in float y, in float height){
  gather current: top<0,-1>;
  refractionXY(ypos,y,height,top,rindex,current);
}

import "util.cgish";

procedure render(in float3 image<_,_>, out float3 pixel, in float current,
                 in float x, in float y){
  lookup image: colour<x, y>;
  float3 unclamped = colour+0.25*current;
  pixel = clamp(unclamped);
}
```

Program IV.14: Wave propagation in CGiS: CODE

representation is not possible. CGiS here appeals to the programmer's understanding that just like the GPU's peculiarities result in some advantages, they do not come for free.

Some other constraints on familiarity have been made to increase *readability*. For example, the programmer may not pass a variable multiple times to output variables of a procedure, effectively prohibiting aliasing; implicit type conversions are present, but only in specific cases; interactions between multiple instantiations of a single kernel are confined to specific parts of a source file. As Bjarne Stroustrup said: *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."* [S07a]; CGiS tries to hide the foot and makes it impossible to shoot it, while retaining its use as a means of transportation in the restricted setting of a GPU stream programming language. It strives to have a clean design, where as well as possible the meaning of arbitrary chunks of code can be understood without referring to too much context.

So, from these points of view, CGiS can be said to have met its goals with respect to the user-oriented features. It is instructive to compare CGiS now with the languages in Section III.3. Obviously, CGiS bears the most similarity to BROOK; RAPIDMIND and

ACCELERATOR are too different in their programming models for a meaningful comparison. The model of BROOK is very similar to that of CGIS,[11] and therefore a comparison can lay out some differences on the outside.

For example, CGIS can switch seamlessly between streams-of-structs and structs-of-streams. CGIS also supports automatic data layout: Whereas BROOK works around output restrictions with multipass rendering, CGIS uses adaptive data layout and data reordering operations so as not to redo executions. Whereas CGIS allows loop splitting and special operator symbols, BROOK uses what CG has to offer. Integer streams did not work in the latest tests, whereas CGIS with is dependence just on generating code for assembly language itself could implement the integral types and operators by going down to assembly level; and BROOK does not automatically work around the hard loop iteration limits by nesting loops as `cgisc` does. Another distinction between BROOK and CGIS is the higher level control flow. Whereas in CGIS the whole of the algorithm to be implemented on the co-processor is neatly implemented in a single language in a separate source file, the programmer of a BROOK version has to write some the parallel parts of the algorithm (corresponding to `CONTROL` in CGIS) in C-code. Thus, CGIS offers a cleaner differentiation between the various targets.

On other levels, BROOK offers a higher abstraction and more possibilities than CGIS. This concerns, for example, mechanisms to realign streams of different sizes or to reduce a stream into a smaller stream instead of a singular value. The dependence on CG can also be seen as a positive feature, because the CG compiler implements an important part in the generation of the eventual code, namely the generation of the low-level code. Therefore, a large part of the transformations and optimisations as presented in Chapter V are not needed in BROOK; from the standpoint of the compiler implementor, that is an advantage.

All in all, CGIS and BROOK offer largely similar solutions for a common problem, with both sides being offering higher comfortability in one or the other area. But apart from that, the two systems are very different internally, but quite similar to the programmer. The differences are mostly concerned with the implementation, with the different levels on which the compilers work and the affected algorithms.

IV.7 Summary and Outlook

We have seen in this chapter the explanation of CGIS, the language. In a sequence of exhibitions with increasingly larger scope, all features of the CGIS language were mentioned. By way of an example, most features have been shown in context. We have seen the goals CGIS strives to fulfil and argued for the achievement of some of them: those which were concerned with the language itself, in particular familiarity, readability, visualisability and abstraction. The rest of the CGIS system has to be presented in the next chapters, and it has to be evaluated with respect to the remaining goals. This exposition starts in Chapter V with the compiler itself and the runtime system of CGIS.

[11] That two of four independently developed models happen to be very similar can be seen as an argument for the strength of said model.

V

The CGiS Compiler

optimism: n.
What a programmer is full of after fixing the last bug
and before discovering the *next* last bug.
E. S. RAYMOND, *The Jargon File*, 2003

This chapter describes the CGiS system. It can be divided coarsely into three parts:

- The actual compiler, `cgisc`, with all auxiliary libraries and tools;
- the runtime system;
- the infrastructure, such as the build system, tests and documentation.

All those parts are covered in this chapter.

The CGiS compiler has three different kinds of backends, resulting in three different runtimes and different sets of analyses, transformations and generators:

(1) The GPU backend, targeting NV30, NV40 and G80 GPUs;

(2) The CUDA backend, targeting CUDA;

(3) The SIMD backend, targeting AltiVec and SSE CPUs.

For the most part of this chapter, only (1) is of concern, as it is the central part of my work; (2), although interesting in its own right, is implemented merely as an auxiliary feature for the evaluation of (1), and thus described as such. (3) is the work of Nicolas Fritz [F08, FLW07]. (2) and (3) are shortly presented in Section V.6.a.

The main component of the system, and thus the main topic of this chapter, is `cgisc`. However, it is instructive to start the discussion at the end of the compilation process, namely at the runtime system and the generated code. This is the topic of Section V.1. When this is covered, the choices for the compiler's internals become more clear, and the objectives of the transformation phases are more easily motivated. Section V.2 then covers the internal representation of the compiler. This entails the intermediate language

and other internal objects, and how this is transformed into the textual representation of C++ and GPU code. The proper compilation is then the topic of the next three sections. Section V.3 covers everything related solely to CGiS code, from parsing it into the intermediate representation onto analyses and transformations on the code; Section V.4 covers the generation of intermediate GPU code from the CGiS code; and Section V.5 covers analyses and transformations on the GPU code. Section V.6 describes everything else relevant to the compiler system; the CUDA and the SIMD backend, auxiliary components and a list of contributors to the software. Section V.7 sums up and concludes this chapter.

Figure V.1 Coverage of the CGiS system in Chapter V

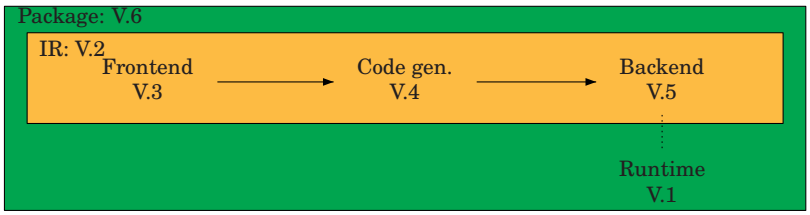


Figure V.1 provides an overview of the structure of the system and its coverage in this chapter. Arrows denote information flow between the compiler's phases, the dotted line is a dependency of separate components. More detailed phase diagrams will be presented in Sections V.3–V.5.

Hints. Throughout this chapter, a concept called *hints* will be discussed. A hint is a directive to the compiler annotating the source code with additional information, such as upper iteration bounds on loops or information regarding the use of external streams. These hints are part of the CGiS grammar, but have not been introduced in Chapter IV, because they cannot be appreciated without knowing the reasons for their very existence. Hints are explained in more detail at their first occurrence, in Section V.1.c, and used freely afterwards.

V.1 Runtime System

This section describes the runtime system of CGiS for GPUs. This is defined as everything happening behind the scenes when the programmer uses one of the abstract access functions. These functions were first mentioned in Section IV.1.b and explained more thoroughly in Section IV.4.a. The discussion entails both the CGiS library with which the programmer has to link and the generated code itself using this library.

Instead of proceeding function by function, the description is divided into topical units. Section V.1.a describes the facilities to create the OpenGL context and allocate memory. Section V.1.b considers the handling of stream data, and Section V.1.c considers the handling of GPU programs. Final remarks are presented in Section V.1.d.

V.1.a Context

The generated code can be regarded as a translator from certain high-level operations to the low-level OpenGL operations. To ensure correct operation of this abstraction layer, the code needs to store information both about the application and about the OpenGL

state. This is called the *context* of the generated program. This information is divided into two parts:

- common information needed by all CGIS programs
- specific information needed by a single CGIS program

For example, when a multitude of CGIS programs work together, they have to share a single handle to the OPENGL driver, but each program needs to access only the GPU kernels generated for its specific CGIS program.

Among the specific data, the program uses the following types of objects:

- a pointer to the current context, encapsulating the common data
- texture objects, encapsulating OPENGL textures, and copies of scalar data
- kernel objects, encapsulating GPU programs
- sizes of streams and their positions in textures
- pointers into the application's memory

Most specific data are covered in the subsequent sections.

In CGIS terminology, the common information is called *context*. It is stored in an object of type `CGiS_OpenGL_program`. As the name suggests, there are other such objects available, namely those of type `CGiS_SIMD_program` and `CGiS_CUDA_program`. Both are derived from the superclass `CGiS_program`, and that is all the user needs to know. The function `get_CGiS_program` returns a pointer to an object of type `CGiS_program`, which happens to belong to one of the subclasses, as indicated by a function argument.

To set up the system and start interfacing with the OPENGL driver, the context opens a window with function calls to the windowing system's primitive interface functions. This window is, in general, invisible, because all rendering is done into off-screen memory buffers. In case of programs using the `show` statement, however, it opens a visible window. The processing of setting up the window and creating the offscreen buffers is highly dependent on the windowing system. The codes for Windows and X are completely different. This is the only such place, however; as soon as the contexts are created, any other operation uses standard OPENGL functions, either core functions or extension functionalities [K07]. These are independent of the windowing system or operating system. As such, the runtime can work on any system which features a Windows or X windowing interface; that is, on any current system. The X-part of the system has been tested only on Linux; it is very unlikely that it should not run on other X-based systems, provided the hardware is supported.[1] Note that the *generated code* is still independent of the system; the user need not compile a CGIS program several times for various systems, he only has to compile the generated C++ program for each system (as he must do anyway).

The cleanup function frees the storage allocated in the host memory for texture rearrangement. It also deallocates some OPENGL resources, reclaiming memory used by the driver. The complete OPENGL context is cleared by deleting the object of class `CGiS_OpenGL_program`.

[1] Currently, NVIDIA supports also Solaris, FreeBSD and MacOS X.

V.1.b Data Storage

CGiS streams are held in OpenGL *textures*. Considering external streams, all of the following information needs to be present.

- sizes of the streams
- space to hold the streams for reordering
- information about which stream sizes should agree
- pointer to the streams' positions in the application's memory
- information about the type of the data

As an example, let us consider the tasks to be performed for a stream declared as `extern inout float2 Stream<SIZE>;`.

The size, as it is unspecified, has to be set by the application; it is stored in an internal variable `IStream_size0`. Because the stream is one-dimensional, the variable `IStream_size1` is fixed at `1`. When the pointer to the input data is provided, the code as displayed in Program V.1 is executed. The pointer is stored, for it has to be used again to transfer the data back into the application's memory. The memory to hold the texture data is allocated, if necessary, and the input data are then distributed into this memory. Where the data is going to reside is specified by a location specification; in this case, it is to reside in the `yz`-components of texture 2.

```
static const location_t locs_Stream[2] = { {2,1},{2,2} };

bool set_data_NAME(const CGiS_data_NAME tex, void* const data){
  switch(tex){
    case CGiS_NAME_Stream:
      textures[2].maybe_alloc();
      textures[2].number_null();
      Texture::distribute_float2_elements(static_cast<const float2*>(data),
                        locs_Stream,textures,IStream_size0,IStream_size1);
      outpointer_Stream = static_cast<float2*>(data);
      break;
    ...
  }
}
```

Program V.1: Distributing data into textures

The function `setup_data_`*`NAME`* calls library functions to create the necessary memory on the GPU and to upload the textures thereto. The function `get_data_`*`NAME`* works very similar to `set_data_`*`NAME`*, only in the other direction. Instead of possibly allocating memory on the GPU, it first possibly has to download the texture into CPU memory, and then it constitutes the application's stream from the textures.

The reordering presented here is not necessary in all cases. When the data layout as expected by the kernels is the same as the natural array layout of the application (Section V.5.a), the distribution and constitution phases are omitted.

V.1.c Directing the GPU

The function `execute_NAME` is the translation of the the `CONTROL` section of a CGiS file. Just as the `CONTROL` section can be broken down into independent sequences of `forall` loops with a single call each (Section IV.3.d), the same holds for the actual execution. Thus, we consider here only one such call.

```
// forall loop 2 (id=1):
// Call to kernel test_cross.
program->attach_FBO();   // (1)
program->texture_to_renderbuffer(0,&(textures[5]));   // (1)
program->draw_to_buffers(1,true);   // (2)
textures[2].hook_texture(GL_TEXTURE_RECTANGLE_NV,0);   // (3)
textures[3].hook_texture(GL_TEXTURE_RECTANGLE_NV,1);   // (3)
program->run_program(TC_RECT,test_cross_PSkernel,"test_cross_PSkernel",
                     NULL,buffer_size_x,buffer_size_y, 0,0,
                     buffer_size_x,buffer_size_y);   // (4)
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,GL_COLOR_ATTACHMENT0_EXT+0,
                          GL_TEXTURE_RECTANGLE_NV,0,0);   // (6)
program->detach_FBO();   // (6)
```

Program V.2: A fragment of an execution function

Program V.2 presents a fragment of code corresponding to one such call. The general sequence of events is as follows:

(1) If a program can render its output in a texture, attach the texture as an output buffer.

(2) Notify the OpenGL runtime of the number of output buffers desired by the program to execute.

(3) Hook the textures holding the input data to texture stages.

(4) Run the program on specific ranges of the input and output buffers and with specific scalar inputs.

(5) Copy data from output buffers into textures.

(6) If a texture had been attached as an output buffer, remove the attachment.

All calls other than `glFramebufferTexture2DEXT` are functions of the CGiS runtime.

Inputs and Outputs

Considering first the attaching of input and output data, we have to keep in mind that GPUs do not support read-write memory (Section II.3.b). Thus, if a texture is needed as an input buffer (3), it cannot be contemporaneously bound as an output buffer (see Section V.5.a for binding considerations). In these cases, the program writes into a temporary output buffer (2) and copies the data from this buffer into the texture (5).[2] In the example of Program V.2, it so happens that the output textures and the input textures are disjoint. Thus, the output texture is attached as an output buffer (1, 2) and later detached (6).

In all aspects of memory management, texture packing has to be considered, in particular its relations with textures bound as framebuffers. Suppose a program writes into

[2]This copy operation takes place solely in the GPU memory, without using the host memory.

a stream *A* which happens to reside in the same texture as a stream *B*. Unfortunately, when binding a texture as an output buffer, its contents become undefined. Therefore, when writing only into the components for stream *A*, there is no guarantee that the components for stream *B* are unaffected. Thus, `cgisc` introduces *compensation copies*, that is, instructions which copy the data for stream *B* into the correct places in the output buffer. Hence, the texture has to be bound as an input texture, even when the CGiS kernel itself does not need data from that texture.

This problem is aggravated in the presence of external streams (Section IV.4.b), where the compiler cannot ensure that a texture is *not* used by multiple streams. To this end, CGiS uses a *hint*. A hint is an annotation by the programmer serving at least one of the following purposes:

(a) to direct a compiler optimisation into a particular direction, when the programmer believes a heuristic to need aid;

(b) to assert a certain condition on input data.

A hint always has the following form:

$$\begin{array}{rcl} \mathit{HINT} & ::= & \texttt{\#HINT (}\ [\mathit{HINTEL}]^{+}_{,}\ \texttt{)} \\ \mathit{HINTEL} & ::= & [\mathit{ID}\ \texttt{:}]^{?}\ \mathit{ID}\ [\texttt{=}\ \mathit{INTCONST}]^{?} \end{array}$$

A hint thus contains only a simple keyword, or a keyword together with a number, optionally prefixed with a profile identifier. The profile identifier specifies that a particular hint should be in effect only for a specific profile.

In the case at hand, the hint `no_texture_reuse` ensures that no texture is used for other streams in external programs.

Scalar inputs are sent to the program via a CPU pointer upon invocation of the program (`NULL` in (4) in Program V.2). The library function `run_program`, which is responsible for the execution of the program, supplies it with these data via GPU program parameters (see the following section).

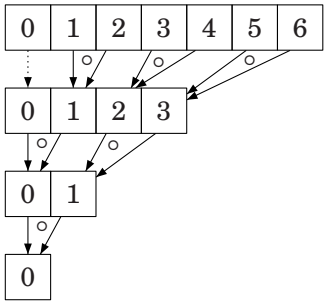
The functions `get_texture_`*NAME* and `set_texture_`*NAME* simply return respectively set the OpenGL id of the texture associated with the referred stream. As specified in the semantics (Section IV.4.b), the user is responsible for not changing the contents or any attributes of the texture.

Programs

The actual task of `run_program` is to upload the kernel to the GPU, feed it with the coordinates necessary to fetch its input data and write its output data, and start it. Also, it passes the raw index information through to the kernels, so that they can implement the index operators and shift operations. These tasks are executing on the upper part of the graphics pipeline (Section II.1.a): A vertex program transforms the coordinates into a format suitable for the GPU to compute the output positions, and the rasterisation creates the fragments with data interpolated from the vertex program's outputs.

The coordinates are quite straight-forward in a classical streaming computation, where the streams are aligned and the coordinates span the entire data range. For reduction computations, the coordinates have to be computed differently. To see why, recall that a reduction is performed recursively by catenating any two adjacent pixels. Figure V.2 shows schematically how (intermediate) output values stem from the catenation of two

Figure V.2 Reduction schema



(intermediate) input values, or from an unmodified value, in case of an odd number of elements (see the first element in the top row).

The function `run_program` also provides debugging facilities, such as outputting the computed coordinates, the number of written pixels (with a GPU feature called *occlusion query*), or the actual output of the computation. Its use for uniform scalar parameters has already been mentioned.

The kernels for intrinsics (Section IV.3.c) are defined in the runtime library. Apart from that, the implementation of intrinsics uses the same features and infrastructure as normal kernels.

Visualisation

The visualisation works much in the same way as the execution of a normal streaming. In fact, it *is* a standard program execution, but in another context. To this end, the program has to temporarily switch to the visible window, adjust the state accordingly and then switch back afterwards. Program V.3 gives an example for this.

```
// show 1 (id=1):
// Switch to on-screen window:
common_VS->dehook();
twindow->switch_to_window();
twindow->setup2D();
common_VS->hook();
// Render IMAGE:
textures[4].hook_texture(GL_TEXTURE_RECTANGLE_NV,0);
program->run_program(TC_RECT,Visualise_IMAGE_PSkernel,"vis_shader",
                     NULL,textures[4].size_x(),textures[4].size_y(), 0,0,
                     textures[4].size_x(),textures[4].size_y(), true);
// Switch to off-screen buffer:
common_VS->dehook();
twindow->switch_to_pbuffer(tbuffer);
twindow->setup2D();
common_VS->hook();
```

Program V.3: A show operation

V.1.d General Remarks

As has been shown in the preceding sections, quite a lot is going on behind the scenes. The code examples show only the generated code, to show the reader on a relatively high level what is going on; most of the real work is done in the library functions using a variety of OPENGL operations. Indeed, the problem in implementing a GPGPU program by hand lies not so much in the principle sequence of operations, but more in the *quantity* of *different* OPENGL operations which have to be used to implement even a simple example.

Portions of the pipeline touched by GPGPU include:

- context creation and management, for different windowing systems
- program composition
- shader creation and management
- texture management, including data upload and download
- framebuffer management, including renderbuffers and read-backs
- execution of programs, including managing the data buffers and uniform data

This alone suffices to demonstrate the value of GPGPU languages vs. shading languages, and as such strengthens the claims of Section III.1.

V.2 Internal Representation

This section deals with the internal representation of the code. Here, we shall see how the instructions and their operands are represented, and how they form larger control flow representations, finally making up whole programs. Of course, it is not the intent to present the complete data structures. Instead, the representation is exhibited only as far as necessary to get an overview of the whole and to understand the remaining sections.

The internal representation is designed as an object-oriented class hierarchy. Portions of the hierarchy are used in the discussion to show the interrelation between the various components. For these diagrams, the same holds as for the grammar fragments in Chapter IV: They are intended for exhibitory purposes and might leave out details which are irrelevant to the discussion at hand.[3] The intent of the diagrams is to make *clear* the internal workings, after all, and not to drown the reader in abundance of minuteness.

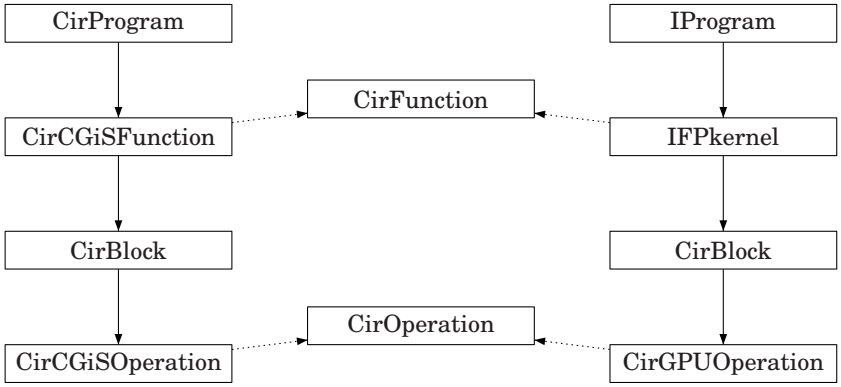
In these diagrams, dotted arrows denote an "is-a" relation; for example, in Figure V.3, a `CirGPUOperation` is a a `CirOperation`, just like a `CirCGiSOperation` is. Solid arrows denote an "association" relation, to be made more precise in the respective figure's explanation.[4]

Figure V.3 graphically represents the main classes used to store the code internally. The associations are of different cardinality and organisation:

[3] As a non-trivial example, parts of a subhierarchy may be left out for good or expanded later, and in general everything which is used solely in the CUDA-part or the SIMD-part of the compiler is deferred.

[4] The prefix `Cir` stands for CGiS *intermediate representation*. The prefix `I` stands for *internal* and denotes objects which are going to get an own representation in the output program.

Figure V.3 Representing code in `cgisc`

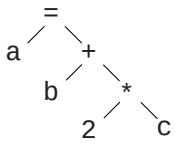


- A program has a set of functions.
- A function has a directed graph of blocks.
- A block has a sequential list of instructions.

V.2.a Operations

It is often customary to represent expressions as a tree or a DAG of operations in a stage of the compiler [M97]. For example, Figure V.4 shows two ways to represent the assignment statement `a = b+2*c;`: (a) shows a tree representation, (b) shows the representation of `cgisc`. The representation as a sequence of instructions results from the need of the program analysis tool PAG (Section V.6.a).

Figure V.4 Two representations of the statement `a = b+2*c;`



(a)

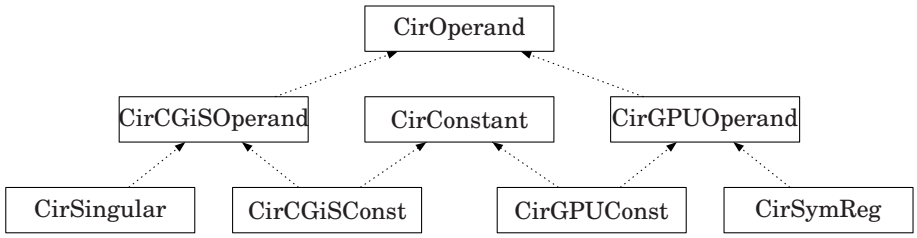
```
MUL t1, 2,c;
ADD t2, b,t1;
ASSIGN a, t2;
```

(b)

Figure V.5 shows an extract of the class hierarchy of the operands used by the internal instructions. A few points warrant mentioning.

- A `CirSingular` is a CGiS variable which is not an iterator or a stream.
- Something inheriting from `CirConstant` is not a variable with a constant value, but a proper constant. That is, if constant propagation (Section V.3.c) deems a variable to be constant, it is not flagged as such in the operation but replaced by a object of type `CirConstant`. In general, constants can take the place of any operand in any operation.

Figure V.5 Operands of internal operations



- GPU operations work on registers. Thus, a variable on the GPU side is called a symbolic register, or *symreg* for short. An actual register, or *actreg* for short, is then a proper hardware register.

GPU operations can use a variety of modifiers to change their workings. Among them, swizzling and masking are of particular importance. Thus, a `CirGPUOperation` has some properties specifying the swizzles for its operands and the mask for its target, if applicable.

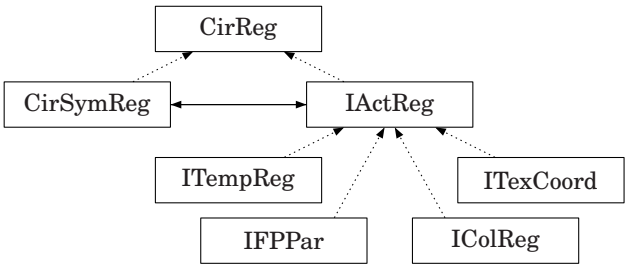
The hierarchy of operations is split quite broadly; in addition to the standard arithmetical operations, there are a number of specialised operations. For example, an operation of type `CirCGiSDynMask` is solely responsible for the dynamic mask operations, and a `CirCGiSIndex` fetches the index of elements. These operations are direct consequences of particular CGiS features. Other specialised operations stem from particular transformations of the internal representations. For example, instructions of kind `CirCGiSGuardedAss` and `CirCGiSSetGuard` get generated by if-shadowing (Section V.3.b), whereas a `CirCGiSDataComp` is a compensation copy (Sections V.1.c and V.5.a). Thus, the operations are on a variety of levels. The procedure of mapping them to the primitive hardware operations is explained in Section V.4.

V.2.b Registers

Regarding the GPU registers, Figure V.5 presents one of the white lies of these diagrams; Figure V.6 shows the actual hierarchy of registers. The actregs correspond to the four parts of registers which may occur as operands in operations (see also Section V.1):

- `ITempReg` are typed, general purpose, read/write registers. These are the main work registers of the kernels.
- `ITexCoord` are the registers in which the fragment program receives input from the vertex program. Through these registers, the index information provided in the program call is received. They are also used in implementing the base offset for `gather` operations.
- `IFPPar` are the registers which receive the uniform parameters, which are set before the execution of a program.
- `IColReg` are the registers holding the outputs of the program. These are the values which are written at the positions specified by vertex program, which gets, in turn, its inputs from the position values at the invocation of the program.

Figure V.6 Kinds of registers



It is important to note that the actual registers have no knowledge about the outside of a kernel. To a fragment program, they are simply some registers without any special properties except some restrictions on their flow (for example, the input register cannot be written to).[5] It is the runtime which has to ensure that these registers are linked to the correct hooks in the rest of the system. For example, a fragment program writes only to a colour register bound to a particular stage, and it is oblivious to which texture is currently bound to that stage or at which position it is going to write. That a write to `result.color[1].yz` arrives where it should, is the responsibility of the runtime system, that is, the binding is performed at execution time. The runtime system performs the necessary operations based on the location variables of the streams (Section V.1.b) and the binding specifications of the kernels (Section V.1.c).

The operations in `cgisc` work solely on symregs, which in turn have a position somewhere in an actreg. Section V.5.c shows how this is handled in the register allocation, particularly in the presence of vectorisation. In general, this builds on the fact that the fundamental unit of values in CGIS is not the variable or register, but the *component* of a variable or register. A symreg or actreg is merely a composition of several such components as specified by a CGIS source operation or needed by a hardware operation.

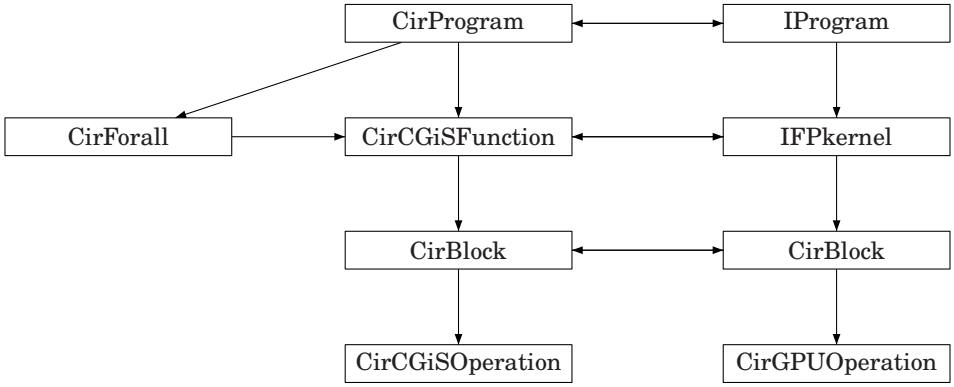
V.2.c Functions

The operations and operands are sequentially stored in blocks, and the blocks together form a control flow graph in a classical way. This graph, in turn, is the essence of a `CirFunction`, and thereby of a `CirCGiSFunction` or an `IFPkernel`. They are in turn assembled into a program according to the control flow structure of the `forall` in the `CONTROL` section.

Figure V.7 shows the interrelationships between the various internal structures (compare with Figure V.3). Functions have a one-to-one correspondence with kernels, and in general the same control flow (block) structure. The high-level control flow structure is formed by the sequence of `forall` loops, represented as the `CirForall` objects. They all call a function and note the necessary mappings between actual parameters of the call and formal parameters of the function. There is no one-to-one correspondency between operations, of course; a `CirCGiSOperation` can be implemented by more than one `CirGPUOperation`, or several can be implemented by one (see Section V.4.b for the generation and Section V.5 for further transformations).

[5] Therefore each is an `IActReg` to the program and the instructions.

Figure V.7 Functions in `cgisc`



V.2.d Output

When all code has been generated and the internal representation of the final code has been created, what remains is to create the textual representation of the code. That is, a mapping from the data as represented in this section into the textual C++ and GPU code in Section V.1 has to be created. Fortunately, this is quite straight-forward, because this activity essentially is compositional.

For example, a `CirShowForall` has an operation to write the necessary code for its execution into the `execute_`*NAME* function. This in turn is achieved by calling appropriate methods of the function calls and textures to output their representations; but the code for one `forall` loop is independent of the code for other `forall` loops.[6] A similar argument can be made for all other internal structures resulting in an external representation, from the textures to the kernels, which serialise their control flow graph into the sequential instruction stream in GPU assembly language.

To facilitate debugging and understanding the compiler's output, automatically generated comments are sprinkled throughout the generated program, but most importantly in the GPU code. As examining assembler output can be a tiresome task, comments which mention what the compiler desired a particular instruction sequence to do can tremendously aid in determining why that sequence does not what the programmer desires to be done.

V.2.e Profiles

Capabilities of various targets are separated through *profiles*. A profile is one of three things:

(1) a target specified on the command line of the compiler, instructing `cgisc` for which architecture it should compile a CGiS program;

(2) a variable defining what code paths the compiler should take during code generation;

(3) an object carrying certain numeric attributes of a target.

[6] The *output* codes for the representations of loops are independent. Of course, to actually arrive at the internal representations involves arguing about larger part of the program: This internal step of code generation is not compositional.

(1) determines (2). Profiles as internal objects in the sense of (3) are relevant only for the various GPU architectures, where they hold information such as whether the architecture supports branches, how much data it can output per kernel or the identification string of the assembly language program. For the GPU architectures, the code generation process is in general the same sequence of transformations and generations, governed by profile data. This localisation of differences is one aspect of retargetability.

For GPU architectures in contrast to the SIMD and CUDA targets (Section V.6.b), the code generation follows their own paths, partially using common functions such as functions for if-conversion or inlining (Section V.3.b). SSE and AltiVec backends can use a common infrastructure, but the capabilities of these targets are largely the same and only the syntax of the actual output instructions is different.

V.3 The Frontend: CGiS Code

This section is concerned with the internal representation of the CGiS source code and the operations on it. First, we shall see how CGiS creates the internal structures during parsing. Then we shall cover the transformations of these structures. This description is divided into two parts: One part on the transformations which are necessary to implement CGiS on (particular generations of) GPUs, and one part on general optimisations which may help the performance and are not mandatory.

The general structure of these phases is presented in Figure V.8.

V.3.a Parsing

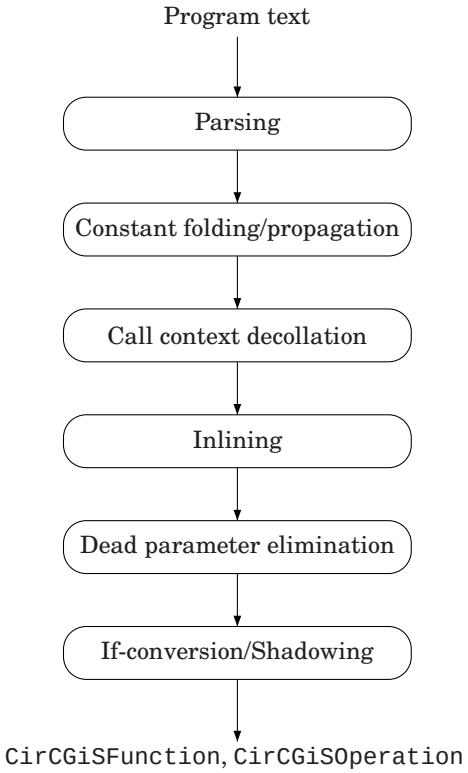
CGiS input is parsed by a classical flex/bison [LMB92] combination. The intermediate representation, that is, the control flow graph and its contents, are built during parsing. This stage also performs type checking and, if necessary, inserts type casting operations; this ensures that all subsequent phases need not worry about type correctness of the considered operations.

Also during parsing, structs are split into their components. Suppose a struct is defined and used as in Program V.4. During parsing, each operation on a struct is split into a multitude of operations on the components. The end result looks like in Program V.5.

Not only does this simplify the rest of the compiler, because henceforth all values to be considered have a primitive type. It also makes it easy to regard a stream-of-structs as a struct-of-streams. Recall Program IV.3 for an example; in terms of Program V.4, the programmer could very well iterate over the stream `X.Cy` with the familiar struct component notation. Internally, the components form streams in their own right anyway, so all that is needed is to provide the appropriate syntax to allow exploitation of this fact.

Arithmetical expressions are generated instruction by instruction. Recall that an expression is not represented as a tree of operations and operands, but as a linear sequence of operations (Figure V.4). The intermediate values are explicitly represented as temporary variables. Also, the workings are mostly independent of the final target; for example, even when a particular vector operation (e. g., `sin`) cannot be implemented on a complete vector, the operation receives the vector as a target, and it is the responsibility of later phases to break up the operation into the componentwise operations (Section V.4). On this level, swizzles are separate operations; the GPU targets can later incorporate them into other arithmetical operations (Section V.5.b).

Figure V.8 Subphases of the frontend



Type change operations are handled differently by the parser depending on the target and one of three different modes of operations in `cgisc`:

- `int` can be processed adequately on `int`-hardware. The parser inserts operations to change the types according to the rules of CGiS (Section IV.2.a). Also, signed and unsigned integer types are different and treated as such. This is the implementation method on the G80.
- `int` can be treated as `float`. In this case, simply all occurences of an `int` keyword are treated as a `float` keyword, except for in external streams. In the streams, we have to retain the original type information, because the user will, after all, submit the streams in `int` format. This affects only the data transfer phases: The generated code must include appropriate type conversion operations. This is the standard implementation method on older hardware. It is exact as long as the values can be represented in the mantissa of 32-Bit floating point numbers, only primitive operations are used and integer division does not produce remainders.
- `int` can be simulated on `float`-only hardware. Basically, an `int` is treated as a `float` with a fractional part of 0. All computations are performed using floating point operations, because the hardware does not allow any other kinds of operations. Then, the results are truncated to simulate a conversion to integers. For

```
PROGRAM struct_extern;

INTERFACE
struct {
  float Cx, Cy;
} coord_t;

extern in coord_t X<16>;
extern out coord_t A<16>;

CODE
procedure f(in coord_t x, out coord_t a){
  coord_t coord = x;
  coord.Cx = 5;
  a = coord;
}

CONTROL
forall(fx in X, fa in A) f(fx,fa);
```

Program V.4: A trivial example using structs

basic operations among integers, the only difference to the previous approach is rounding after division.

Simulating `int` on `float` is only optional behaviour for older hardware. It can support only few cases of real integral operations. Simulating operations depending on the bit-patterns (such as the binary bitwise operations `&` and `|`) by arithmetical operations on floating point values is in general not possible.

As far as signed and unsigned operands are concerned, `cgisc` uses the unspecified, but almost certainly secure[7] way of just using the values interchangeably (Section IV.2.a).

V.3.b Transformations

This section concerns various transformations on CGiS code. Its contents differ from those in Section V.3.c in that the transformations in this section are *necessary* for implementation, whereas the optimisations in Section V.3.c are optional. Some operations on CGiS code fall into both categories, depending on the target. This shall be explained as we go along.

If-Shadowing

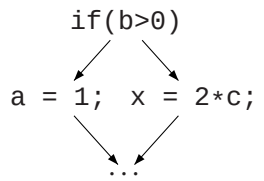
When implementing conditionals, there are basically two choices. They can be implemented as a traditional conditional change of control flow, or using a technique called *if-conversion* [AK02]. By if-conversion, control flow changes are eliminated for the cost of additional instructions; that is, the conditional is translated from a *control flow divergence* to a *data operation*. This is usually done to improve performance, but for GPUs, its uses are more drastic.

[7] Issue 6 of [N07c]: *"This specification says if a value is read a [sic!] signed integer, but was written as an unsigned integer, the value returned is undefined. However, signed and unsigned integers are interchangeable in practice"*.

```
PROGRAM struct_extern;

INTERFACE
extern in float X$Cx<16>;
extern in float X$Cy<16>;
extern out float A$Cx<16>;
extern out float A$Cy<16>;

CODE
procedure f(in float x$Cx, in float x$Cy,
            out float a$Cx, out float a$Cy){
  float coord$Cx = x$Cx, coord$Cy = x$Cy;
  coord$Cx = 5;
  a$Cx = coord$Cx, a$Cy = coord$Cy;
}

CONTROL
forall(fx$Cx in X$Cx, fx$Cy in X$Cy, fa$Cx in A$Cx, fa$Cy in A$Cy)
  f(fx$Cx,fx$Cy,fa$Cx,fa$Cy);
```

Program V.5: Representation of Program V.4 after struct splitting

Figure V.9 Conditionals, real if-conversion, if-shadowing



(a) Conditional control-flow

```
g1 = b>0;
g2 = !g1;
[if g1] a = 1;
[if g2] x = 2*c;
...
```

(b) If-conversion

```
g1 = b>0;
g2 = !g1;
a' = 1;
[if g1] a = a';
x' = 2*c;
[if g2] x = x';
...
```

(c) If-shadowing

Figure V.9.b shows the output of if-conversion when fed with the input of Figure V.9.a. The bracketed condition is to signify that a particular statement has an effect only if that condition holds. Thus, the statement is *guarded* by a condition, by a *guard*. Whereas in Figure V.9.a the difference between the statements to be executed in one case or the other is implemented by branching the control flow to one or to the other statement, in Figure V.9.b *all* statements are visited. Which statements actually produce an effect is determined by the guards.

There is a trade-off to consider in whether or not to use if-conversion. This tradeoff is a matter of optimisation, and thus the topic of Section V.3.c. Here we are concerned with implementing conditionals on hardware which does not support proper control flow, which means on the NV30 generation. Unfortunately, this hardware also does not support guarded instructions, rendering the fall-back to if-conversion impossible.

With this resort closed, there needs to be another way to implement either conditionals or guards. Without any means of data-dependent control flow, the only possibility is to simulate the guards somehow. Here, the LRP instruction (Section II.3.a) comes to the rescue. LRP takes the three arguments λ, x, y and computes $\lambda x + (1 - \lambda)y$. Now suppose

the instruction `[if g] a = b+c;` shall be implemented. Furthermore, suppose that register `Rg` holds a `1` if the guard `g` is true, and a `0` otherwise. Then, the statement can be implemented by the code sequence

```
ADD Rtemp, Rb,Rc; LRP Ra, Rg,Rtemp,Ra;.
```

In this way, the `LRP` instruction can be used as a *guarded assignment*.[8]

Thus, if-conversion as in Figure V.9.b could be implemented. But to avoid having to follow every single computation in a branch with an appropriately guarded assignment, instead the whole computation of a branch is performed on temporary values, and afterwards guarded assignments are used to conditionally transfer the values into the correct registers. This procedure, called *if-shadowing*, is illustrated in Figure V.9.c.

To complete the implementation, a little bit more work has to be done. Consider the program fragments of Program V.6. In (a), we see two `if`-statements subject to if-shadowing. In (b), we see the naïve application of the algorithm, with the obvious meaning of the registers and `[if c]` standing for guarding. Unfortunately, the translation is not correct: Both fragments read from an uninitialised shadow register.

```
if(c)
       a = a+2;

if(c)
       b.x = b.x+3,
       b = b+4;
```

(a) Two CGiS branches

```
       a' = a'+2;
[if c] a = a';

       b'.x = b'.x+3;
       b' = b'+4;
[if c] b = b';
```

(b) Wrong if-shadowing

```
       a' = a+2;
[if c] a = a';

       b'.x = b.x+3;
       b' = b'+4;
[if c] b = b';
```

(c) After renaming

```
       a' = a;
       a' = a'+2;
[if c] a = a';

       b' = b;
       b'.x = b'.x+3;
       b' = b'+4;
[if c] b = b';
```

(d) After initialisation

Program V.6: Problems in if-shadowing

In the first case, this can be remedied by renaming the first operand to the true operand, as displayed in (c). But in the second case, this would also lead to wrong code: Observe in (c) how the second operation reads from a shadow register of which only the `x`-component is defined. To enable the correct working of the code, either the other components have to be initialised correctly, or the `ADD` operation would have to be split into two, one of which reads from the shadow register, and the other one reads from the original register.

This approach surely seems elegant, but is not feasible in the case of *horizontal* operations. Therefore, `cgisc` copies the outer value into the shadow register before the

[8] Luckily, we have at least the `LRP` instruction; otherwise, simulation would remain possible, but get quite intricate [R97, IMR83].

guarded assignments, as displayed in part (d) of Program V.6. It transpires that both shadowings now do not change the semantics.

All in all, the complete algorithm works as follows. A variable used inside a branch which is also visible in the outer scope shall be called *outer variable* here.

(1) Create guard variables for the branches, which hold `1/0` depending on the result of evaluating the condition.

(2) In each branch, replace all occurrences of visible outer variables with fresh shadow variables. (We do not need to shadow variables declared in the branch.)

(3) At the beginning of each branch, introduce initialisation operations for the shadows of outer variables which are read (even if only in part) before written inside the branch.

(4) At the end of each branch, introduce guarded assignments from the shadow variables to the outer variables.

(5) Place the two branches sequentially after another.

For nested conditionals, point (1) in this list has to be modified slightly to also take the outer conditional into account.

As a final note, we turn back to Program V.6.d. In the upper example, it seems at first glance that the initialisation is superfluous. At second glance, that remains so, but a later stage (the copy-elimination stage, see Section V.3.c) removes this unnecessary assignment and transforms that code into that of Program V.6.c (top).

Inlining

Another optimisation which can be used to increase performance in customary systems, but is a necessary prerequisite for implementation on non-recent GPUs, is *inlining*. Inlining means to replace the invocation of a procedure with its body, ensuring that the operations work on the correct variables.

For example, consider Program V.7. Procedure `main1` calls procedure `f` once. By some mechanism peculiar to the hardware (Section V.4.c), the actual `in` values are passed to the procedure parameters, the control is transfered to the called procedure, and afterwards the `out` values are passed to the actual variables and control flow returns to after the function call. This is not possible on architectures not providing said mechanisms.

Procedure `main2` shows a procedure equivalent to `main1`, where the called procedure `f` is inlined. Observe the statements of `f` appearing with the variables of the main procedure.

The cursory explanation is already nearly a complete specification of the actual working. Consider a call of function `f` from `m`.

(1) Transfer the control flow graph of `f` into the one of `m`, replacing the function call statement.

(2) Introduce assignments before and after the code of `f`, to simulate parameter transfer.

Again, a mechanism traditionally used to improve performance has to be employed to ensure implementability. Section V.3.c talks about inlining in the established sense of performance optimisations.

```
procedure f(in float i, out float o){
  if(i>0) o = 0;
  else o = i;
}

procedure main1(in float a, out float b){
  float temp;
  f(a,temp);
  b = 2*temp;
}

procedure main2(in float a, out float b){
  float temp;
  float i = a, o;
  if(i>0) o = 0;
  else o = i;
  temp = o;
  b = 2*temp;
}
```

Program V.7: Inlining

Call Context Decollating

Recall that inside of a kernel, reads from and writes to global memory use static array identifiers (Section V.2.b).[9] It is the responsibility of the runtime system to map the actual arrays to these identifiers.

This cannot be ensured in all cases, when a kernel is invoked from multiple *call contexts*, that is, with different sets of actual parameters. Suppose, for example, that a kernel writes into two streams `A` and `B`. If they reside in the same texture, the kernel needs to use the same identifier for both (write into different components of the same framebuffer); if not, it needs to use different identifiers (write into components of different framebuffers) (Section V.5.a). More profoundly, several kinds of data may be passed to an input parameter: A stream iterator involves a memory lookup, a scalar parameter involves fetching a global program parameter and an index operator involves fetching the interpolated data from the vertex shader. Thus, a single implementation would not work in all contexts. Therefore, the call contexts are *decollated*: A function called multiple times in different contexts is cloned for each context. After decollation, code generation can be performed specific to each call context.

V.3.c Optimisations

Two classical program transformations are employed in the front end, namely limited forms of dead code elimination and constant propagation.

Dead Parameter Elimination

Dead code elimination [AK02] (see also Section V.5.b) is the process of removing computations which do not influence the result of a program. These are all computations which influence the values of variables visible to the outside, that is, of all `out` variables. These

[9] That is, they read from a fixed texture stage and write into a fixed renderbuffer.

entail both the arithmetical instructions which compute the actual value (data dependencies) and instructions which determine the control flow, that is, which arithmetical instructions do take part in the computation (control dependencies).

Dead code turns up most often as a result of optimisations, not in the actual code written by a programmer [M97]. For this reason, `cgisc` does not implement full dead code elimination on the CGiS level.[10] In the front end, only dead parameters are eliminated. These can turn up in particular when structures are passed to functions.

```
PROGRAM struct_partly;

INTERFACE
struct {
  float4 c1, c2;
} struct_t;

extern in struct_t SIN<8>;
extern out struct_t SOUT<8>;

CODE
procedure main(in struct_t s_in, out struct_t s_out){
  s_out.c1 = s_in.c1, s_out.c2 = s_in.c1;
}

CONTROL
forall(a in SIN, b in SOUT) main(a,b);
```

Program V.8: Dead structure components

Consider Program V.8. The programmer writes the code as working on values of type `struct_t`, but it uses only one component of parameter `s_in`. Because structs are split in the frontend, this actually means that one `float4` parameter `s_in$c2` is unused.

The dead parameter elimination phase notes that this (component) parameter is not used in the procedure and eliminates it from the call. This means that the generated code need not hook the texture holding that component as an input to the kernel.

Dead parameter elimination is implemented simply by scanning the instruction stream for any kinds of usage of the parameter. This covers all realistic cases of dead parameters and does not need a full data flow analysis.

Constant Propagation and Folding

Constant folding is the process of evaluating computations on constant values at compile time. This can turn up, for example, when the programmer breaks up a constant in parts for better readability:

```
int size = 1+32; // Base + #rounds
```

A situation coming up more often is that of type conversions: The propagation of the integer constant to a `float` constant in `float f2 = f1+1;` is also a constant computation.

[10] A later phase implements dead code elimination on the level of GPU instructions (Section V.5.b) after other optimisation and transformation phases.

Constant propagation is the process of propagating a constant assignment to a variable through the uses of this variable. Program V.9 shows an example for this. The procedure `func` actually computes `b = 7.8/(a+13.3);`, and that is the computation resulting from constant propagation.

```
PROGRAM const_prop;

INTERFACE
extern in float in_single<16>;
extern out float d1<16>;

CODE
procedure func(in float a, out float b){
  float x = 5.5;
  float y = 7.8;
  float c = a+(x+y);
  c = y/c;
  b = c;
}

CONTROL
forall(a_in in in_single, a1 in d1) func4(a_in,a1);
```

Program V.9: Constant propagation

Constant folding is implemented within the parser while building the internal representation. Constant propagation (including additional constant folding) is implemented as a separate phase using a forward data flow analysis [M97].

Because of the component based nature of the computations, the analysis tracks the constancy and values of individual components of vectorial values. For example, considering the procedure in Program V.10, the value `xy` is only partially constant, which has to be taken into account for the propagation.

```
procedure fold(in float a, out float o){
  float2 xy = a;
  xy.x = 8;
  float xc = xy.x;
  float yc = xy.y;
  o = xc+yc; // 8+a.
}
```

Program V.10: Component based constant propagation

If-Conversion

In Section V.3.b, we have briefly seen introduced the concept of *if-conversion*. For implementation on the NV30 architecture, the related concept of *if-shadowing* had to be used, but later architectures allow real if-conversion. That means that instructions can be amended with a modifier stating that it is to take effect only if a certain condition register has a specific flag set. Figure V.9 shows this in a schematic way on CGiS code. The actual implementation on the GPU is exemplified by Program V.11. As usual, it is

assumed that the value of the condition is present in the form of a `1` or `0` standing for `true` respectively `false`.

```
                        # Suppose Rb holds 1/0 for b/!b.
if(b>0.0) c=3.0;        MOVC DUMMY, Rb;
else c=4.0;             MOV Rc(NE), 3;
                        MOV Rc(EQ), 4;
```

(a) CGiS code (b) Translation with if-conversion

Program V.11: If-conversion on the GPU

To understand this, some facts about the condition registers on GPUs have to be noted (see also Section II.3.a). The `MOVC` instruction moves the value of condition `b` into a dummy register. The value of that dummy register is unimportant, it matters only that during this process of the condition code register is set. For our concerns, the important thing is the *zero flag*, which is set for those components for which the condition is equal to `0`. Because of the use of replication of a single condition, this means that all components are then set or unset according to the condition. To use this, the `MOV` inside the branch is marked as using the condition code register. The `(NE)` modifier means that it updates exactly those components for which the *zero flag* is not set, which is, of course, exactly what we want to do.[11] Then, the same operation is applied to the `else` branch, resulting in a neatly if-conversioned conditional. In this particular case we can use the dual to the `(NE)` modifier: `(EQ)` updates only those components for which the zero flag is set. In case there had been operations modifying the condition code registers before (i. e., nested conditionals), the flag register would have had to be reinitialised with `Rb`.

This shows *how* `cgisc` can perform if-conversion. It remains to be seen *when* it should do so. In contrast to if-shadowing, which is a replacement for conditionals on incapable hardware (Section V.3.b), condition code registers are available on the same hardware that would also support proper conditionals. Clearly, the benefits of using one or the other implementation have to be compared.

On the one hand, a branching control flow can lead to problems. When a branch prediction unit mispredicts a branch, the wrong instructions are fetched, and their speculative executions have to be rescinded [HP03]. This is a problem in classical CPUs. In GPUs, control flow deviations lead to a severe problem for the SIMD based hardware. *SIMD* here means not the vectorial instruction, but the fact that instructions on vectors are processed in batches; that is, a set of fragments is divided into subsets for which the same instruction has to be executed. For example, on the G70 architecture, the minimal size of those subsets is 880 fragments [N06]. Thus, a branch where one path is taken by 400 fragments T and another one by 300 fragments F, needs to be executed in two batches. In the first batch, the T-branch would be executed for all 800 pixels, discarding the computation for the F pixels; in the second batch, the opposite would happen. On the G80 hardware, the granularity is going down to 32 pixels, which ameliorates the problem.

Nevertheless, tests have shown that on the NV40 architecture, true branches are always at least as fast as converted branches. Thus, per default, branches are not converted

[11] This is a simplified version. Actual code would employ additional static masking to cope for the fact that multiple values are present in a physical register (Section V.5.c). An instruction which uses both the condition code register and a static mask writes only into those components unlocked by *both* kinds of restrictions (they are subject to an $\wedge$, not a $\vee$).

on that profile. On the G80 architectures, converted branches are (very slightly) faster if the branch bodies are short. The heuristic uses a threshold of 5 instructions: Larger branches are kept, smaller branches are subject to if conversion. However, again the programmer can use hints to overrule that decision. When used at the beginning of a condition body, the hint `force_conversion` specifies that it should use if-conversion, and `no_conversion` specifies that the implementation should use proper conditionals. Recall that hints may be prefixed with a profile specifier to restrict their applicability to cater for differences such as the mentioned behavioural change from NV40 to G80.

General remarks. After the presented operations and transformations, the CGIS code has been transformed in such a form that it can structurally be implemented on graphics hardware. This concerns mainly the control flow capabilities and workarounds for restrictions of these. The next sections are concerned with translating that code into a form suitable for the hardware regarding instructions (Section V.4) and data (Section V.5).

V.4 Code Generation

The generation of GPU code works as a mapping from CGIS instructions (Section V.2.a) onto primitive GPU operations. Thus, it performs the tasks of instruction selection and instruction ordering simultaneously. Figure V.10 shows the subphases of code generation. The result is a correct implementation of the kernel currently under consideration. It is amenable to further modifications in later stages of the compiler; see Section V.5 for more on this.

There are a multitude of targets to consider. On the one hand, `cgisc` targets several GPUs. The CUDA backend needs a quite different kind of code generation, and the SIMD backend another one, in turn. A generational approach to code generation, that is, a *code generator generator*, is indispensable for a readable and extensible specification of code generation.

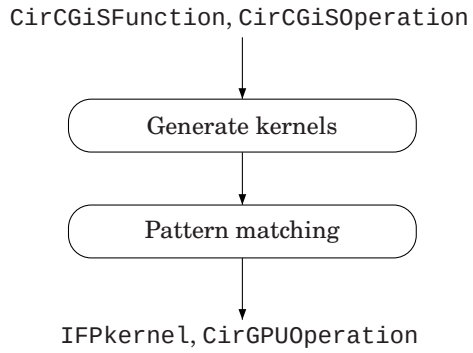
To this end, `cgisc` employs the pattern matching system `OORS`. `OORS` was developed for `cgisc` to express code generation and optimisation, but it is a general code generator generator and optimiser generator [G06, GL07]. Section V.4.a introduces the `OORS` system as a whole, and Section V.4.b describes how `OORS` is employed in `cgisc`. That section also explains some of the more interesting features of the translation process. Section V.4.c is concerned with the translation of control flow.

V.4.a Pattern Matching with OORS

In the most general sense, `OORS` is a *pattern matcher*, which receives as its input a string of attributed elements and creates as its output a string of attributed elements. This is performed first by searching for *patterns* in the source string. A pattern is a possibly non-contiguous sequence of elements with specific attributes. Second, the matched pattern gives rise to some new elements with attributes dependent on the actual elements and attributes of the matched pattern. These then get inserted into the output string. The matching process is successful if every element in the input string is matched exactly once.

For its use in code generation, this amounts to receiving a sequence of operations and creating another sequence of operations. As a trivial example, we might want to create

Figure V.10 Subphases of code generation



assembly code for an addition with a constant. In this setting, the operation `a = b+2;` should give rise to `ADD Ra, Rb,2;`, where `Ra` and `Rb` are the registers for `a` respectively `b`. Symbolically, this would mean:

(1) In the input sequence, search for an occurrence of the pattern `?1 = ?2+?3;` (the *element*), where `?1` and `?2` are variables and `?3` is a constant (the *attributes*).

(2) Create a new element `ADD !1, !2,!3;`, where `!1` and `!2` are the registers for the variables `?1` respectively `?2`, and `!3` =`?3`.

(3) In the output stream, insert that instruction at the position corresponding to the position of the instruction in the input stream.

Obviously, a one-to-one correspondence is not particularly interesting. The beauty of the approach comes from two features: On the one hand, matching and replacement rules might be more complicated, allowing for code generation in a realistic manner; on the other hand, rules and sets of rules can be defined using inheritance relations with specialisations, so that multitarget backends can be implemented with relatively small effort.

As an example of the first property, consider the case where one target instruction can implement a multitude of source instructions. For example, the target hardware might support a multiply-add-operation. In `OORS` terms, the search pattern would be represented as `[A = B*C; ... X = A+D;]`, that is, *two* instructions with particular attributes. The ellipsis (a *wildcard*) mentions that the operations need not be adjacent: The replacement rule will match both instructions, and the generated instructions will be inserted sequentially at the position corresponding to the position of the first matched instruction. In this case, the generated instruction sequence would consist of a single instruction: `[MAD RX, RB,RC,RD;]`, where the `R?` are the registers corresponding to the variables.

Obviously, the presence of the ellipsis, that is, of other interjacent instructions, can lead to problems, if the matching is performed in a naïve way. For example, were the sequence `A = B*C; D = A+1; X = A+D;` matched by the rule, the order of the latter two computations would effectively be turned around. This is prevented by what `OORS` calls *implicit conditions*, which ensure that in presence of a reordering of instructions,

this cannot affect the semantics; when it would, the pattern cannot be matched. Basically, the implicit condition is a hand-written check which investigates the operands and targets of the instructions and checks for data dependencies. This check is automatically called for any search pattern featuring a wildcard.

It is obvious that with more complicated rules, a single string might be matched in several ways. To this end, OORS rules have a *cost*, and the matcher performs those matchings with the best overall cost. This can be done either in a greedy way, where at each step the locally best match is detected, or in a global matching approach which detects the matching with the overall least cost.

As an example of the second important OORS feature, the inheritance mechanism, consider the task of writing code generators for different, but similar architectures. This is the case, for example, for the various GPU generations. To enable easy retargeting, matching rules are organised in sets called *profiles*, following the nomenclature for CGiS' targets. Profiles can inherit from another profile, overriding certain rules, omitting others and adding some. For example, the backends for NV30, NV40 and G80 GPUs are quite similar, but different in their support for loops. The NV30 does not offer any control flow instructions. The NV40 backend adds to the NV30 backend instructions for looping with predefined upper bounds. The G80 has deprecated these instructions, so these rules are removed again for the G80 backend, and other rules targeting the G80 instructions are added.

Apart from the code generation method, OORS also supports code *optimisation*. This follows the same basic usage model as in the generation mode: Rules are specified with patterns on elements and attributes, which give rise to other elements and attributes. The difference to the generation mode is that the optimisation mode operates *on one string* instead of *between two strings*. That is, an existing string is matched and transformed, and this process is repeated until no further matchings are possible. For code optimisation, this amounts to classical peephole optimisation.

Much more information about OORS including its precise syntax, a formal and a readable description of its algorithm and several examples, is available in [G06]. OORS is available as open source software.

V.4.b Pattern Matching in cgisc

For the task at hand, code generation in `cgisc`, the input to OORS is the sequence of `CirCGiSOperation` objects in a block of a procedure, and the output is a sequence of `CirGPUOperation` objects in the corresponding block of the corresponding kernel. Therefore, before the pattern matching process starts, the control flow graph of the `CirCGiSFunction` is copied into the `IFPkernel`. The pattern matcher normally runs in local, greedy matching mode, because the global matching mode takes too long to execute for little gain, if any; if so desired, the user can turn on global matching with a command line switch.

Many arithmetical instructions can be translated in a straightforward way. For example, an addition `a = b+c;` creates symbolic registers for all variables and issues an instruction `ADD Ra, Rb,Rc;`. At this point the compiler is neither interested in nor knowledgeable about the actual registers, that is, about the allocation of the symregs into the actregs (Sections V.2.b and V.5.c). However, it is vital to know the *width* of the operands, because not all operations can be translated as such.

For example, some operations, such as `LN2` computing logarithm to base 2, are natively available only for single components. An operation `a = ln2(b);` for a vectorial `b` therefore has to be translated into a sequence of single-component operations like

```
LN2 a.x, b.x; ...  LN2 a.w, b.w;.
```

Such a pattern turns up oftentimes, also as a subpattern of larger translations. For example, `a = ln(b);` is translated into a sequence of *scalar* `LN2` instructions for the components of `b`, which is then subject to a *vectorial* multiplication with the constant $\ln 2$ ($\log_2 b \cdot \ln 2 = \ln b$). The replacement pattern can turn out to be highly complex; for example, a left rotation is translated using the identity

```
x <<< y  =  (x<<(y&31)) | ( x>>(32-(y&31)) ),
```

where the << and >> have to be issued componentwise, and the other GPU operations can be vectorial.

Other operations have to be translated by a multitude of rules, depending on specific conditions. Consider the seemingly simple dot product `b\c`. If `b` and `c` have width 3 or width 4, this can be implemented by the special operations `DP3` and `DP4`. For width 2, later GPUs support a `DP2` operation, but on the NV30 it has to be implemented by separate multiplications and an addition. For width 1, the dot product degenerates into a simple multiplication. And this all is irrelevant for integers, which are not supported by any of the `DP`*n* instructions, necessitating a sequence of `MUL` and `MAD` operations.

While this might sound daunting at first, it actually shows how to *localise* issues of hardware differences. For example, a general rule for binary vectorial operations can cope with the dot product in general. The NV30 overrides this rule for two-component operands, and the G80 provides special rules for integer operands. Thus, a large part of the instruction selection process can be retained; rules have to be amended with special conditions if they are to be restricted, and new rules for new features have to be introduced.

Now, it is worthwhile to look at a few specific rules to illustrate certain aspects of the compilation. Program V.12 shows a fragment of the translation rules in a very simplified form. The merging rules are responsible for *vectorisation*. Two statements of the form `a1 = b1+c1; ...  a2 = b2+c2;` can be executed in parallel, if the operands `ai` etc. fit into one register each. Section V.5.c explains in more detail how this is finally implemented; the rule in Program V.12 simply shows in general how the opportunities for this optimisation are found.

The search pattern searches for two possibly vectorisable operations. `OORS` uses the `$`-notation familiar from YACC and similar tools, where `$$` represents the current item (here: operation) and `$1`, `$2` etc. represent the items in order of their appearance in a rule (counting search and replacement patterns together). The condition checks the side conditions for this optimisation. For example, it would not be possible to vectorise the sequence `a1 = b1+c1; a2 = b2+7;`, because the right operand is constant in one operation and a variable in the other one. The *implicit condition* (Section V.4.a) already ensures that the rule is matched only when there are no dependencies requiring that the latter matched operation cannot be lifted in front of that sequence. The replace pattern then generates a `CirGPUOperation` for the two `CirCGiSOperation`, *merging* the operands and the targets into one special vector register. (Refer to Section V.5.c on how this is used later in the code generation process.) The rule is flagged with a certain integer constant, which is `OORS`' way of marking a rule as *optional*: It is activated only

```
rule merge_un : ( 8 ){
  search: [ CirCGiSBinOp(is_vector_arith($$->opcode()),
            *,
            CirCGiSBinOp(is_vector_arith($$->opcode())
          ]
  condition: {
               // The operations must have equal opcodes...
               // ... and equal constancy.
               // The registers must not already be wrapped...
               // ... they must have the same base type...
               // ... they must not be the same...
               // ... they must not be at a fixed position.
               // We must have space for wrapping.
             }
  replace: [ CirGPUBinOp(
               to_gpu($1->opcode()),
               new CirWrapperReg($1->operand1()->symreg(),
                                 $3->operand1()->symreg())
               new CirWrapperReg($1->operand2()->symreg(),
                                 $3->operand2()->symreg())
               new CirWrapperReg($1->target()->symreg(),
                                 $3->target()->symreg())
             )
           ]
}
```

Program V.12: A skeleton of a vectorisation rule

conditionally, in this case when the user switches on the optimisations by a command line flag of the compiler. The reason for that is that opportunities for this optimisations present itself only seldomly in CGiS code; vectorisable code tends to be expressed already in vectorised form thanks to CGiS' appropriate vectorial types, and occasional superword level parallelism has no effect on the actual execution time, but introduces additional burdens on the compiler by constricting register placement. This wrapping were more useful when extracting parallelism out of *sequential* code; the CGiS syntax is already too advanced for that.

Just as an example of an optimisation rule, Program V.13 presents a rule which merges multiplication and addition into a multiply-and-add operation. It is presented here to complete the treatment of pattern matching in `cgisc` although this optimisation is executed much later, not in the current phase.

The search pattern looks for two instructions to merge which have to be set up in such a way that the operands align and the target of the multiplication is overwritten. The condition ensures, together with the implicit condition, that the reordering of the addition does not change the semantics of the code. The cost is negative, because the number of operations is *removed*.

The replacement pattern shows exactly one instruction. The `OORS` semantics mean that this instruction is generated and takes the place of the first matched instruction. The instructions of the search pattern could be marked in the search pattern to be kept, but as they are not, and because they do not turn up in the replacement pattern with the $-syntax, they get deleted from the instruction stream.

```
rule mad {
  search: [ CirGPUBinOp($$->opcode() == OP_GPU_MUL),
            *,
            CirGPUBinOp($$->opcode() == OP_GPU_ADD &&
                        (op_is_reg($$->operand1(),$1->target()) ||
                         op_is_reg($$->operand2(),$1->target()) ) &&
                        ($$->operand1()!=$$->operand2()) &&
                        $$->target() == $1->target())
          ]
  condition: {
             // The operations must not be differently guarded...
             // ... and have to use the full registers.
             }
  cost: { return -1; }
  replace: [ CirGPUTerOp(OP_GPU_MAD, $1->operand1(), $1->operand2(),
                         $3->operand2(), $3->target() ]
             // Copying of masking, negation and guarding omitted.
}
```

Program V.13: A skeleton of a peephole optimisation rule

V.4.c Control Flow

As has been said, the `OORS` pattern matcher works solely on the level of basic blocks. Control flow issues come into effect, however, because the generated instructions need to incorporate the GPU instructions for control flow. The final code is, after all, just a linerisation of the representations of the instructions inside the blocks (Section V.2.d). Thus, high-level pseudo-instructions for control flow are inserted into the stream of `CirCGiSOperation` of the branch and merge blocks. These are then matched and translated into the appropriate `CirGPUOperation` objects.

Note that at this point only those control flow constructs are present which are actually available on the hardware. That is, all decisions about how to implement conditionals and whether functions must be inlined have already been taken before (Section V.3).

True Conditionals

Native conditionals are expressed by special pseudo-instructions which mark the beginnings of the two branches and the end of the complete conditional. That is, one does not generate labels to jump to, but the assembler generates the jumps based on the directives.

Which of the two branches to jump to is based on the contents of a condition code register, much in the same way as guarded instructions; see Section V.3.c. There it is also explained how a condition is used to a set or unset a condition code register.

Conditions themselves are represented as `float` values of 1 or 0, as this is what certain set-on-condition operations on `float` return.

Programs V.14 and V.15 give a symbolic view of the translation of a conditional into GPU code. The `IF` instruction receives as an argument the condition flag determining which of the two branches to follow. Program V.14 shows the source code; observe the hint to enforce translation into a true conditional (Section V.3.c).

```
if(a<1.0) #HINT(no_conversion)
  e = 0.5;
else
  e = -0.5;
```

Program V.14: A conditional in CGIS

```
SLT Rcond, Ra,1;   # Evaluate condition.
MOVC Rcond, Rcond; # Set or unset condition code.
IF NE;             # Start true branch.
MOV Re, 0.5;
ELSE;              # End true branch, start false branch.
MOV Re, -0.5;
ENDIF;             # End false branch.
```

Program V.15: Program V.14 translated to NV40 code

Loops

Loops, just like conditionals, are implemented by enclosing the body with special loop begin and loop end markers. The loop condition is implemented by a data-dependent break instruction just after the formal loop start (in `while` loops) or at the end (in `do` loops).

The looping is implemented with different mechanisms dependent on the hardware. In the NV40 hardware, GPU loops need to be issued with a fixed number of iterations. This number must not be larger then 256. Obviously, both of these conditions are rather harsh. Fortunately, there is a data-dependent break instruction, and loops can be nested. Thus, these restrictions can be virtualised in CGIS: consider a hardware with maximally l loop iterations. A loop with maximally n iterations for $n > l$ is implemented as a two-layered loop nest, where the outer loop is issued with an iteration number of l and the outer loop with an iteration number of $\lceil n/l \rceil$. A data-dependent break instruction is used to break out of the loop, enabling dynamic (and divergent) loops.

The compiler does not perform a sophisticated analysis to compute the maximal iteration number. Instead, the user can add a hint to a loop specifying this number. If none is given, l^2 is assumed.[12] This is particularly useful if the number is assured to be less than l, because then only *one* GPU loop suffices. Program V.16 shows such a hint in the CGIS program. Program V.17 shows the implementation with a NV40 style loop.

```
for(int i=from_to.x; i<from_to.y; i=i+1)
#HINT(max_count=64)
{
  temp=temp+i;
}
```

Program V.16: A loop in CGIS

[12] The maximal number l^2 of iterations is not a real restriction because the hardware such restricted also has constraints on dynamical instruction count.

```
MOV Ri, Rfrom_to_x;          # Initialise loop.
REP {64};                    # Start loop.
SLT Rcond, Ri,Rfrom_to.y;    # Evaluate condition.
MOVC Rcond, Rcond;           # Set or unset condition code.
BRK (EQ);                    # Exit if condition false.
ADD Rtemp, Rtemp,Ri;         # Loop body.
ADD Ri, Ri,1;                # Loop iteration.
ENDREP;                      # End of loop.
```

Program V.17: Program V.16 translated to NV40 code

Function Calls

In absence of a parameter stack, procedure parameters have to be passed in registers. To this end, the subprocedures work on their own sets of registers. For each call context, the parameters are passed to these registers, the control flow is transfered to the subprocedure, and afterwards the values of the formal output parameters are passed into the actual parameters.

Programs V.18 shows a simple example. Note the hint for function `help`: To guide the compiler in much the same way as for if-conversion (Section V.3.b), hints can declare whether a function should be inlined (`force_inline`) or not (`never_inline`), assuming an architecture allows the choice. Per default, all procedures are inlined.

Program V.19 shows the relevant fragment for this program symbolically. Observe how the subprocedure entry point is identified by a label, and the entry point of the main program is identified by the reserved label `main`.

```
procedure help(in float i, out float o)
#HINT(never_inline)
{
  o = 2.0*i;
}

procedure f(in float i1, out float o1){
  float f1t1, f1t2;
  help(i1,f1t1);
  help(f1t1,f1t2);
  o1 = f1t2+1;
}
```

Program V.18: A simple function call

V.5 The Backend: GPU Code

When the code has been generated in a symbolic form, the work is far from being finished. Although the *instruction* selection process is, by and large, finished, almost all relevant decisions regarding the *data* still remain. This holds inside kernels and concerns mainly the allocation of symregs to actregs, that is, the mapping of values to registers. It also holds with respect to the outside world, that

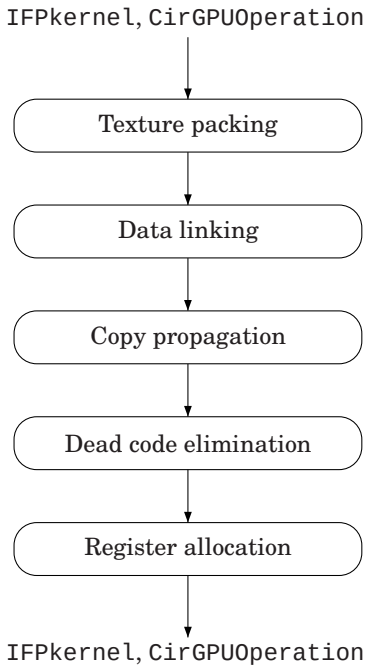
```
help:
  MUL Ro_help, 2,Ri_help;   // Procedure body of help.
  RET;
main:
  MOV Ri_help, Ri1_f;       // Prologue.
  CAL help;                 // Call.
  MOV Rf1t1_f, Ro_help;     // Epilogue.
  MOV Ri_help, Rf1t1_f;     // Prologue.
  CAL help;                 // Call.
  MOV Rf1t2_f, Ro_help;     // Epilogue.
  ADD Ro1_f, Rf1t2_f,1;     // Statement of f.
```

Program V.19: Program V.18 translated to G80 code

is, concerning the input and output of data: Recall that we have free reign over the data placements.

There are also other, more classical optimisations to be done on the GPU code; in particular, *copy propagation* and *dead code elimination* (Section V.5.b) can profit from the swizzling and masking features of GPUs. Figure V.11 gives an overview of the structure of the backend.

Figure V.11 Subphases of the backend



V.5.a Texture Packing

We have now arrived at a point where the kernels have been created in a symbolic form. The next task is to compute the mapping of streams into textures. We shall call this

phase *(texture) packing*.

The number of input stages and especially of output buffers is limited. The more pressing parameter is the number of output buffers, which is merely 1 in older generations, changed to 4 later, and is at most 8 in recent hardware. Each output buffer can hold up to 4 components. Packing has to take that restriction into account and to create a packing which is implementable on the hardware of choice.

First, let us fix a few constraints on the program and the data to be able to explain the algorithm more easily.

Preliminary Considerations

Program V.20 shows that the packing choice cannot be made locally. Suppose we are working on an architecture with only one output buffer. Each procedure can make do with a particular mapping of the X-streams into textures, but there is no way to create a packing satisfying both calls.

```
PROGRAM copykern;

INTERFACE
extern in float A<16>;
extern out float2 X1<16>;
extern out float2 X2<16>;
extern out float2 X3<16>;

CODE
procedure f(in float i, out float2 o1, out float2 o2){ ... }

procedure g(in float i, out float2 o1, out float2 o2){ ... }

CONTROL
forall(a in A, x1 in X1, x2 in X2) f(a, x1,x2);
forall(a in A, x1 in X1, x3 in X3) g(a, x1,x3);
```

Program V.20: A program illustrating the need for global texture packing

To remedy this, `cgisc` introduce *copy kernels*. A copy kernel is a small CGiS procedure which takes inputs from some textures, reorders them in some way, and outputs them into other textures. For example, suppose in Program V.20 we have a mapping of `A` in one texture, `X1` and `X2` in another one, and `X3` in another one. `cgisc` would then introduce copy kernels after procedure `g`. That procedure would write into temporary space, and the copy kernels would then reorder the temporary data into the correct textures.

Program V.21 shows a representation of how such a copying might look like. After `g` has written into temporary space, texture 2 (comprising `X1` and `X2`) gets updated with the new `X1` (in iterator `t1`) and the unchanged `X2` (in iterator `x2`), and texture 3 (holding `X3`) gets updated through iterator `t2`.

Such copy kernels are introduced whenever a texture packing satisfying all kernels cannot be found. In this case, the best packing (with respect to the yet to be mentioned heuristics) of those needing the least number of copy kernels is chosen, and the appropriate copy kernels are inserted in those place where the hardware framebuffer restrictions demand so. Therefore, we may assume without loss of generality in the discussion of the heuristics that the hardware can implement the selected mapping.

```
PROGRAM copykern;

INTERFACE
extern in float A<16>;
extern out float2 X1<16> : packing(1) : RG;
extern out float2 X2<16> : packing(1) : BA,
extern out float2 X3<16> : packing(2) : RG;
intern float2 Tg1<16>;
intern float2 Tg2<16>;

CODE
procedure f(in float i, out float2 o1, out float2 o2){ ... }

procedure g(in float i, out float2 o1, out float2 o2){ ... }
procedure copy_g1(in float2 i1, in float2 i2,
                  out float2 o1, out float2 o2){
  o1 = i1; o2 = i2;
}
procedure copy_g1(in float2 i1, out float2 o1){
  o1 = i1;
}

CONTROL
forall(a in A, x1 in X1, x2 in X2) f(a, x1,x2);
forall(a in A, x1 in X1, x3 in X3, t1 in Tg1, t2 in Tg2, x2 in X2){
  g(a, t1,t2);
  copy_g1(t1,x2, x1,x2);
  copy_g2(t2, x3);
}
```

Program V.21: Representation of Program V.20 with automatic copy kernels

Another point of interest is the call context decollation which was mentioned in Section V.3.b. To reduce the dependencies on texture packing, procedures are cloned for each call context, and the packing choices are therefore more localised; otherwise, different streams (from different call contexts) would have to be packed in the same way to have compatible data accesses. Thus, again without loss of generality we may assume that each user-specified procedure is called exactly once with one set of streams passed through iterators to the function parameters.

Global and Local Considerations

With the specified setting, we can investigate the actual packing process. Several questions come to mind:

(1) What parts of the code generation and the runtime workings are affected by the packing?

(2) Which characteristics of packings are desirable in these parts?

(3) How do we select a desirable packing?

We start with the investigation of (1) and (2) together. (3) is implemented with a simple generate-and-visit method which is explained afterwards.

Texture packing has the following consequences.

(a) It determines the number of input and output stages and thereby the number of memory accesses in a kernel.

(b) It determines whether a kernel can use *framebuffer objects (FBOs)*, a method to directly write into texture space, or whether it needs to write into the general-purpose framebuffer and afterwards copy into the texture memory.[13]

(c) It governs whether input data received by the API functions need to be reordered or can be uploaded in a whole chunk as a texture.

(d) It might waste space when a texture for n components per element is used only by $m < n$ components per element.

Of these considerations, (c) is of particular importance. Suppose a texture holds two streams A and B of type `float` in components `x` and `y`, respectively. First of all, this means that the elements of the CPU arrays have to be reordered into this interleaved ordering, putting an additional burden on the data transfer times. Second, consider a kernel which reads from stream A and writes into B. Because a texture cannot be bound both as an input buffer and an output buffer (via the FBO method), we have to write into the temporary framebuffer and afterwards copy the data back into the texture ((5) in the example in Section V.1.c). For this to work, we have to introduce *compensation copies*. Because we cannot selectively update only the `y` components of the texture, we have to introduce an operation into the kernel which is solely responsible to copy the values of the A stream. These operations are called compensation copies. In this case, they do not cause much harm, because the stream A is read anyway. However, for the same reason of the non-selective upgrading, we have to introduce compensation copies even if the kernel only writes into B and would not normally read from A; and this, in turn, means that the texture must be bound as an input buffer and thus we cannot use FBOs. So, all in all, reorderings should be avoided.

After this global consideration, let us now focus our attention to single kernels. The implementation favours FBOs, as they have been found in general to provide (minor) speed advantages above copy-to-textures. Again, this was found to depend on the driver version. Therefore, the user can again guide the heuristics by providing a hint: Providing the hint `no_FBO` to a procedure (between parameter list and body) disables the generation of FBO code for that procedure.

With respect to single kernels, we can argue about the *badness* of a partitioning for that particular kernel, which is a representation of the burden faced in implementing that kernel with that packing. Let π be any packing, and k be any kernel. We assume that π is a *valid* packing for that kernel, meaning that k can be implemented with that packing. Let I be the set of textures used for inputs, and O the set of textures used for outputs. Then, the local badness is defined as

$$b(\pi, k) = 50 \cdot |I \cap O| + (|I| + |O|).$$

In effect, this just means that badnesses compare first with respect to the necessary number of texture copies (needed when a texture is used both for input and output) and secondary with respect to the necessary texture stages and buffers.

[13] This is something different than the copy kernels (Section V.1.c).

In all that we have to consider compensation copies. There are two sources of compensation copies. In general, compensation copies must be introduced if a texture is only partially written, that is, if only one stream in a texture is being written. That is definitely the case when the packing has decided to pack several streams into one texture; but it may be the case if the texture in question is an external texture and we do not know whether other streams are present in there used by other programs. Therefore, if the user does not supply the hint `no_texture_reuse` (Section V.1.c), the algorithm has to consider the worst case and issue compensation copies.

Computing a Packing

When considering the global badness of a packing, the reordering costs come into effect. For a packing π, let $\omega(\pi)$ be the number of components which have to be reordered. For example, if a struct with four components resides suitably aligned in the same texture, then $\omega(\pi) = 0$. If a single component lies in another texture, then all components have to be part of the reordering, and thus $\omega(\pi) = 4$.

Conceptually, the packing algorithm works as follows.

(1) Create valid partitionings of streams into textures, that is, packings which do not violate size or type constraints or the user specifications. This is done with an algorithm from [H63, K05].

(2) Check the partitionings against all kernels.

- If a partitioning π is valid for all kernels k_i, let $b(\pi, k_i)$ be the associated badnesses. The global badness is then $B(\pi) = 1000 * \omega(\pi) + \max_i b(\pi, k_i)$. We keep the mapping with minimal $B(\pi)$.
- If a partitioning is not valid for a number v of kernels, and we have not found a valid mapping or a mapping violating a smaller number of kernels, we record the bad mapping. Again, we keep the mapping π with minimal $B(\pi)$.

(3) If we have found any valid mapping, this is the resulting packing.

(4) Otherwise, take the recorded best invalid mapping, create appropriate copy kernels and take that as the resulting packing.

All in all, this means that the packings are compared first and foremost with respect to their reordering costs, and only secondary with respect to their effects on kernels.

After the compiler has created the texture packing, the necessary information is stored in the `foralll` loops: which textures to hook to which input phases and to which output buffers, what kinds of copies to issue and what FBOs to use.

Linking

The *linking* phase concludes the texture packing process. This phase inserts the inserts the location information about the data into the in- and output instructions inside the kernels. This entails the following activities:

- creating the compensation copy instructions
- patching the information about where to fetch data and whereto to write data into the instructions
- precolouring the symregs used in texture fetch instructions

The precolouring phase will be used in the register allocation process, to be explained in Section V.5.c.

V.5.b Optimisations

Peephole Optimisations

A few peephole optimisations have been implemented with OORS. The MAD rule of Section V.4.b is one example for this. There are also some other rules which can perform constant folding or merge the setting of a condition code register with the computation of the boolean value. Most optimisations on GPU code in cgisc, however, are formulated as a data-flow analysis working on the control flow graph and with worklists of instructions; casting these into the pattern matching syntax of OORS is possible, but not useful.

Copy Elimination

Copy elimination is the process of removing superfluous assignments [M97]. It consists of *copy propagation*, which propagates the source of an assignment to replace the usage of that assignment's target, and the elimination of the then useless assignments. The elimination is done by standard dead code elimination and thus is the subject of the following section.

```
MOV R2, R1;
MOV R3, R2.x;
MOV R4, -R2;
ADD R5, R3,R4;
```

(a) Before copy elimination

```
ADD R5, R1.x,-R1;
```

(b) After copy elimination

Program V.22: Copy elimination on GPU code

Copy propagation is an interesting topic in GPU assembly code generation, because the operations support various modifiers on their operands. In cgisc, both swizzling and negation are taken into account. Program V.22 shows the result of copy elimination (assuming that there are no further uses of registers R2, R3 and R4). Eliminating the move from R1 into R2 is standard for copy elimination. To include the swizzling (here: replication) and the negation as operand modifiers into an arbitrary operation is a feature of the GPU assembly language.

Copy propagation is implemented as a forward data flow analysis. It works in much the same way as constant propagation on CGiS code, except for the fact that negation and swizzling have to be carried on as well.

Dead Code Elimination

Liveness analysis is a prerequisite for register allocation. As mentioned in Section V.2.b, the fundamental unit of computation in CGiS is the *component*, not the value. Thus, a liveness analysis has to work on components of symregs instead of symregs. That and the handling of guarded statements are the only differences to standard liveness analyses [NNH99].

More importantly, the liveness analysis is followed by *dead code elimination*. Only seldomly does a programmer write really dead code, that is, code which is never executed or does not contribute to the externally visible result. More often, dead code comes

up during a particular optimisation. As an example, the first three instructions of Program V.22.a become dead after copy propagation, if there are no uses of R2, R3 or R4 later on. Thus, dead code elimination removes these instruction, resulting in Program V.22.b.

Dead code elimination is implemented as a backward data flow analysis. It eliminates dead code by removing instructions and by inserting masks and For example, consider the transformation in Program V.23. For ease of exposition, we assume the registers have width 2. Part (a) shows that a vectorial value is only partially overwritten. Part (b) shows the result of component based dead code elimination.

```
MUL R1, R2,R3;          MUL R1.y, R2.y,R3.y;
MOV R1.x, 0;            MOV R1.x, 0;
```

(a) Before dead code elimination (b) After dead code elimination

Program V.23: Component based dead code elimination

V.5.c Register Allocation

Any compiler outputting assembly code in the end has to perform some sort of register allocation. In this phase, the scant physical resources available to store intermediate values are designated to hold the operands of the instructions in the intermediate representation [WM95]. In cgisc, these operands are the symregs, and they are going to be mapped onto actregs (Section V.2).

Standard Graph Colouring

```
in(a);
b ← a/2;
c ← a+b;
d ← c*b;
out(d);
```

Program V.24: An example to demonstrate register colouring

A standard approach based on graph colouring [C82] has to be modified in the setting of CGiS on GPUs. To see this, we recall the graph colouring approach, using as an example some code for a conventional imperative language. Program V.24 presents a small sequence of code with symbolic registers. The task is now to map these symbolic registers onto actual registers.

For example, one could produce a mapping with only two registers, assigning a and b to register R1, and c and d to register R2. The resulting code is presented in Program V.25.a. The alert reader immediately observes that this code actually computes the wrong function: $(\frac{a}{2} + \frac{a}{2}) \cdot \frac{a}{2}$ instead of $(a + \frac{a}{2}) \cdot \frac{a}{2}$. This is because the symregs a and b have been assigned to the same actual register. Thus, the addition instruction gets as input operands twice the same register, which is holding solely b at that moment. Program V.25.b, on the other hand, presents a correct implementation, as is easily verified. The underlying mapping, here, maps a, c and d to R1, and b to R2.

To automatically compute such a colouring, a two-step approach is used. First, the *live ranges* of variables are computed. A live range of a variable is the collection of the points in an execution of a program where that particular variable is live. That *liveness* is

```
in(R1);
R1 ←R1/2;
R2 ←R1+R1;
R2 ←R2*R1;
out(R2);
```

(a) A wrong colouring

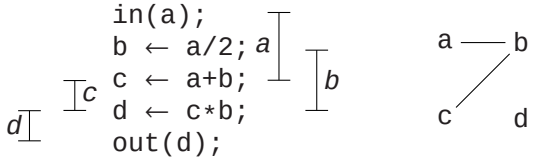
```
in(R1);
R2 ←R1/2;
R1 ←R1+R2;
R1 ←R1*R2;
out(R1);
```

(b) A correct colouring

Program V.25: Register colourings on Program V.24

defined as the existence of an execution path[14] from that point onwards to the end of the execution, on which the current value of the variable is read. In other words, the variable might be read from before it is being written to. Figure V.12.a shows the live ranges for our straightforward, sequential example.

Figure V.12 Live ranges and interference graph for Program V.24



(a) Live ranges (b) Interferences

In a second step, the overlapping[15] live ranges are considered. Variables cannot be held in the same register, if their live ranges *interfere*. Figure V.12.b presents a graph of the live ranges, displayed as nodes, with the interferences displayed as edges between the nodes. Register allocation, then, corresponds to graph colouring: The nodes of the graph have to be assigned colours, such that no incident nodes receive the same colour. Interpreting a colour as a particular physical register, the colouring restriction and the structure of the graph ensure that no two interfering (adjacent) values get assigned the same register (colour). In our example, one could assign the colour blue to nodes `a` and `d`, green to `b` and red to `c`. A minimal colouring would use two colours; assigning one colour to `b` and another one to `a`, `c` and `d` would correspond to the register allocation of Program V.25.b.

Componentwise Colouring

The general colouring algorithm has to be modified for register allocation in `cgisc`. The reason is the peculiarity of the representation and implementation of vectorial values.

- CGiS supports vectorial scalars of various widths and primitive scalars. That is, the length of the CGiS types can be smaller than the width of the native registers.
- GPUs support swizzling and masking (the kind of Section II.3.a, not the CGiS kind of Sections IV.2.c and IV.2.d). By this, the components of actual register components partaking in an operation can be chosen freely, although the number of registers is fixed.

[14] We are considering only static paths.
[15] Overlapping live ranges are also called *intersecting*.

A further point stems from the texture packing of CGIS/`cgisc`.

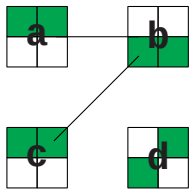
- Texture fetches, that is, array loads, do not support swizzling on the memory. Thus, input data have fixed components, but not fixed registers.

All in all, the only sensible way of allocating registers is to perform it *component-based*. This is done on both ends of the allocation algorithm:

- The subjects of colourings are the *components*. That is, not a single `float2` gets allocated, but each of the components gets allocated separately.
- The logical components get assigned *two* colours. One colour stands for the actual register, the second one for the component. To ensure a correct colouring, two interfering components must differ in at least one colour.

Consider a colouring of Program V.24. We assume hereby that all values are of type `float2`. Figure V.13 shows such a colouring. Here, the two way colouring is be represented by a square, in which subsquares (representing the components) are filled with specific colours (representing the registers). In this case, we can make do with *one* colour, that is, with one single register: Because of the componentwise allocation, the register can hold two `float2` values at the same time.

Figure V.13 A component colouring for Program V.24



Program V.26 shows in symbolic form how that program might be translated into GPU code. (*Note:* For ease of exposition, this program uses the swizzle and mask syntax of CGIS, not that of GPUs[16]. To the reader of Chapter IV, it is familiar, and to the uninitiated, it is more accessible than the GPU notation.) We observe how a single register is sufficient to hold two small values.

```
IN R.xy, [a];
DIV R.zw, R.xy,2;
ADD R.xy, R.xy,R.zw;
MUL R.yz, R.xy,R.zw;
OUT [d], R.yz;
```

Program V.26: Register allocation on GPUs for Program V.24

[16]For the curious, the real GPU code would look like this: `TEX R.xy, ...; MUL R.zw, R.xxxy, 0.5; ADD R.xy, R.xyxx, R.zwxx; MUL R.xz, R.xxyx, R.xzwx; MOV outcolor.yz, R;`. Note the fill components on the right sides to align the swizzled components with the masked components on the left site.

Superword Level Parallelism

So far, we have seen that it is possible to hold multiple symregs in a single actreg. But the power of the two-colour-approach ranges further. Consider Program V.27.a, which performs two multiplications. The values are considered to be of type `float`.

```
x_scaled ← x*3;
y_scaled ← y*3;
```

(a) Two `float` multiplications

```
MUL C1.x, C2.x,3;
MUL C3.x, C4.x,3;
```

(b) Two multiplications on `float` components

```
MUL R1.xy, R2.xy,3;
```

(c) One multiplication on `float2` components

Program V.27: Superword level parallelism

Program V.27.b shows a schematic translation into GPU code. The `Ci` are some components arising from colouring. But this translation is wasteful: We employ two four-component multiplications, while with masking rejecting all but two `float` multiplications. But both multiplications can be performed in a single instruction, as shown in Program V.27.c. This assumes that `x` and `y` are allocated to the `xy`-components of register `R2`, and `x_scaled` and `y_scaled` are allocated to the `xy`-components of register `R1`. In other words, it supposes that in the colouring, for each of the two pairs, both elements share the same register colour and have differing component colours.

This can be enforced in the register allocation by *glue edges*. If `cgisc` were to encounter the situation of Program V.27.a, two separate actions would take place: Modifying the instructions to work in parallel on the different values, and ensuring that this is actually possible by guiding the register allocation.

First, two *wrapper registers* would be created (Program V.12). Wrapper registers work as a container for symregs. A `CirWrapperReg` is a subclass of a `CirSymReg`, and therefore can be used as an operand in GPU operations. In this case, a wrapper register comprising the symregs `x` and `y` would be created, and another one for the target symregs. These would then be an operand and the target for a new multiplication instructions. Symbolically, it could be written as `MUL {x_scaled,y_scaled}, {x,y},3;`.

Second, *glue edges* are drawn between the symregs in a single wrapper register. The register colouring phase then makes sure that symregs connected by a glue edge get the same register colour. In this way, glue edges lead to *Superword Level Parallelism* [LA00].

As it happens, the same mechanism used for this superword level parallelism is also used to ensure that the components of a single symreg get allocated into a single actreg: All components of a single vectorial scalar also receive glue edges among each other. Strictly speaking, it is not necessary to use a single physical register for all components: If a programmer defines a variable of type `float2` and never uses together the `x` and the `y` component as an operand, then these components do not need to reside in the same register. However, if these components would never get used together anyway, the programmer would quite certainly not have used a `float2` to hold the data. In a two-component struct, which would be the right tool for that job, the components are allocated separately. Thus, the compiler silently assumes that by using a vectorial type the programmer intends to use it as such, and inserts appropriate glue edges automatically.

This is also the reason for the limited use of *synthesised* superword level parallelism in practice, as alluded to in Section V.4.b: The programmer expresses the program already in appropriate SIMD form, thus the possible SIMD parallelism is obvious.

Colouring

When starting the colouring phase, glue edges hold together symreg components for which superword level parallelism can be exploited, and interference edges denote symreg components which have to be coloured with different colours. It is necessary, however, to also start with a precolouring for input data. This is because reordering (swizzling) is not possible in texture fetches: When a `float` stream has been decided to reside in the `z`-component of a texture, then the texture fetch can save the value only in the `z`-component of a register. Therefore, input data start the colouring phase with one of the two colours, the component colour, already fixed.

V.6 Remaining System Parts

The preceding sections focused on the compilation of CGIS code into GPU code and the execution of the generated code. This section sums up the other parts of the system. This entails mentioning the other parts of the compiler proper, and a discussion of the other software provided in the system.

The complete package of the system is built using an autoconf [G07a] based system under Unix and solution files for Microsoft Visual Studio 2003 under Windows. Provided that the compilation system is set up correctly, the user can compile the complete system including tests with the tip of a finger. Third party utilities such as YACC or the `OORS` compiler are needed when the programmer changes the relevant source files. The CUDA runtime needs the CUDA toolkit and SDK.

V.6.a Internal Components

General Remarks

The complete CGIS system offers a multitude of debugging and inspection facilities. To aid during development, additional debugging information about almost everything done during the compilation can be output. Information about the current state of the program (for example, before and after some source transformations) can also be output in GDL [A07a] form, aiding in debugging. Figure V.14 shows an extract of such a GDL graph.

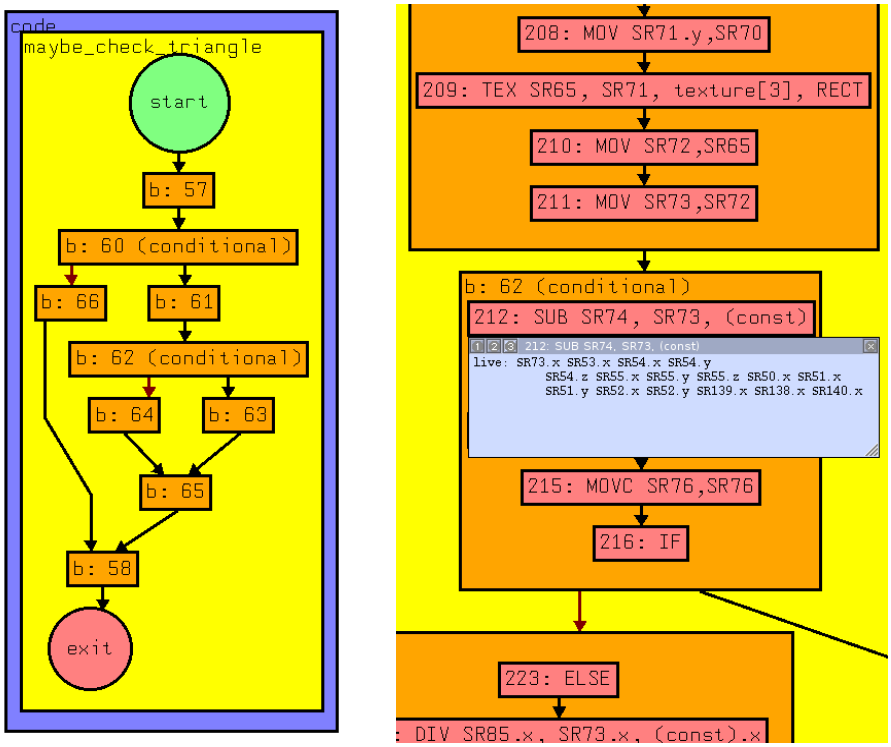
CUDA

A small CUDA backend exists solely for testing purposes. This backend does not currently support full CGIS: Most importantly, reductions are not supported. The main use of the CUDA backend is to arrive at CUDA code from the same source code by essentially the same compiler. By comparing the performance differences one can get an idea of were the problems in the restriction to the OPENGL model lie. Section VI.2.a uses the CUDA backend for a performance comparison.

The code generation for CUDA is radically different to that of GPU code. CUDA employs its own parser and code generator, that is, the abstraction level of `cgisc`'s output lies more on par with C. In this sense, the CUDA backend is comparable to the CG output of BROOK.

This abstraction means that the compiler is freed from many considerations. For example, register allocation is a non-issue, because it is handled in the later phases by the

Figure V.14 Extract from a GDL graph



(a) Control flow overview (b) Detailed view of annotated instructions

CUDA compiler. Also, whereas the GPU backend has to implement computations which are not directly supported in the GPU instruction set by sequences of native instructions on registers, the CUDA backend can simply output a call to a library function. A library header file included with the CGiSCUDA runtime library provides CUDA implementations for these functions, e. g., rotation operators and vectorial operations such as cross and dot products.

All in all, the CUDA target is a completely different issue than the standard OpenGL GPU target. That the output code lies on a much higher level of abstraction means that the code generation phase is much shorter; and the high-level CUDA API makes sure that also the runtime support has much fewer issues to face than the OpenGL runtime. Many experiences in the GPU backend of `cgisc` cannot (or: need not) be productively used in the CUDA backend.

SIMD CPUs

Besides the GPU runtimes, CGiS also targets SIMD CPUs. Both the SSE family of extensions for Intel and AMD CPUs [I06a] and the AltiVec extensions for PowerPC CPUs [F06, FLW07] can be targeted.

For a large part of its operations, the SIMD compiler can use the compiler code presented in this chapter. However, optimising output for SIMD CPUs entails a different set of optimisations and transformations. For example, obviously the SIMD compiler does not have to care about the packing of data into textures. It also outputs C-intrinsics for the operations, avoiding the need for register allocation. On the other hand, the input code

needs to undergo transformations to keep the SIMD units occupied in a data-parallel fashion. In particular, a transformation called *kernel-flattening* translates the SPMD parallelism presented by parallel execution of kernels (that is, a `foralll` loop) into SIMD parallelism on vectors. This means a transformation of *task* parallelism (multiple elements are operated upon in parallel), as it is expressed in CGIS and used in the GPU compiler, into *data* parallelism. The runtime system also has different tasks to perform. It does not have to cope with a graphics pipeline model of the target, but has the opportunity to schedule iterations and to reorder data as such to maximise the usage of caches in streaming memory accesses and gathers.

A backend for the CELL processor [IST05] is currently under development. This will introduce task parallelism on top of the AltiVec data parallelism. It is also planned to use the program analysis tool PAG [AMW95] for various of the data flow analyses which have been implemented by hand so far. The PAG-analyses can then replace the hand-written analyses, as they will be more powerful and general. For example, PAG is able to carry data information across procedure calls, in contrast to the hand-written analyses; in general, they are probably easier writable than hand-written analyses, in particular in presence of guarding, masking and swizzling. Ultimately, PAG shall replace the hand-written code, for it allows a robust framework for general program analyses. The PAG compatibility is also the main reason for the internal representation of the CGIS code in the way of sequences of `CirOperation` (Section V.2.c, Figure V.7).

V.6.b Other Parts of the System

A few auxiliary libraries are part of the CGIS system. These entail a debugging output library and the handling of command line parameters. A more important component is `libbmp`, a library designed to manipulate images. This library is used extensively in many tests (Section VI.2) to transform data in a specific way or to provide default CPU implementations against which the CGIS code can be measured.

To test CGIS in larger contexts, several complete sample implementations have been written in CGIS. For example, the wave propagation example of Section IV.5 belongs to these. These program are designed to show CGIS' performance or to demonstrate other points in CGIS. Chapter VI is solely concerned with these implementations.

V.6.c Acknowledgements

The autoconf build system was created and is maintained by Gernot Gebhard, who also is responsible for some parts of the auxiliary libraries. He also wrote an early version of the `OORS` pattern matcher grammar by translating my hand-written code generator into the `OORS` syntax and an early version of the parsing code. Nicolas Fritz is responsible for the SIMD part of CGIS. This entails the code transformation and generation phases used in creating SIMD code, and the runtime library. Fritz also is creating the CELL backend, the connection to PAG and PAG analyses. The GPU runtime library uses the open source library GLEE [W06] written by Ben Woodhouse to load the OPENGL extensions, that is, to provide the prototypes for the extension functions and the symbolic constants and to initialise the function pointers.

V.7 Summary and Outlook

This chapter has presented the CGiS software package. This entailed the compiler, the runtime library and supportive components. We have seen how the compiler parses, analyses and transforms the CGiS code, how it generates GPU code, and how it further analyses and transforms this code. We have also seen the runtime support and seen what happens behind the scenes of the API functions. Finally, we have taken a quick tour of the other components of the CGiS package. Let us now take a brief look at some of the goals CGiS strives to fulfil (Section IV.1.a).

Compatibility is achieved by basing the runtime system on cross-platform standards. OpenGL is available for all relevant platforms, that is, for all platforms supporting the hardware itself. Compatibility to legacy hardware has been achieved by program transformations to cater for control flow restrictions and by introducing the concepts of texture packing and copy kernels. These features also make up a part of *abstraction*.

Adaptability is a purely internal feature. It is not visible to the outside and of no concern to the user. Internally, adaptability has been achieved foremost by the pattern matching code generator. The adaptability has already been tested during the development of `cgisc`, because the G80 generation of hardware introduced several new capabilities. Of particular importance were integer types and operations. Because new instructions have been introduced and not all retained instructions have been orthogonally extended to integer types, the code generation process had to be modified quite extensively. As far as possible, though, the changes could be *localised*, and the inheritance and selective overriding features of `OORS` were useful in confining the necessary changes in the code generation phase to small portions.[17] Other adaptability features concern the quantitative characteristics of GPUs which are abstracted in profile characteristics.

The SIMD code generation, which is out of the scope of this work, can also make use of the CGiS infrastructure of the compiler. It completely diverges at the pattern matching time. This is as far as commonalities can be abstracted. Future developments will provide an additional opportunity to test the claim of adaptability.

This concludes the description of the system. We have seen a description of the language and a description of how it is implemented. What remains now is to see all of these parts in action: Descriptions of whole programs using CGiS to implement data-parallel algorithms and evaluations of their performances. This is the topic of the following chapter.

[17] Other parts of the software had to undergo more drastic revisions, as should be expected upon introduction of a new basic type.

VI

Applications

If such claims sound too good to be true,
keep in mind that they were made by TEX's designer,
on a day when TEX happened to be working,
so the statements may be biased; but read on anyway.
D. E. KNUTH, *The TEXbook*, 1984

In the last Chapters, we have seen some of the design goals of CGIS investigated. This chapter presents a number of applications written in CGIS. The goals of this chapter are manifold; in general they can be summed up as *"establishing that programming* CGIS *is worthwhile"*, particularly amounting to *efficiency*. Therewith, this chapter concludes the detailed discussion of the CGIS system.

Section VI.1 starts with a preliminary discussion about the efficiency of GPU programs: What do we mean by this, and how can it be measured? Section VI.2 forms the heart of this chapter, describing in detail various applications and their implementations. Their performances are evaluated according to various metrics, and we shall investigate the results for other information. Section VI.3 recapitulates the performance results and puts them into a larger context, arguing for why CGIS achieves its goals. Section VI.4 sums up and concludes this chapter.

VI.1 Basic Concepts

When evaluating the efficiency of the CGIS system, we have to make certain decisions on the overall scope and manner of evaluation. On an absolute level, it has to be decided *what* to measure. Absolute measurements are meaningless without comparison to existing references; thus, we have to establish *against which* competitive implementations to contend. Also, the battlefield has to be fixed, that is, *where* the measurements take place. In this setting, this amounts to a choice of hardware, operating system and hardware driver and their versions. This section explains the various options in these three dimensions and the choices taken in performing the evaluations later on.

For efficiency, the main factor to measure is the execution time. But *which* execution time has to be measured? First, let us fix some nomenclature. The various *"short-ids"* are used in tables and figures.

- The time to initialise the OPENGL system is called *initialisation time*, with the short id *init*.
- The time to upload data and to download data is called *transfer time*, with the short id *data*.
- The time to execute the program, including all internal data copies, is called *execution time*, with the short id *exe*.
- The sum of transfer time and execution time is called *run time*, with the short id *run*.
- The sum of run time and initialisation time is called *overall time*, with the short id *overall*.

Note that there are some problems associated with the measurements of these API functions. OPENGL functions are in general *asynchronous*, meaning that they may return control flow to the caller before all the work has been done. The GPU might still be working on the commands, or the commands may even still be stored in a command reordering buffer in the driver. Thus, simply measuring the execution times of OPENGL functions is not advisable. For example, in some cases I have experienced that the execution of a program does not actually take place until a read operation tries to download its results. This means that with a naïve measurement of the function calls, the `execute_`*`NAME`* function would account only for some command buffer changes inside the driver, whereas the `get_data_`*`NAME`* function does all the work.

Thus, for a precise measurement, the driver has to be forced to execute all commands in its buffer before a timing measurement is taken;[1] but this negates all the optimisations which would be in place had the driver retained the opportunity to reorder or cluster operations in the way that it was actually designed for. Thus, it has to be kept in mind that adding up initialisation, transfer and execution times might lead to a different result than measuring the overall time when OPENGL is left to its own. But in the tests, the differences between measuring run time as one function or divided into its three components were not significant; in particular, the interpretation of the results does not change due to the detailed timing measurements.

These timings already provide some information taken by themselves. Execution times for various sizes of the data load tell how well the GPU *scales* to increasing load. The ratio of transfer time to run time gives insight on where the *bottleneck* in the implementation lies. The ratio of run time to overall time also gives information about the *scaling* of the implementation.

Which of these timings to chose for comparison with other implementation depends on the second choice of the evaluation: against what to compare these absolute measures to arrive at a meaningful relative metric. The most obvious choice for comparison is a CPU implementation, because the main usage of CGIS is to outsource computations from

[1] OPENGL offer two functions: `glFlush` orders the command buffer to be *submitted* to the GPU, `glFinish` additionally returns only after all commands have *completed* execution. Thus, `glFinish` is what has to be called to ensure that all computations had taken place.

the CPU. The CPU implementation can be either a hand-written, cache-conscious, fine-tuned version, possibly using SIMD-intrinsics and inline assembly; or standard C++-code, compiled by a modern, optimising compiler. For the tests, the latter alternative was chosen, because it is the standard usage model of C++.

Having fixed that, run time and, to a lesser extent, overall time are the important metrics to discuss in this context. Run time measures the costs of one-shot outsourcing, without amortisation over a sequence of programs. Overall time adds the constant initialisation costs, which are relevant only if the application really runs only a single program once. Execution time is useful to compare differences in scalability; as we shall see, cache effects on the CPU can prevent it from scaling as well as a GPU.

As a secondary choice, some examples have been implemented in CUDA. CUDA offers a state-of-the-art exploitation of GPUs, and in this way we can see how much we are losing by having to go through the OPENGL programming metaphor.

The third dimension is formed by the systems on which the measurements take place. On the one hand, having a number of different GPUs opens up the possibility to compare between them, and maybe argue about what algorithms can benefit from what GPUs. On the other hand, the GPUs should be rather advanced in order to enable arguments about the state-of-the-art. In the end, this amounts to a comparison within NVIDIA's G80 family. (This also is a prerequisite for CUDA.) The following three GPUs have been elected to partake in the comparisons.

GPU	proc.	speed[2]	memory	bandwidth
GeForce 8600 GT	32	1.18 GHz	256 MB	22.4 MB/s
GeForce 8800 GTS	96	1.20 GHz	640 MB	64.0 MB/s
GeForce 8800 GTX	128	1.35 GHz	768 MB	86.4 MB/s

For short, these GPUs are called 8600, (8800) GTS and (8800) GTX, respectively.

All three GPUs were tested in different base systems. The text will argue about the differences this is likely to make. The software was largely fixed: All systems were running under a recent Linux system with the newest driver version (100.14.11), and all programs have been compiled with `-O2` by GCC. Having different versions of these software programs would be a potential major source of superficial differences between different systems; in particular, different driver versions can make more of a difference than the hardware itself.[3]

The actual systems were the following:

- The 8600 resides in a system with Intel Core 2 Duo E6700 CPU, 2.66 GHz, FSB 266 MHz, with 2 GB DDR2 RAM, Kernel 2.6.21.5 (Ubuntu 7.10) with GCC 4.1.3.
- The 8800 GTS resides in a system with AMD Athlon 64 X2 CPU, 2.4 GHz, FSB 200 MHz, with 2 GB DDR RAM, Kernel 2.6.18.8 (openSUSE 10.2) with GCC 4.1.2.
- The 8800 GTX resides in a system with Intel Core 2 Duo E6300 CPU, 1.86 GHz, FSB 266 MHz, with 2 GB DDR2 RAM, Kernel 2.6.21.1 (Ubuntu 7.04) with GCC 4.0.3.

[2] The given speed is that of the processing units.
[3] The driver issue will be treated in more detail in Section VI.2.f.

To show that CGIS also works on older hardware and to compare the performances of different generations of GPUs, some tests also have been made on an NV40-class GPU:

GPU	pipelines	speed	memory	bandwidth
GeForce 6800 GT	16	0.35 GHz	256 MB	32.0 MB/s

This GPU was tested in the AMD system which also harboured the GTS. Some tests have been prohibitively slow on that GPU; there, no measurements have been included.

Concerning CUDA, the tests could not be run under the same systems as the other tests. The driver available under Linux sometimes showed irregular patterns in the times for certain functions. Thus, the tests have been performed on a later driver version, which unfortunately is available only under Windows. (A driver bug also prevented the tests of RC5 under Linux (Section VI.2.f).)

VI.2 Sample Applications

To test the CGIS language and the CGIS system, a variety of real-world tasks have been implemented on GPUs. This section explains some of them to point out strengths and weaknesses of CGIS' handling of the GPU, and in particular the chosen abstraction. To this end, the algorithms are quite varied in nature. Although all inherently parallel (to facilitate an implementation in the CGIS model), they differ on points such as density of computation, memory access pattern and extra-GPU control flow. In all cases, the CPU implementations have been written in a normal, yet performance-conscious way. This means that they have been implemented in standard, sequential C code, but sometimes not exactly implementing the CGIS algorithm, if that would have been unreasonable for a CPU implementation. For example, the CPU implementation of *life* works on char data instead of floating point data. The sections on the various applications make clear any changes on the algorithm provoked by this.

So as not to disrupt the flow of the text, a number of performance graphics have been deferred to the end of this chapter.

VI.2.a Mandelbrot

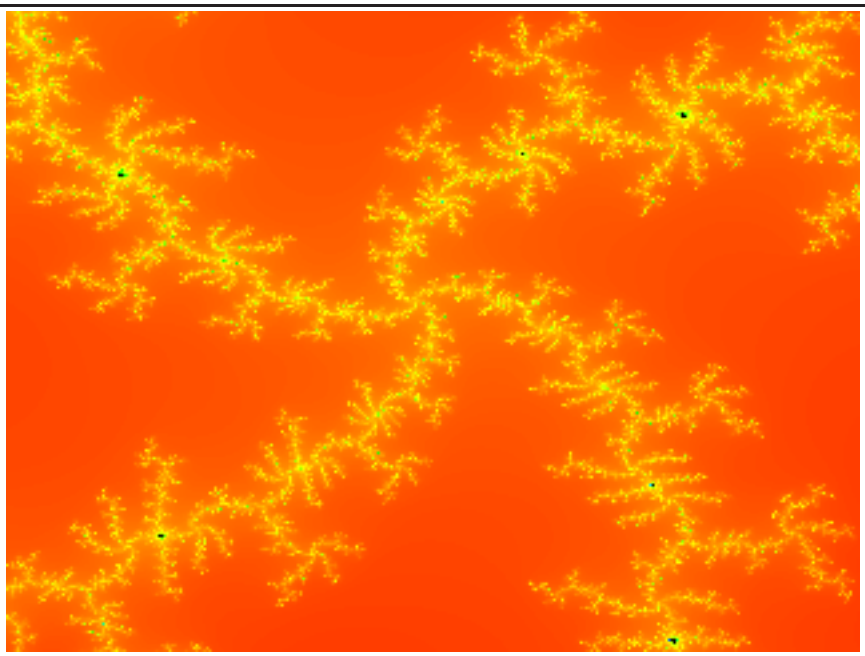
The fascinating *Mandelbrot Set* [W05] has mesmerised countless spectators by its sheer beauty, yet it is defined in a very simple and abstract way.

Algorithm and Implementation

Let $c \in \mathbb{C}$. The sequence $(z_n^c)_{n\in\mathbb{N}_0}$ is defined as $z_0^c = c$ and $\forall n \in \mathbb{N}.z_n^c = (z_{n-1}^c)^2 + c$. The *Mandelbrot Set* is defined as $M := \mathbb{C}\backslash\{c \in \mathbb{C} : \lim_{n\to\infty} |z_n^c| = \infty\}$. For computing the familiar picture for some $Z \subset \mathbb{C}$, the recurrence $z_n^c = (z_{n-1}^c)^2 + z$ is computed for each $z \in Z$ until either

- $|z_n^c| \geqslant 2$ for some n, or
- a prespecified maximal iteration count is reached.

Figure VI.1 A Mandelbrot computation



This is implemented in a program we shall refer to simply as *Mandelbrot*. Colours are assigned to the points depending on the outcome of this process.

Figure VI.1 shows an example output. For this particular picture, the plane was set to $Z = \{(x, y) : x \in [0.27525, 0.28371], y \in [-0.6101, -0.6015]\}$, and the threshold iteration count is 500. This is also the setting used for one performance test.

Program VI.1 shows the CGIS implementation of the complete iteration. For each point, the recurrence is computed until one of the break conditions is fulfilled. The maximal iteration count is provided as a uniform parameter. Because it is settable by the user at execution time, hints cannot be used. The computed value is stored in a variable which is later used to compute the image.

Note that there is a potentially huge number of computations in a very tight loop per element. Memory transfer is minimal: The initial values z_0^c are transfered to the GPU, each element computes solely on these z_0^c and writes back the iteration count or a sentinel value, and that value is read back. As far as the control flow is concerned, because of the fractal nature of the Mandelbrot set, the number of iterations needed in neighbouring elements may be equal in some regions and extremely diverse in other regions.

The OPENGL driver reports the following numbers of native instructions for Mandelbrot: 24 arithmetical and 2 texture instructions.

To sum it up, the example *Mandelbrot* possesses the following characteristics.

- high arithmetical density per element
- large amount of GPU work between CPU–GPU data transfers
- pure streaming memory accesses

```
PROGRAM mandelbrot;

INTERFACE
extern in float RE<SIZEX,SIZEY>;
extern in float IM<SIZEX,SIZEY>;
extern out float OUT<SIZEX,SIZEY>;
extern in float max_;

CODE
procedure mandelbrot(out float point, in float re, in float im,
                     in float max_){
  point = -1;
  float tempx=re, tempy=im;

  int round=0;

  while(round<max_){
    float temp = tempx;
    tempx = tempx*tempx - tempy*tempy + re;
    tempy = 2*temp*tempy+im;

    if(tempx*tempx + tempy*tempy >= 4){
      point = round+1; round = max_;
    } else round = round+1;
  }
}

CONTROL
forall(point in OUT, re in RE, im in IM)
  mandelbrot(point, re,im, max_);
```

Program VI.1: Mandelbrot

Performance

Tables VI.1 and VI.2 show the performance of the algorithm on varying resolutions (data sizes) for the three systems. Figures VI.11 and VI.12 show the same data graphically. The first test case consists of the aforementioned subset of the plane visualised in Figure VI.1. The second test case is a deeper computation, involving up to 5000 iterations in the plane $Z = \{(x,y) : x \in [0.324505, 0.32451], y \in [-0.048551, -0.048555]\}$.

As this is the first example application, the figures are discussed in detail; subsequent discussions can then concentrate on the peculiarities. Consider first Figure VI.12.a. It presents the raw performance data for the three G80-class GPUs. The X-Axis denotes increasingly larger data size, doubling at each step. The Y-Axis denotes the overall time for the GPU in milliseconds. For each size, we see three bars, each one corresponding to one GPU. Each bar is divided into a lower part which denotes the largely invariant initialisation time, and an upper part which denotes the run time. It can be seen easily that the performance of the three kinds of GPUs neatly follows the order of their hardware capabilities, with the 8800 GTS leading over the relatively small 8600 by a large margin, and the high-end 8800 GTX outpacing the GTS. Also, we see that in this case, the initialisation time is a significant factor for the smallest data sizes, but gets dwarfed

Table VI.1 Performance measurements of Mandelbrot (first set)

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
320×240	37	30	3	7	140	24	4	7	53	28	2	7
453×339	68	26	4	9	283	24	7	8	101	28	4	7
640×480	135	27	6	15	582	24	14	10	204	28	9	9
905×607	273	30	13	24	1177	24	28	13	404	29	18	11
1280×960	552	29	25	38	2297	25	56	20	810	30	35	16

Table VI.2 Performance measurements of Mandelbrot (second set)

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
320×240	246	27	3	31	391	24	4	15	364	28	3	12
453×339	494	27	4	56	768	24	7	22	716	28	4	17
640×480	990	28	6	105	1519	24	14	37	1437	28	9	28
905×607	1977	29	13	198	3088	24	28	66	2865	29	18	47
1280×960	3950	29	25	371	6127	25	56	122	5731	29	35	83

pretty soon when the data size is approaching larger dimensions.

Subfigure VI.12.b presents the execution time relative to the time for the smallest data size. The logarithmic scale means that an ideal scaling would lead to a straight 45° line. In this particular cases, the workload in the higher dimensions is not exactly a multiple of the workloads in lower dimension: Because of the very nature of fractals, the fraction of pixels for which the highest iteration count has to be reached can vary strongly with the chosen points, and these depend on the resolution. Other tests will show more reasonable results.

Subfigure VI.12.c compares the run time and overall time of the GPU with the execution time of the CPU in the same system. This is devised to demonstrate the usefulness of outsourcing the computation to the GPU. We observe several key points. First, even the lowly 8600 outperforms the CPU by a factor of 5–10. This is even though that system happens to combine both the most powerful CPU in our tests and the least powerful GPU. The relative performance gap increases to nearly 50 for the run time on the largest data set on the 8800 GTX. Second, the benefit increases with increasing data size. As a third and last point, we see that including the initialisation time in the comparison makes for a substantial difference only for the 8800s, because there the run time takes up a smaller part of the overall time.

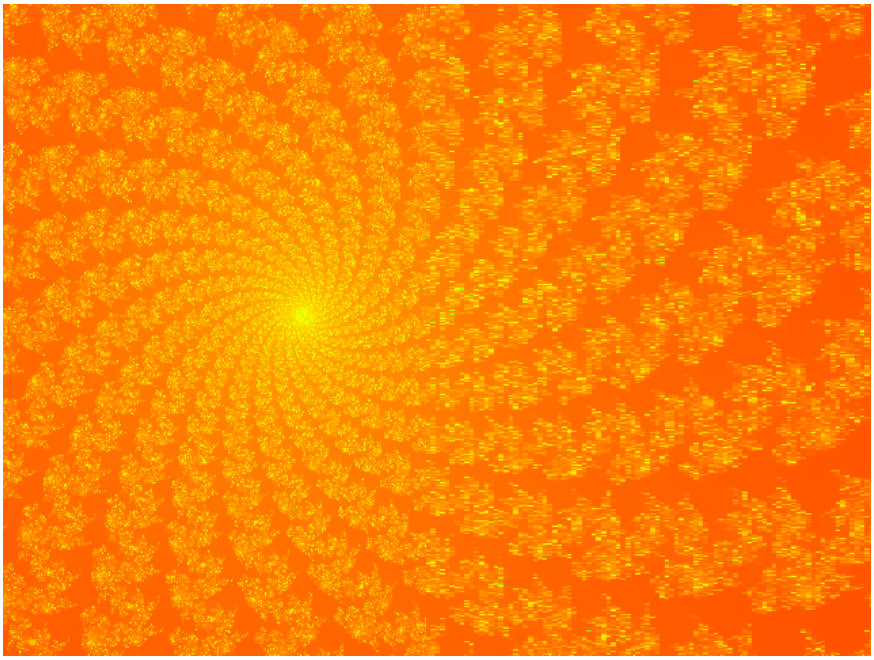
Figure VI.11 and Table VI.1 tell a similar story for the other test set. Here, the actual execution time is much smaller than for the second test set, and therefore the transfer time takes up a larger part of the run time. This is the reason for the relatively poor performance of the GTS, which suffers from a generally slow system; this will be investigated in more detail in Section VI.2.c.

Other Considerations

Truth to be told, the Mandelbrot set exhibits one problem of contemporary GPUs, namely that of precision. The x86 floating-point architecture internally works on a higher precision than the 32-Bit of single-precision floats. Thus, for some pictures which represent a very narrow subset of $\mathbb{C}$, the CPU produces a sharper result: Intermediate values which fall together in single-precision float can be distinguished with the higher bits of the

internal registers. The effect of this can be observed in the second test set[4]. That example uses a very narrow subset of the plane and 5000 iterations. Figure VI.2 consists of two halves merged together: The left half is computed on the CPU, the right half on the GPU. In the right half, more artifacts are visible than in the left half. If GCC is forced to single-precision arithmetic by the option `-ffloat-store`, the higher-precision register value are moved into memory after each intermediate step. This also results in a slightly less crisp picture, which means that the precision is probably to blame for the difference in the computation.

Figure VI.2 Precision issues in Mandelbrot computation: CPU (left), GPU (right)



The Mandelbrot computation also lends itself to implementation on the older NV40 architecture. The program runs unmodified on the architecture, it just has to be compiled with the appropriate profile. Table VI.3 shows the relevant entries for the first test set on that architecture. We see a main difference in the execution time: The 6800 takes significantly longer to execute the program than all G80 GPUs (as expected). Still, it is *faster* than the CPU implementations, even though the GPU is already a few years outdated.

Table VI.3 Performance measurements of Mandelbrot on the 6800

size	init	data	exe
320×240	39	4	52
453×339	39	7	72
640×480	39	12	114
905×607	39	26	185
1280×960	41	48	312

[4]This is the reason for why there *are* two test sets for Mandelbrot.

Another interesting performance comparison is with the CUDA system. Tests have been performed with the CUDA code generated by `cgisc`. Consider Table VI.4. As mentioned before, the tests had to be run under the Windows system, which had quite large initialisation times. That notwithstanding, we can compare the runtimes of the two test sets.

Table VI.4 Performance measurements of Mandelbrot with CUDA

size	init	run	CUDA
320×240	160	12	16
453×339	168	12	47
640×480	156	31	15
905×607	164	40	125
1280×960	164	71	46

First test set

size	init	run	CUDA
320×240	164	27	31
453×339	168	27	328
640×480	156	63	47
905×607	160	98	1188
1280×960	156	188	140

Second test set

Only the 8800 GTS on Windows has been tested.

As expected, the CUDA version performs better than the GPU version. But let us investigate the issue further. The two drops in performance of the CUDA version in the second and fourth size are a result of the division of elements into blocks and threads (Section III.4.b). For the other sizes, a neat division is possible (e. g., 640×480 in 20×30 blocks with 32×16 threads); thus, a block can be filled with the maximal number of 512 threads, or something close to it. For odd sizes such as 453, the implementation issues only 1 thread per block. The performance results clearly show the effect of that: Performance with CUDA is only possible with a carefully selected mapping of threads into blocks.

If we disregard these mapping artifacts as not relevant to the performance comparison, we can see by comparison of the two test cases and by investigating the increasing sizes, that the performance advantage of the CUDA version is comparatively higher for lighter work loads (first case) than for heavy loads (second case); this suggests that the issue is not in the generated code, but more in the fact that the handling of the GPU is hindered by the OPENGL programming model.

VI.2.b Life

Martin Conway's *Game of Life* [G70] (*Life*, for short; see also Section IV.3.c)) is, semantically speaking, a simulation of birth and death of a population, as provoked by a sustainable density of individual entities in available space and adequate populational support for reproduction. Syntactically speaking, it is a two-dimensional cellular automaton with two states per cell and a state transition function defined by the states of a cell itself and of its eight neighbours.

Algorithm and Implementation

The rules of the game are as follows. At the beginning, a $n \times m$ array of cells is initialised with living cells (state $s^0_{i,j} = 1$) and dead cells (state $s^0_{i,j} = 0$). Given the states of the cells at time step σ, the states at step $\sigma + 1$ are defined as

$$\forall i \in \{1, \ldots, n\}. \forall j \in \{1, \ldots, m\}. s^{\sigma+1}_{i,j} = \begin{cases} 1 & s^\sigma_{i,j} = 1 \wedge \xi^\sigma_{i,j} \in \{2, 3\} \\ 1 & s^\sigma_{i,j} = 0 \wedge \xi^\sigma_{i,j} = 3 \\ 0 & \text{otherwise} \end{cases}$$

where $\xi^{\sigma}_{i,j}$, the neighbourhood of point (i, j) at time step σ, is defined as

$$\xi^{\sigma}_{i,j} = \sum_{\substack{i' \in \{-1,0,1\} \\ j' \in \{-1,0,1\} \\ i' \neq 0 \vee j' \neq 0}} s^{\sigma}_{i+i',j+j'};$$

here we let $s^{\sigma}_{i+i',j+j'} = 0$ for indices out of range.

In other words, a living cell stays alive if it has two or three neighbours, otherwise it dies; a dead cell comes alive it it has three neighbours, otherwise it stays dead; and the surroundings of the game board are assumed to be a dead zone.

```
PROGRAM life;

INTERFACE
#HINT(no_texture_reuse)

extern inout float field<SIZE,SIZE> : packing (1) G;
extern in float size_minus_one;

CODE
procedure iterate(inout float element, in float2 index,
                  in float size_minus_one){
  gather element: tl<-1,-1>, t<0,-1>, tr<1,-1>, l<-1,0>, r<1,0>,
                  dl<-1,1>, d<0,1>, dr<1,1>;
  float sum=0;
  if(index.y>0){
    if(index.x>0) sum+=tl;
    sum+=+t;
    if(index.x<size_minus_one) sum+=+tr;
  }
  if(index.x>0) sum+=l;
  if(index.x<size_minus_one) sum+=r;
  if(index.y<size_minus_one){
    if(index.x>0) sum+=dl;
    sum+=d;
    if(index.x<size_minus_one) sum+=dr;
  }

  if(sum>3 or (sum+element<3)) element = 0;
  else element = 1;
}

CONTROL
forall(e in field) iterate(e,indexXY(e),size_minus_one);
show(field);
```

Program VI.2: Game of Life

A complete CGIS implementation of a single time step is shown in Program VI.2. Cells are modelled as `float` elements with value 0 or 1. Observe the *explicit boundary checks*: The general rule that lookups are clamped onto the range of the stream is not useful here, because lookups into the outside have to be regarded as returning a 0. Thus, the GPU

implementation does not get the boundary check for free, that is, the CPU is not at an advantage here.[5] Also note that the procedure `iterate` uses the convenient `inout` flow of the state variable: This means that the new states have to be written into another field, and that the *contents* of this field are then copied into the real state field. This would be absolutely unreasonable to do on the CPU: Thus, the CPU implementation performs a simple pointer switch between input and output fields. Also, representing a living or dead cell with a `float` variable would be unnatural on CPUs. Therefore, the algorithm is implemented also with `int` and `char` datatypes; highest CPU performance is achieved with `char`, and thus this type is used for the performance comparisons. As a final note, the complete simulation is implemented as a tight iteration over `execute_life()` without any other functions: The initial state is uploaded once, then thousands of iterations may follow, and then the final state is downloaded.

For the performance tests, a start position is subjected to 500 iterations.

The OPENGL driver reports the following numbers of native instructions for Life: 56 arithmetical and 9 texture instructions.

To sum it up, the example *Life* possesses the following characteristics.

- very light arithmetical density per element (few computations per memory accesses per element)
- huge amount of GPU work between CPU–GPU data transfers
- data copying (`inout` buffers require mirror-copy algorithm)
- additional boundary checks on the GPU
- larger element size than on the CPU (`float` vs. `char`)

Performance

Table VI.5 Performance measurements of Life

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
256×256	281	57	3	190	570	43	9	91	559	53	4	76
362×362	573	61	8	353	957	47	15	152	1147	58	11	115
512×512	1362	57	15	680	2386	47	74	271	2377	57	26	195
724×724	2717	67	35	1339	4654	45	80	514	4944	62	49	363
1024×1024	5640	79	77	2585	34825	53	302	994	23449	67	164	682

Table VI.5 and Figure VI.13 show the relevant performance data. Figure VI.13.b shows that the computation scales very well to larger sizes: We see a slightly super-linear scaling, which will become the pattern for all tests. This is likely to be due to the GPUs being able to hold a large number of computations in flight when they are waiting for the memory (Section II.4.a): With more computations waiting to be scheduled, more latencies can be hidden, whereas CPUs can only have a very limited number of instructions waiting for data in flight.

Figure VI.13.c presents us with a mixed effect. On the one hand, the 8600 system performs relatively weak, gaining only a moderate speed-up of about 100%, whereas the

[5] Of course, the boundary check is only *conceptually* free on the GPU. That is, in actuality the GPU is here at a disadvantage, because it uses both the clamping specified by the OPENGL setting and then the conditionals to disregard some values.

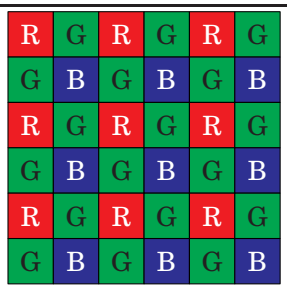
8800s outperform their CPUs by a factor of 5–10 on suitable sizes. This is less due to the relatively weak 8600 GPU, but more to the by far superior CPU in that system. Also, that system does not show the curious effect for the largest data size: The relative advantage of the 8800s to their CPUs skyrockets from around 10 to the high 20s. This is likely due to cache effects on the CPUs: 2 arrays of $1024 \cdot 1024$ chars[6] fit into the 2 MB L2-Cache of the 8600 system, but get evicted in each iteration from a 1 MB cache. That effect is pronounced here because the algorithm iterates over the whole array *multiple times*.

All in all, here we see another algorithm which can beneficially be implemented in CGIS on GPUs. This time, the high memory bandwidth is of particular importance. And to top it of, the GPU implementation visualises the simulation, too.

VI.2.c Demosaic

Most digital cameras do not record full colour information per pixel. Instead, the sensor array receives the light filtered by a *Bayer Pattern*, so that each sensor records only one colour component. Figure VI.3 displays that pattern. Because human eyes are more sensitive to variations in green light than in red or blue light, half of the sensors record green information and only a quarter each record red or blue information.

Figure VI.3 Bayer RGB-pattern



For example, Figure VI.4 shows a part of a bird's eye area with green feathers (the left parrot in Figure VI.6). The sensor values are mapped to grey scale, with white representing high intensity at that point and for that colour. Observe how in the whitish eyeball all colours are set with relatively high and uniform intensity; in the green feathers, the green pixels are set to high intensity, the blue pixels to moderate intensity, and there is almost no red component.

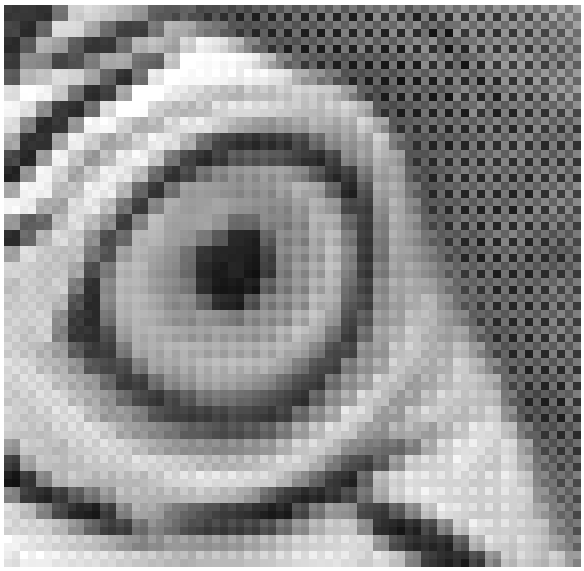
Algorithm and Implementation

There exist various algorithms to reconstruct the complete image information from this pattern. Because of the mosaic-like nature of the Bayer pattern, this phase is called *demosaicing*. The algorithm implemented in our example *Demosaic* is that of [MHC04]; it is visualised in Figure VI.5.

Consider a blue sensor and its neighbourhood (displayed at the top). To compute the complete RGB information for the corresponding pixel, the algorithm gathers information from the neighbourhood in a certain way (displayed at the bottom). Obviously, there is no processing needed for the blue component, which has been recorded faithfully. To reconstruct the red and green information, the algorithm takes into account two kinds of information:

[6] Recall that the CPU implementation uses `char` instead of `float` as the base type.

Figure VI.4 Bayer patterned input to produce Figure VI.6 (detail)



(1) the bilinear interpolation of the desired colour in the nearest neighbours recording that colour, and

(2) the colour gradient of the pixel's own colour with respect to its nearest neighbours.

The results of the bilinear interpolation and the gradient are then weighted with certain factors which have been experimentally established as producing a high image quality [MHC04].

The CGIS program implementing that algorithm is shown in Program VI.3.

The final result of that transformation is a fully coloured image, for example, Figure VI.6.

The OPENGL driver reports the following numbers of native instructions for Demosaic: 73 arithmetical and 14 texture instructions.

To sum it up, the example *Demosaic* possesses the following characteristics.

- moderate number of arithmetical operations per element
- large number of gathers per element
- overall small amount of computation for the whole algorithm

Performance

Demosaic follows a natural streaming paradigm and should therefore be efficiently executable on GPUs. But is it worth it, considering the other costs of outsourcing the computation to the GPU?

The performance figures in Table VI.6 and Figure VI.14 tell us that it is *not* worth it. Consider Figure VI.14.c. We see that only in the GTX system for a large data set the run time approaches that of the CPU, but it does so from *above*: In no test case is the

Figure VI.5 Reconstructing an image from a Bayer pattern

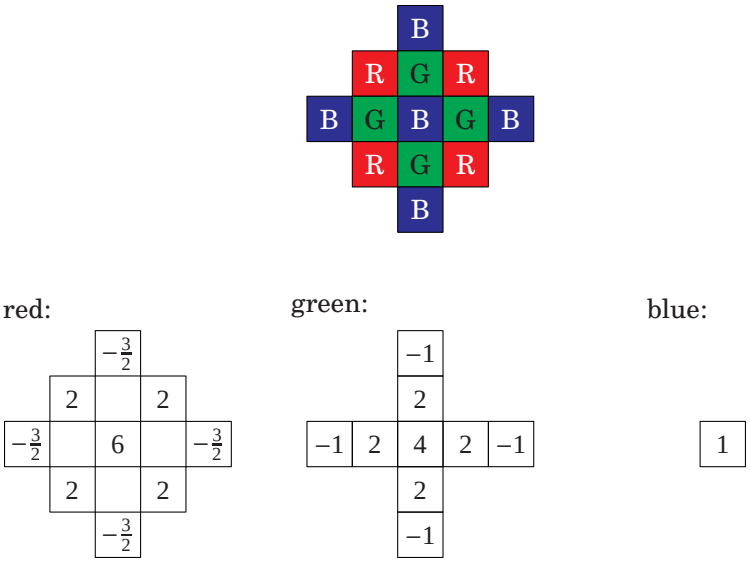


Table VI.6 Performance measurements of Demosaic

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
384×256	4	27	5	9	6	25	9	12	6	31	6	12
384×512	8	30	9	10	13	24	18	14	12	31	13	14
768×512	16	30	17	12	26	25	35	16	24	32	24	16
1536×512	32	27	34	16	50	25	71	22	48	32	49	19
1536×1024	64	29	69	26	99	25	140	35	97	33	96	28

use of the GPU beneficial. Also, Figure VI.14.a paints a curious picture of the relative performances: the 8600 system suddenly is the fastest one.

Figure VI.15 tells us the reason for this effect. Figure VI.15.a shows the run time of Demosaic broken down into execution time and transfer time. We see that the bulk of the run time is taken for data transfers. Figure VI.15.b strengthens that point, plotting the ratio of transfer time in run time. Figure VI.15.c gives the absolute transfer times logarithmically scaled. We see that run time is completely dominated by the transfer times, and that therefore the bottleneck lies in the CPU part of the system: The CPU is not able to feed the data fast enough the exploit the GPU's power; a power that can be deduced from the execution times in Table VI.6.

We also see in Table VI.14 that the execution times itself hardly change with increasing data size: a low base cost is increased to at most less than thrice for 16-times the data size.

All in all, Demosaic is a prime example for the fact that even such a naturally parallel algorithm cannot currently be efficiently implemented on the GPU as a co-processor. The raw execution times show that the GPU *is* faster than the CPU, but the overhead in the co-processor use case is too high to make the GPU profitably usable. Demosaic becomes worthwhile if more computations are performed on a particular image instead of just the demosaicing, compensating the overhead.

```
PROGRAM demosaic;

INTERFACE
extern in float mosaic<SIZEX,SIZEY>;
extern out float3 picture<SIZEX,SIZEY>;

CODE
procedure demosaic(in float plain, out float3 colour,
                   in float posx, in float posy){
  ... // gather neighbourhood as per Figure VI.5
  float3 retval;
  int ixi = posx, iyi = posy;
  bool nred_row = (iyi&1)>0, blue_col = (ixi&1)>0;

  if(not (nred_row xor blue_col)){ // red or blue position
    float gradient = plain - (rr+uu+dd+ll)*0.25;
    float primary = (r+u+d+l)*0.25 + 0.5*gradient;        // alpha
    float secondary = (ul+ur+dr+dl)*0.25 + 0.75*gradient; // gamma

    retval.g=primary;
    if(nred_row) { retval.b=plain; retval.r=secondary; } // blue
    else { retval.r=plain; retval.b=secondary; }         // red
  } else { // similar code for the two kinds of green sensors
  }
  colour = 0 max (1 min retval);
}

CONTROL
forall(pixel in picture, plain in mosaic)
  demosaic(plain,pixel,indexX(plain),indexY(plain));
```

Program VI.3: Demosaic

Other Considerations

Demosaic also works on the NV40 architecture, with a slight modification. The G80 version (Program VI.3) uses a bitmask on an integer to check whether an index is odd or even. On the NV40, this has to be replaced with a floating point modulo operation.

Table VI.7 shows the performance of Demosaic on the 6800. Similarly to the Mandelbrot case, we see an expected decrease of performance, but not as high as for Mandelbrot. Again, the execution times increases moderately from a larger base. For so few operations, the smaller power of the 6800 does not carry much weight.

VI.2.d Wave

Wave propagation is a classical algorithm for a physical simulation, where the main reason for implementation lies in the visualisation. Section IV.5 already introduced and explained the program *Wave*, so what remains here is to explain the performance characteristics.

Figure VI.6 Result of demosaicing a Bayer patterned image

Table VI.7 Performance measurements of Demosaic on the 6800

size	init	data	exe
384×256	40	7	83
384×512	40	15	87
768×512	40	28	93
1536×512	40	57	104
1536×1024	41	112	127

Implementation

It can be seen that the procedure `propagate` uses several memory lookups for only a small amount of computation, whereas the `refraction` procedure and thus its callers have a relatively large amount of computation per memory access. The final `render` procedure introduces an arbitrary lookup. Also note that, just like for the CPU in the Life example (Section VI.2.b), here the GPU version uses pointer switching instead of data copying to advance from one time step to the next. So the directing loop on the CPU works by calling `execute_wave` followed by `get_texture_wave` and `set_texture_wave` calls to switch the textures.

The tests have been performed with 75 iterations.

The OPENGL driver reports the following numbers of native instructions for Wave:

- For `propagate`: 42 arithmetical and 14 texture instructions.
- For the two `refraction` procedures: 63 arithmetical and 3 texture instructions each.

- For `render`: 6 arithmetical and 4 texture instructions.

To sum it up, the example *Wave* possesses the following characteristics.

- very light to medium arithmetical density per element, depending on the kernel
- huge amount of GPU work between CPU–GPU data transfers
- texture switching instead of data copying
- additional boundary checks on the GPU
- arbitrary `lookup` operations and some `gather` operations

Performance

Table VI.8 Performance measurements of Wave

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
256×256	2305	27	4	143	2617	24	12	131	3361	31	7	112
256×512	4490	31	11	260	5848	24	25	242	6590	32	16	193
512×512	8434	27	21	493	12178	25	48	448	13018	32	32	352
512×1024	16242	28	44	954	23859	25	94	880	27812	32	61	674
1024×1024	35371	28	89	1888	45919	25	188	1703	56970	33	119	1334

Table VI.8 and Figure VI.16 hold the performance data for Wave. We see again that the 8800s have advantages over the 8600, and that the GPU implementations scale very well. The advantage over the CPUs, as depictured in Figure VI.16.c, is glaring. Again, we see the power of GPUs increase through repetition of computation.

VI.2.e Skeleton

Skeletonisation is the process of deducing the underlying core structure from a bitmap image by stripping away unneccesary information. For example, consider a line with a width of various pixels. When we are caring only for the basic information *"a line from here to there"* (e. g., in character recognition), then the width of the line may be disregarded. So to speak, it is stripped from its outside pixels, so that only a *skeleton* remains.

Figure VI.7 shows an examples of this. A hand-drawn picture[7] in (a) is transformed by a skeleton algorithm into the picture in (b). The basic structure is retained in the skeleton, and the jiggles and overhangs become more pronounced.

Algorithm and Implementation

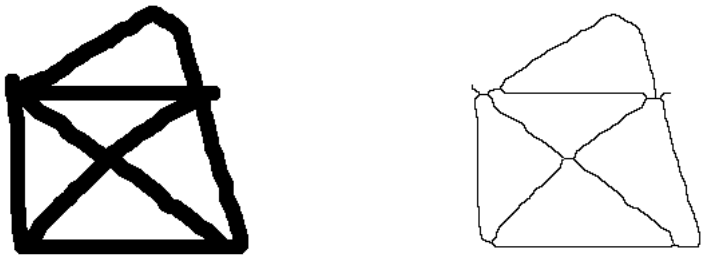
[BFRR01] cites several algorithms for skeletonisation. The test *Skeleton* implements the algorithm from [SR71], which works in several, iterated phases.

(1) Find out which pixels form the contour of the image and remove them, by performing each of the following substeps in parallel:

   (a) check for pixels lying on the bottom contour,

[7] It is intentionally not geometrically precise in order to give rise to a more interesting output image.

Figure VI.7 A drawing and its skeleton



(a) Input drawing (b) Output skeleton

(b) check for pixels lying on the top contour,

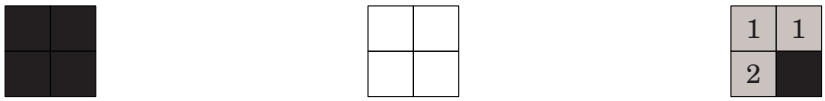
(c) check for pixels lying on the left contour,

(d) check for pixels lying on the right contour.

(2) Have any pixels been removed in step (1)? If so, repeat it, otherwise end the algorithm.

It is not possible to just check for the contour of the drawing in step (1), because contour pixels may form a skeleton of an image after all. Consider Figure VI.8.a. All pixels lie on the outside of the drawing, but removing all of them yields the result of Figure VI.8.b: The drawing has vanished completely. Removing the outer pixels in separate phases, as hinted in Figure VI.8.c, retains a single pixel.

Figure VI.8 Removing contour pixels in parallel or in sequential phases



(a) Input drawing (b) Result after removing all contour pixels (c) Result after removing contour pixels one-by-one

The heart of the algorithm lies in the detection of when a pixel forms part of the contour of the image. This is established by pattern recognition of the pixel's 3×3-neighbourhood. The algorithm does not directly search for a set of patterns in the neighbourhood, but instead classifies the patterns and the neighbourhood according to some conditions. The complete process is too complicated to be explained here; it suffices to say that it employs large boolean expressions and numerous conditionals. The interested reader is referred to the explanation in [BFRR01] for more details.

The picture has to be updated after each substep. On the CPU, this can be done either during the iteration (using a separate array and pointer switching) or afterwards (with a memcpy from an accumulator array). Tests have shown that the deferred memcpy is more performant, so that is used in the tests.

The second phase of the algorithm, the termination check, is a classical reduction operation. Phase (1) records for each pixel whether it has been changed (set to white), and Phase (2) performs an $\vee$-reduction over this information. On the CPU, this reduction is done on the fly by updating a single boolean variable, so step (2) consists simply of the check of a single variable.

In the tests, the images have been created in such a way that the algorithm takes the same number of iterations (11) to converge in each case. Therefore, the workload truly scales with image size.

The OPENGL driver reports the following numbers of native instructions for Skeleton:

- For initialisation of the output array: 2 arithmetical and 1 texture instruction.
- For each of the four subphases: 77 arithmetical and 11 texture instructions.
- For the reduction computation: 15 arithmetical and 2 texture instructions.

To sum it up, the example *Skeleton* possesses the following characteristics.

- reduction on the GPU, none on the CPU
- large amount of computation for memory access
- boolean computations with shortcut evaluation on CPU, floating point computation on GPU
- explicit memory management on the CPU, implicit management on the GPU

Performance

Table VI.9 Performance measurements of Skeleton

	8600				8800 GTS				8800 GTX			
size	CPU	init	data	exe	CPU	init	data	exe	CPU	init	data	exe
320×256	71	30	7	70	110	26	17	100	118	33	11	76
320×512	160	28	18	123	226	25	39	166	252	33	23	123
640×512	322	32	34	227	579	26	112	298	510	33	53	211
640×1024	666	28	75	433	1162	26	278	564	1073	34	115	385
1280×1024	1435	29	152	851	2489	27	641	1073	2421	36	256	734

Table VI.9 and Figure VI.17 show the performance results for Skeleton. A couple of things stand out. Just like in Demosaic, the GTS shows the worst performance. And just like in Demosaic, this is because of increased data transfer times on that architecture (Figures VI.15 and VI.18). These times have also improved with a later driver version (which is, again, only available under Windows right now), so that these huge times should not be seen as normal. In any case, despite the high amount of data transfer, the implementation is still worthwhile for larger image sizes.

Other Considerations

In the particular case of this reduction, CUDA offers another possibility to dramatically speed-up the process. In the reduction, the only information that matters is whether *any* pixel had been changed. A complete reduction of the stream is overkill. Therefore, a hand written CUDA program works in another way: A kernel is run in parallel on all

elements, and all threads write into *one*, shared stream variable. Then, the CPU need to download only that single variable and check it.

Obviously, that process loses some information. Whereas in the full reduction case, we know exactly *how many* pixels have been changed, that is not known in the CUDA implementation. Yet again, the information we get is sufficient for the purposes of the Skeleton algorithm.

Table VI.10 Performance measurements of Skeleton with CUDA

size	init	run	CUDA
320×256	169	96	47
320×512	185	174	65
640×512	169	333	106
640×1024	174	690	172
1280×1024	172	1510	359

Only the 8800 GTS on Windows has been tested.

Table VI.10 shows the results for this, different, algorithm. Clearly, the CUDA implementation outpaces the GPU implementation. This shows that when carefully tailoring an algorithm to the CUDA programming model, one can achieve yet higher performance than what is achievable in *any* kind of GPU programming.

VI.2.f RC5

RC5 [R94] is a widely used block encryption algorithm. A sequence of bytes is divided into 64-Bit sized blocks which are then encrypted independent of each other. That independency is what makes RC5 suitable for implementation in CGiS. In contrast to such *block ciphers*, *stream ciphers* encrypt a stream of data in small blocks sequentially: The encryption of a byte in a string depends on the string prefix. Thus, stream ciphers are not suitable for implementation in CGiS, despite the name.

Because RC5 depends on bit-operations, it can be implemented only on G80 class GPUs. Unfortunately, tests have shown that the latest Linux driver does not correctly implement the read-back from integer textures into the host memory. A newer driver version fixes this problem, but at the time of writing, that version is only available on Windows. Therefore, tests were performed only on the Windows system with Microsoft's Visual C++ on the GTS.

Algorithm and Implementation

A block cypher encrypts the blocks in an input stream independent of each other; therefore, it suffices to explain the algorithm on a single block. We divide the block into two 32-Bit values `A` and `B`. That block is then encrypted with a uniform key over a number of rounds; here, we chose 31 rounds. The key then consists of 64 values of 32 Bit, held in the array `S`. The following pseudo-code describes the algorithm [BR96]:

```
A = A + S[0];
B = B + S[1];
for (i = 1 ; i <= 31 ; i++) {
    A = A ^ B;  A = ROTL(A, B) + S[2*i];
    B = B ^ A;  B = ROTL(B, A) + S[(2*i)+1];
}
```

Program VI.4 presents the CGIS implementation. Observe how rotation is expressed with a natural operator symbol. Although rotation has to be simulated with a sequence of shifts and masks on GPUs, some CPUs [F06] support native rotation. On these targets, the compiler can easily exploit the available instructions instead of having to reconstruct the programmer's intents from low-level C-like code (*idiom recognition* [RWP05]).

```
PROGRAM RC5uint;

INTERFACE
extern in uint2 S<32>; // 31 rounds.
extern in uint4 ABs<SIZEX,SIZEY>; // The stream to be encrypted.
extern out uint4 outABs<SIZEX,SIZEY>; // The output.

CODE
procedure get_S01(in uint2 S<_>, in uint where, out uint2 s){
  lookup S: temp<where>;
  s = temp;
}

procedure encrypt(in uint4 AB, in uintb S<_>, out uintt outAB){
  uint2 S01;
  uint i = 0;
  get_S01(S,i,S01); // Get at 0, S[0] and S[1].

  uint2 AA = AB.xz+S01.x; // Encrypt two blocks at once.
  uint2 BB = AB.yw+S01.y;

  while(i<31){
    i+=1;
    get_S01(S,constant,S01);  // Get at S[2*i] and S[2*i+1].
    AA = ((AA^BB)<<<BB) + S01.x;
    BB = ((BB^AA)<<<AA) + S01.y;
  }

  outAB.xz = AA;
  outAB.yw = BB;
}

CONTROL
forall(AB in ABs, outAB in outABs) encrypt(AB,S,outAB);
```

Program VI.4: RC5

The stream is described as an `uint4` stream. To the application programmer, it is of no concern in which chunks a stream of `uint` data is processed; for the description of the algorithm in CGIS, it opens the possibility to express the inner loop as working on two blocks at once. It becomes obvious how the vector syntax of CGIS, masking and swizzling operators and automatic scalar promotion work together neatly to allow a tight description of this workings in a familiar notation.

The OPENGL driver reports the following numbers of native instructions for RC5: 44 arithmetical and 3 texture instructions.

To sum it up, the example *RC5* possesses the following characteristics:

- small loops per element
- integer computations on CPU and GPU
- no special operator support both on CPU and GPU

Performance

Table VI.11 shows the performance measurements of RC5. As mentioned before, driver issues confined the test to Windows. This is unfortunate because the initialisation time is much higher on Windows than under Linux, as are the data reorganisation times. But still, despite the larger initialisation times, the GPU beats the CPU. In particular, the actual execution time increases only slightly from its socket and really scales from the midsized test size on. This shows that the integer capabilities of the G80 GPUs have not been included in vain.

Table VI.11 Performance measurements of RC5

size	CPU	init	data	exe
2^{20}	94	164	19	16
2^{21}	203	156	62	16
2^{22}	406	163	116	31
2^{23}	804	163	226	51
2^{24}	1625	187	472	94

Only the 8800 GTS on Windows has been tested.

VI.2.g Raycaster

Ray tracing is a method to generate a two-dimensional image from a three-dimensional scene description, just as rasterisation (Section II.1.a). Whereas the scope of rasterisation is only determining visibility, raytracing is an approach which incorporates visibility, lighting, reflection and refraction in one algorithm [G02]. The subset of ray tracing only concerned with determining visibility is called *ray casting*, and a *Raycaster* is a sample application of CGiS.

The two approaches ray casting and rasterisation are radically distinct:

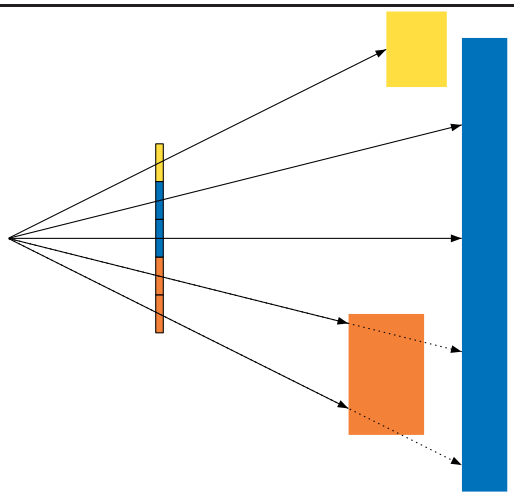
- *Rasterisation*: For each object, find where it maps to on the projection plane. For each point on the plane, discard all objects mapped to it except for the nearest one.
- *Ray casting*: For each point in the plane, find the nearest object which is mapped to it.

Algorithm

Figure VI.9 presents a conceptional overview of the ray casting paradigm. A view point, the desired view plane (shown from the side) and the desired image resolution (the subdivision of the view plane) define *eye rays*. An eye ray r corresponds to a particular pixel p on the view plane and is defined as the ray from the eye o through this point into the scene space. A given ray $r(\tau) = o + \tau \cdot s$ (with slope $s = \|p - o\|$) has intersection points $r(\tau_1), \ldots, r(\tau_n)$ (*hit points*) with objects in the scene, and the colour at p is defined to be the colour of the object intersected with minimal τ_i.

This immediately raises several questions:

Figure VI.9 Ray Casting



(1) For a given ray and a given object, how to efficiently compute the intersection?

(2) For a given ray, how to determine which objects are candidates for intersection and which do not need to be subjected to test (1)?

(3) For a given ray and an intersection at offset τ^*, how to efficiently determine whether another intersectable object might have an intersection point at offset $\tau < \tau^*$?

Question (1) is orthogonal to (2) and (3). Efficient ray-object-intersection tests have been a long-time subject of research and optimisation. One of the benefits of ray tracing with respect to rasterisation as performed by graphics hardware is that ray tracing can cope with *any* kinds of objects, whereas rasterisation hardware is limited to triangles. But obviously, efficient intersection tests are possible only for very regular structures. Thus, many application restrict the data structures to triangles, too, as we do in our tests [MT97].

Questions (2) and (3) concern the data storage format. A naïve implementation with a flat storage format of just all triangles would have to test all rays against all triangles and discard all intersections except for the nearest one. To avoid this, the space is partitioned in some way, and rays are tested against a sequence of subsets of the triangles until an intersection point has been found for one subset. More concretely speaking, let the scene space be divided into non-overlapping volumes $B_1, \ldots, B_\beta$; the way of division is of no concern at the moment, as is the shape of of the B_i or their (non-)uniformity.[8] For each B_i, we have a list $O_1, \ldots, O_{\omega_i}$ of objects which intersect with B_i, that is, those objects which reside at least in part in the volume B_i. Determining the nearest intersection point of the ray $r(\tau) = o + \tau s$ then works as follows.

(1) Find the first volume which lies on the path of r. Let this be B_i and let the interval of the ray lying in B_i be $[\tau_{\min}, \tau_{\max}]$.

(2) For all $O_1, \ldots, O_{\omega_i}$, test for intersection with the ray. Discard all intersections which lie outside of $[\tau_{\min}..\tau_{\max}]$, i. e., outside of B_i.

(3) If there are intersections in B_i, determine the nearest one, and finish.

[8] We assume that the shapes are convex, though.

(4) Otherwise, find the next volume $B_{i'}$ on the way of the ray, that is, the volume containing $o + (\tau + \varepsilon)s$. Compute $\tau_{\min}$ and $\tau_{\max}$ anew with respect to $B_{i'}$, and proceed at (2).

The key to efficient ray tracing is how to organise the B_i. Methods to do this abound [G02]; Most current ray tracers work on *hierarchical* data structures organised in some form of *trees*. Unfortunately, tree traversal is in general a recursive procedure and requires a stack for implementation [EVG04]. Other methods omit the stack altogether, by adding links between the nodes of the tree connecting adjacent volumes directly [PGSS07, HSHH07].

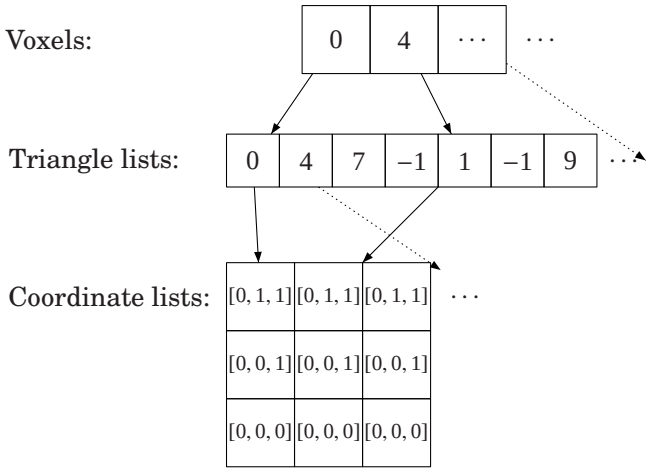
However, tree traversal and linked data structures do not really map very well to the stream-based CGIS programming model. For this reason, the CGIS algorithm uses a *uniform grid* as its data structure. The scene space is divided into a grid of uniform blocks with dimension $x \times y \times z$.[9] Then, traversing this grid with a ray is a simple operation: It entails only a three-dimensional extension of the two-dimensional Bresenham line drawing algorithm [B65] to three dimensions.

Implementation

The Raycaster is implemented as a multitude of CGIS programs, communicating which each other through shared textures.

The grid is represented as a list of *voxels*, rectangular blocks of space. Each voxel carries a list of triangle ids. These triangle id lists are concatenated into a large list, separated by −1, and the voxel stream carries an index into that list to the start of that voxel's triangle list. Figure VI.10 shows this graphically.

Figure VI.10 Triangle lists for the voxels



Each triangle is represented by several values:

- its three corners (also depicted in Figure VI.10)
- its normal
- its material (an id)

[9] x, y and z may be different.

Each material, in turn, simply corresponds to a colour, also by way of an indirection table.

A ray is represented by several values; some of these are not obvious right now but shall become clear from the description of the algorithm.

- its direction (the slope s in the ray equation)
- the current voxel (an id)
- the interval of the ray present in the current voxel ($\tau_{\min}$ and $\tau_{\max}$)
- the slope counter to check in which dimension to traverse next [B65]
- the index in the voxel's triangle list which it is currently intersecting
- the current hit point (offset and object id)
- its state (*traverse*, *intersect* or *done*)
- the final colour at the hit point

To explain the algorithm, first we concentrate on a single ray.

(1) In the beginning, the ray is in the *intersect* state and ready to intersect with the triangles of the start voxel.

(2) The ray fetches its offset into the triangle list, and then fetches the triangle id from the triangle list.

   - Assuming that the id is not -1, the ray then fetches the coordinates of the triangle. It tests for intersection with the triangle. If an intersection is found, it checks for whether it is within the current bounds of the voxel, and whether it is nearer than a previously recorded hit point. If so, the hit point is updated. In any case, the index into the triangle list is increased.
   - If the id is -1, the ray checks whether a hit has been recorded so far. If so, the state is set to *done*, otherwise to *traverse*.

(3) If the state is *traverse*, advance the ray by checking and updating the slope counter.

   - If the ray would traverse out of the scene bounds, set the state to *done*.
   - Otherwise, update $\tau_{\min}$ and $\tau_{\max}$, the slope counter, and the current voxel. Fetch the start of the voxel's triangle list and set the triangle index. Set the state to *intersect*.

(4) If the state is *done*, terminate, otherwise go to Step (2).

After a ray has been processed fully, a separate shading step takes the id of the triangle which was hit, fetches its material, fetches the material's colour, and modifies this according to ambient lighting.

This algorithm is implemented in CGIS by holding all ray's information in streams. The programs corresponding to steps (2) and (3) are run in parallel for all stream elements (that is, for all rays), but only those rays which are currently in the corresponding state partake in the computation. Step (4) is a reduction over all rays, checking for whether

all rays are in state *done*: The computation has to proceed for as long as at least one ray has not finished iterations. The shading step then is performed in parallel for all rays.

Obviously, in general scenes have more triangles than voxels, so more rays should be in state *intersect* than in state *traverse*. Therefore, the main iteration loop runs the intersection program multiple times before running the traversal program. Also, the termination checks are costly, and therefore are executed very infrequently. Still, this does not help the fact that not all rays are active during each program execution. This is a general problem with algorithms with such a divergent control flow.

The OPENGL driver reports the following numbers of native instructions for raycaster:

- The initialisation of the data is implemented with two programs with 46 arithmetical and 2 texture instructions.
- Step (2) is implemented with a program with 96 arithmetical and 11 texture instructions.
- Step (3) is implemented with two programs with 157 native and 11 texture instructions.
- Step (4) is implemented as a streaming program with 7 arithmetical and 1 texture instructions and a reduction program with 15 arithmetical and 2 texture instructions.
- The shading is implemented with a program with 31 arithmetical and 5 texture instructions.

The Raycaster can be implemented in CGIS. This shows that, in principle, even large programs with pointer-based data structures can be implemented in CGIS. However, following multiple levels of indirections through various streams is not an algorithm which really fits into the CGIS programming or implementation model. Also, the highly diverse control flow with its division into multiple programs does not cater to high-performing code. In this area, a CUDA implementation with a flat memory addressing model and suitable allocation on the blocks (Section III.4.b) offers more control than the restricted CGIS model can provide. The CUDA implementation of [PGSS07] implements the complete algorithm within a single kernel.

VI.3 Interpretation of the Results

We have seen several examples of parallel algorithms implemented in CGIS. Their performance has been evaluated in the sections devoted to the examples, but it is instructive to look at the examples as a whole. This section discusses the performance results with respect to the goal of *efficiency*.

In all cases, the execution time of the CGIS code on the G80s outperforms the CPU code. This shows the great potential of GPGPU programming, and the use of CGIS in particular. Obviously, the examples have been *chosen* in such a way that they can be implemented efficiently; but the characteristics of the examples are quite diverse, involving pure streaming code, gatherings or random lookups; low or high arithmetical density; internal iterations, diverging control flow and straight line code; external iterations or once-running examples; stream computations and reductions; explicit memory management, implicit copying, distinction between input and output streams; integer

and floating point computations; visualisation. Therefore, these algorithms may serve as representatives to show both the power of GPGPU and of CGIS. The performance difference between GPU and CPU inside the systems was different for the various systems, which is understandable, given that the poorest GPU is in the strongest host system. Nevertheless, even in these cases the performance advantage held.

When turning from execution time to run time, the world is still quite rosy. Demosaic loses its appeal due to the data transfer costs, which are not amortised across the simple execution. It suffers from being an isolated application with neither post- not preprocessing by other GPU programs nor any kind of iteration, which would amortise the outsourcing costs. Outsourcing to the GPU is worthwhile in cases where there is a large number of computations per element. In the case of Skeleton, Wave and Life, this is achieved by the iterative nature of these algorithm. Mandelbrot and RC5 employ an inner loop per element to end up with a large number of operations.

We have also seen tests of some applications on the older NV40 architecture. This was possible with only minor modifications to compensate for the NV40's lack of integer support, if any were needed at all. In some cases (Life and Wave), the GPU was too slow to be seriously considered a target for today's workloads. In Mandelbrot, however, even the NV40 could beat a modern CPU. Finally, the RC5 example showed the power of integer operations on GPUs, and thus another reason to focus on current generation hardware.

Some of the pertaining pitfalls of GPGPU have been shown, too. In the Mandelbrot application, we have seen that precision issues can lead to a noticeable degradation of output quality (Figure VI.2), and thus to usability of GPUs as a whole. This is not a CGIS issue, but inherent in the GPUs architecture. That particular issue did not come up in the other tests, though. Another issue, namely that of dependence on the driver, has also arisen in the tests. The RC5 example could not be tested on the same platform as the other examples, because the driver produced an erroneous result. During development, performance differences between different driver versions also inhibited a stable development of the compiler.

Some CUDA implementations have shown the power of more directly accessing the GPU. Performance could be gained by bypassing the OPENGL metaphor, but more importantly by implementing an algorithm for Skeleton which would just not have been possible to implement in a standard GPGPU programming model. The next chapter will investigate this as a part of the future language design issues.

We have also seen the influence of the host system on GPGPU performance. Of particular interest in this regard are the data transfer times. At times, they made the system with the less powerful GPU more attractive because the better GPUs could not use their full potential in slower host systems.

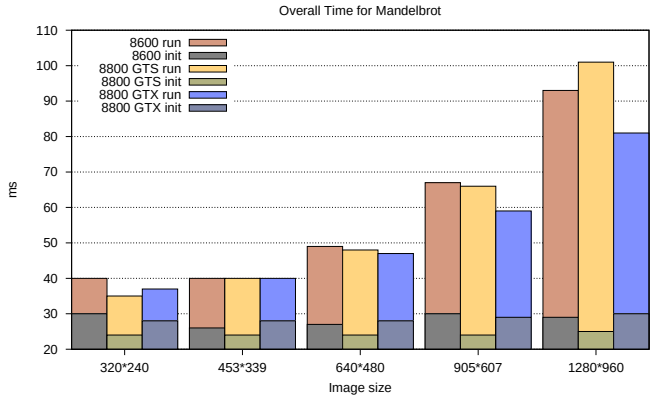
All in all, the goal of *efficiency* surely is achieved.

VI.4 Summary and Outlook

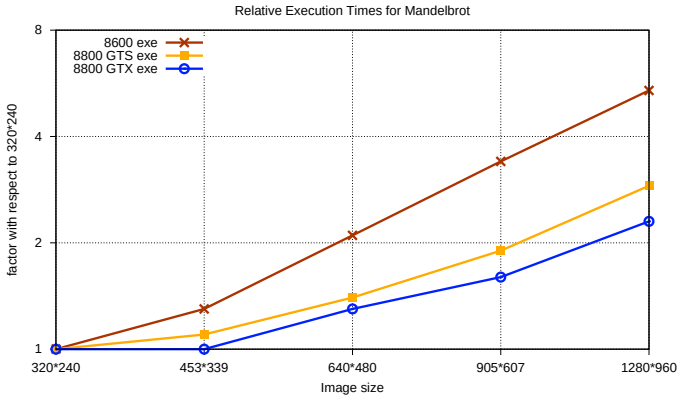
This chapter concludes the description of CGIS. The preceding chapters have explained CGIS, its heritage and its goals, its pragmatics and its implementation. In this chapter, we have finally seen CGIS in action. It has transpired that using CGIS is indeed beneficial in certain situation; thus, a main point has been achieved.

The following, final chapter builds upon all that we have seen so far from CGIS. With the performance evaluation done, we can sum up the worth of CGIS today as a language for GPGPU. We shall also investigate what other information we have gathered from these sample implementations that can guide the future development of CGIS, like languages and GPGPU.

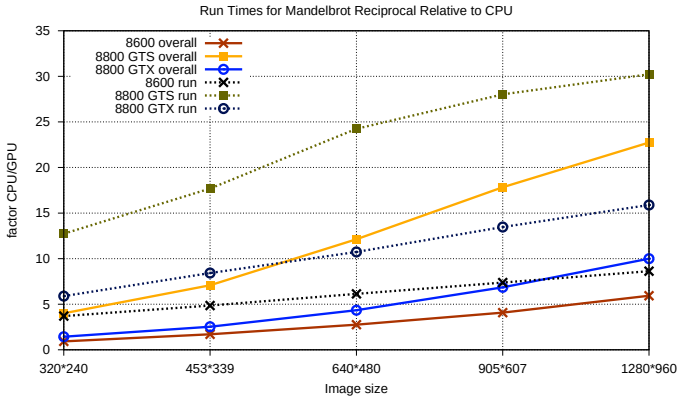
Figure VI.11 Performance results for Mandelbrot (first test set)



(a) Absolute performance of the various GPUs

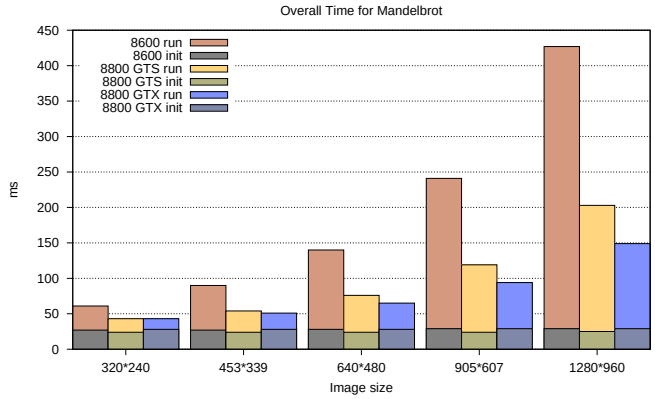


(b) Execution time scaled by data size

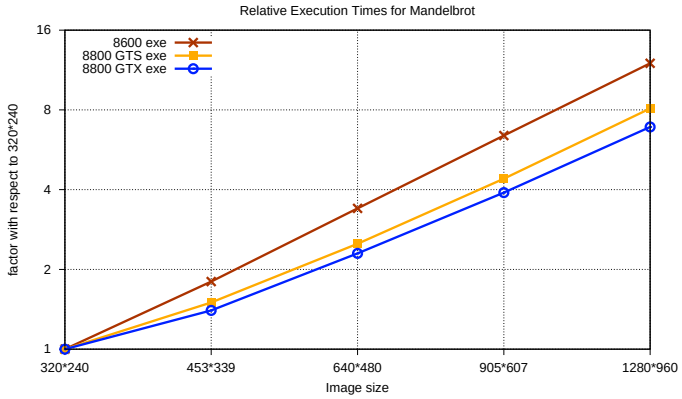


(c) Performance relative to CPU

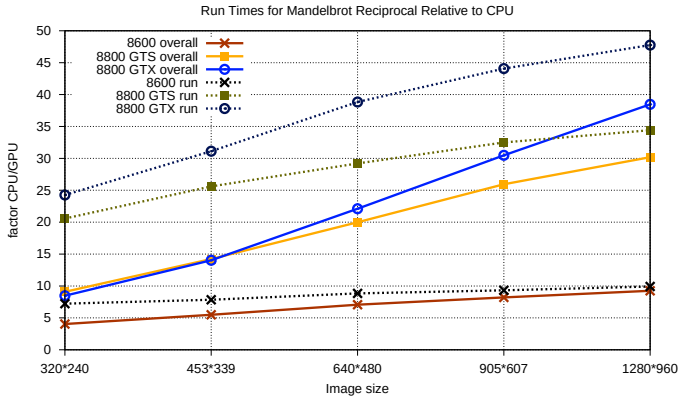
Figure VI.12 Performance results for Mandelbrot (second test set)



(a) Absolute performance of the various GPUs

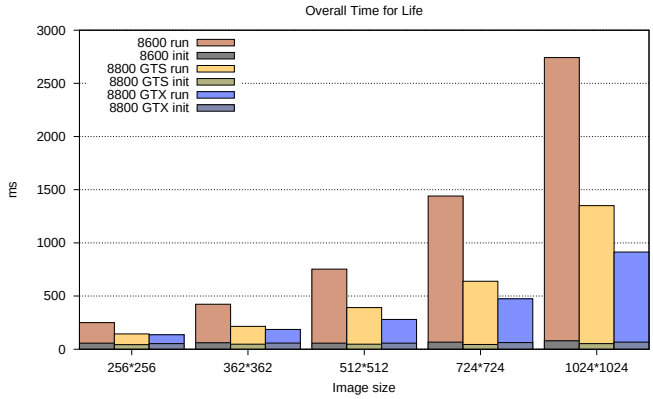


(b) Execution time scaled by data size



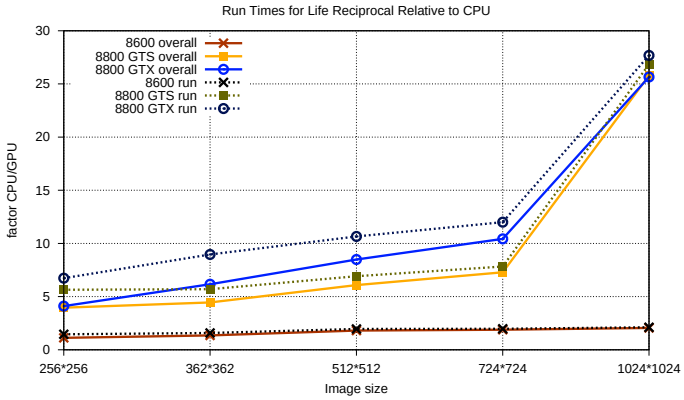
(c) Performance relative to CPU

Figure VI.13 Performance results for Life



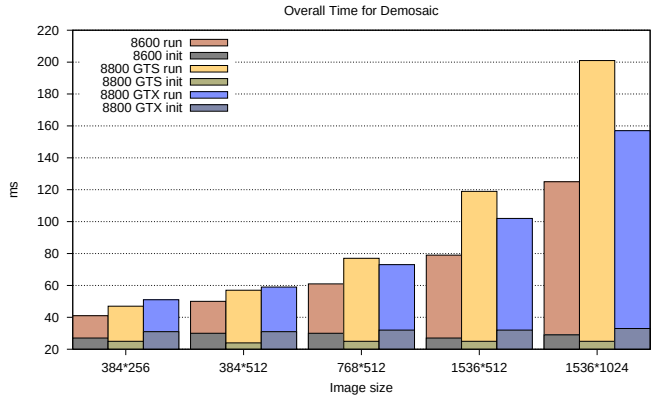
(a) Absolute performance of the various GPUs

(b) Execution time scaled by data size

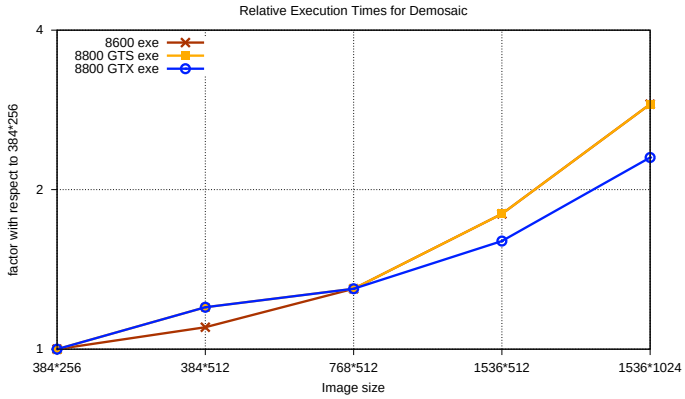


(c) Performance relative to CPU with char data

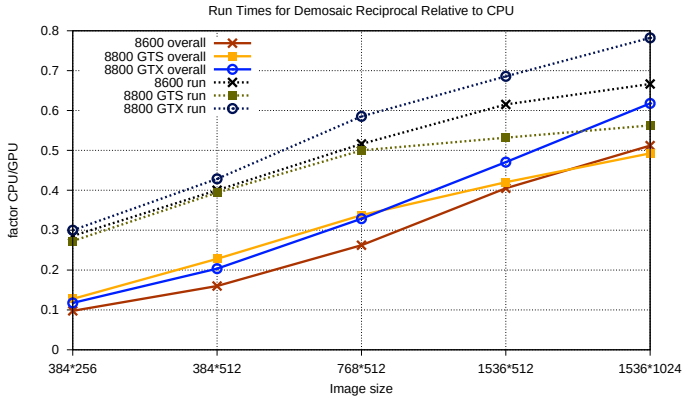
Figure VI.14 Performance results for Demosaic



(a) Absolute performance of the various GPUs

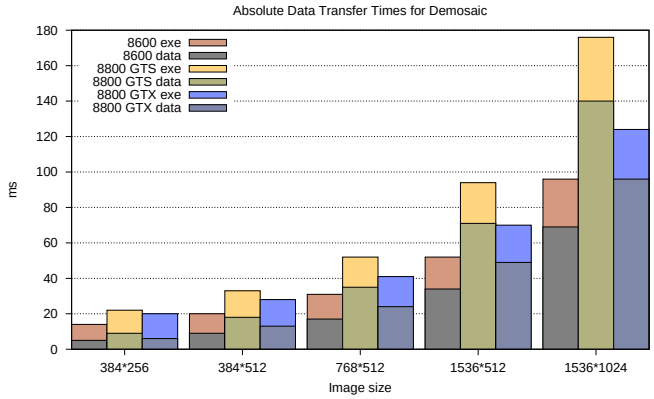


(b) Execution time scaled by data size

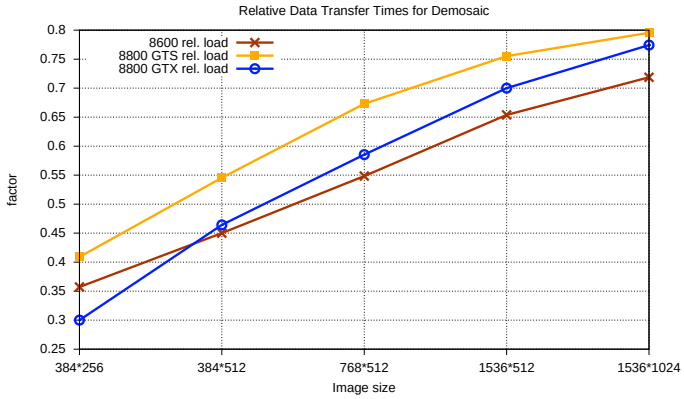


(c) Performance relative to CPU

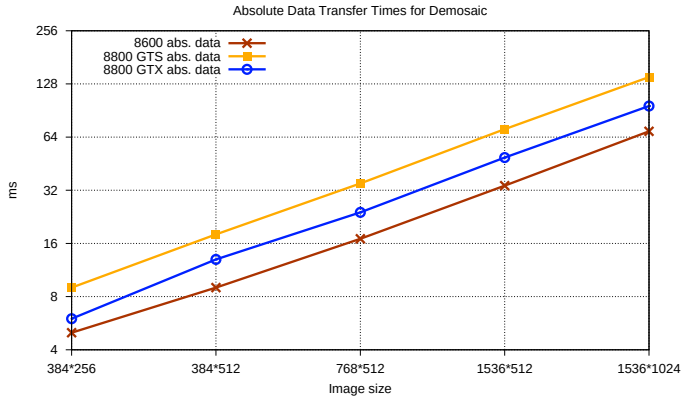
Figure VI.15 Data Transfer Times for Demosaic



(a) Execution time and transfer time of the various GPUs

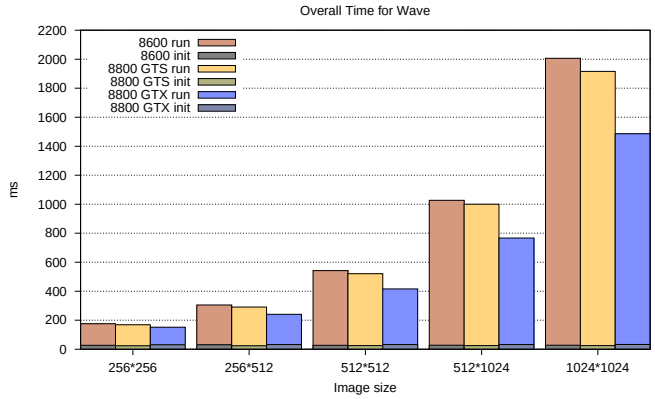


(b) Transfer times relative to run times

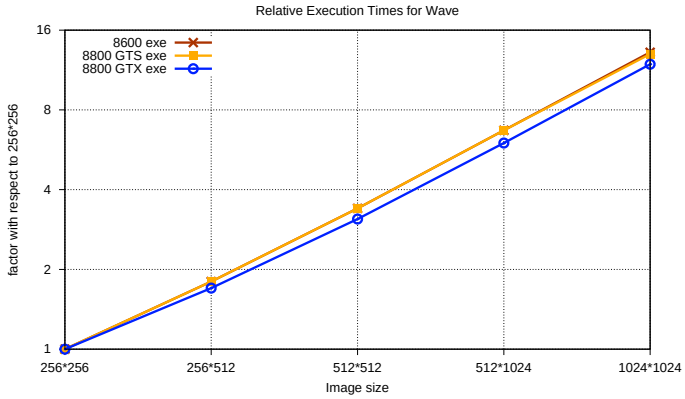


(b) Absolut transfer times

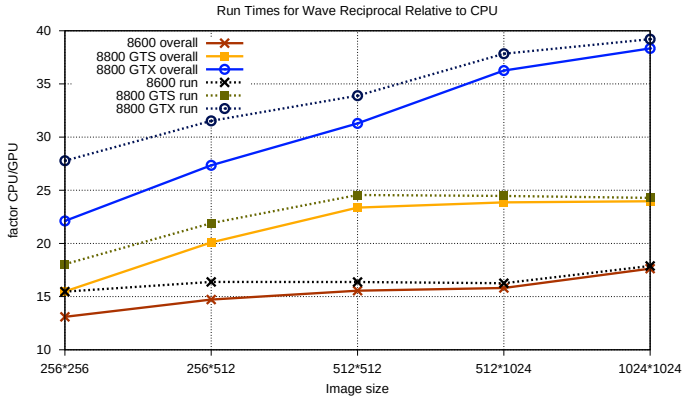
Figure VI.16 Performance results for Wave



(a) Absolute performance of the various GPUs

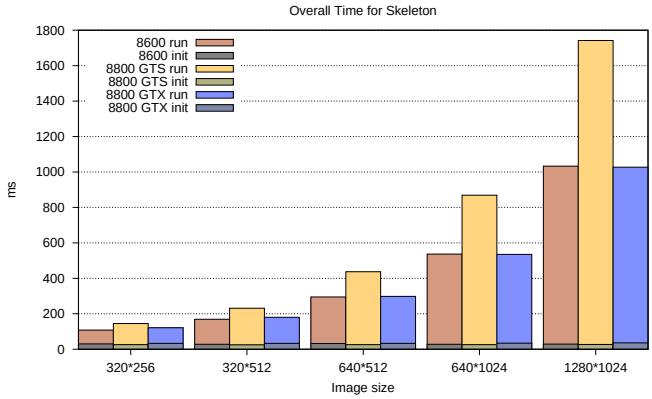


(b) Execution time scaled by data size

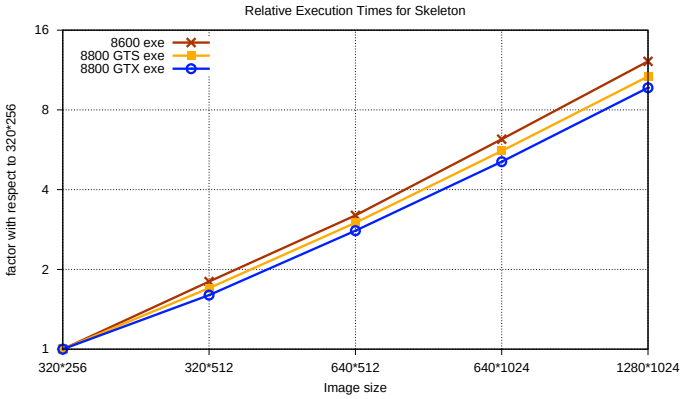


(c) Performance relative to CPU

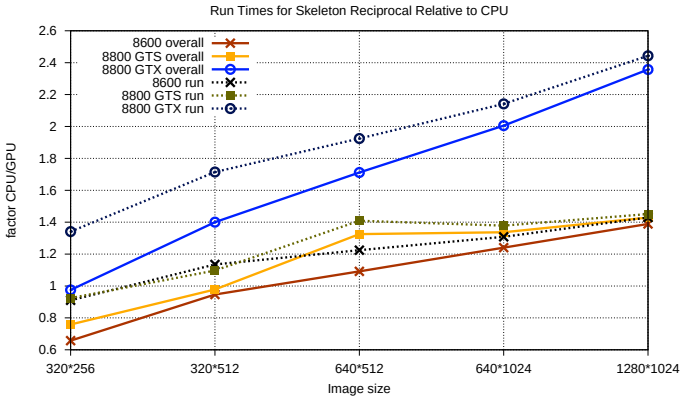
Figure VI.17 Performance results for Skeleton



(a) Absolute performance of the various GPUs

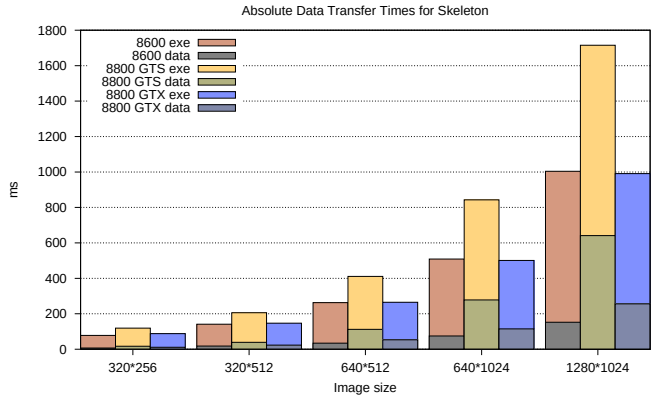


(b) Execution time scaled by data size

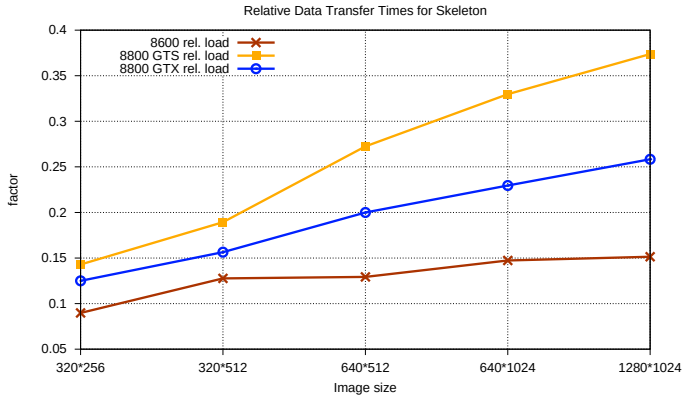


(c) Performance relative to CPU

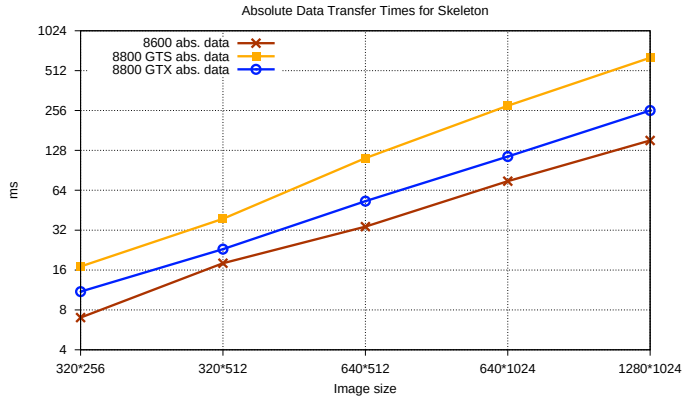
Figure VI.18 Data Transfer Times for Skeleton



(a) Execution time and transfer time of the various GPUs



(b) Transfer times relative to run times



(b) Absolut transfer times

VII

Conclusion

To conclude this part of our subject
it only remains to state summarily the general conclusions
to which our enquiries have thus far conducted us.
J. G. FRAZER, *The Golden Bough*, 1922

The final words have been said about CGIS. Now, it is time to recap the achievements and put them into a greater context.

First, I briefly recapitulate what has been presented in this work.

- Chapter II described the state of the art in GPU technology. The capabilities of current GPUs have been presented and compared with current CPUs. We have also seen the current low-level interface to GPUs and a short history of the art of GPGPU.
- Chapter III presented various approaches to abstract GPU programming. We have seen various shading languages, GPGPU languages and hardware dependent languages, and investigated them for their virtues and shortcomings.
- Chapter IV presented the full description of the CGIS language and the CGIS API. The design decisions have been substantiated by arguing about the reasons for them and by weighting possible alternative approaches. CGIS has been contrasted to other approaches.
- Chapter V described the CGIS compiler, its usage and the runtime system. The compiler's inner workings, its data structures and algorithms have been explained. The various compiler phases were mentioned; Transformations, optimisations and implementation methods have been presented, some of them traditional, some induced by the peculiarities of the target or the language.
- Chapter VI presented several applications which have been implemented in CGIS. A sample of applications of various kinds and with a variety of characteristics has been described. We have seen their performance, compared them to CPU implementations, and argued about the reasons for particular performances. Impediments to high performance have been identified and mentioned.

Looking back, what was the main point of this work; what is the essence of it, and what are its main contributions?

The contribution of this work is the design of the data-parallel programming language CGIS and its implementation on GPUs. The CGIS system has been designed to meet a number of objectives (Section IV.1.a).

- *Familiarity:* The CGIS programming languages achieves familiarity to traditional programming languages by dividing an algorithm into its sequential constituents and a parallel layer on top of them. Sequential and parallel parts are neatly separated. The syntax and semantics of the sequential kernels resemble standard imperative languages such as C or Pascal. CGIS features cautious extensions of syntax and semantics to cope with vectorial data types and the restrictions of the target.

- *Readability:* While still retaining familiarity, CGIS achieves high readability both by extending and by restricting traditional syntactical and semantical features. In particular, a clear distinction of global and local data, clear and precise information flow between them and the avoidance of interdependencies between parallel computations result in a well-defined and easily understandable semantics. This is of great importance in parallel computations. Within kernels, scalar operations have been lifted to vectorial operations in an orthogonal way. Syntax and semantics localise cause and effect of computation and try to make the program text unambiguously readable.

- *Abstraction:* Abstraction is a prerequisite for several other goals. For one thing, abstraction means to disburden the programmer from unintuitive restrictions of the hardware. This can be seen in the orthogonal vectorial and scalar operations inside kernels and in the memory access mechanisms. Thus, abstraction is a key contribution towards readability and familiarity. Furthermore, abstraction is a key component in the technical basis of `cgisc`. Only by abstracting the hardware it is possible to retarget the language quickly to different architectures. In particular, the programmer is freed from interacting with graphics APIs and uses the GPU just as a numerical co-processor to which a parallel computation is outsourced. The highest abstraction, of course, is the unifying abstraction of GPUs and SIMD-CPUs. This is an abstraction of paradigms for parallel computing: CGIS programs describe independent, data-parallel computations on primitive values, and the compiler translates these into parallel executions of kernels on locally-SIMD-targets (GPU) or sequential executions of cross-kernel-SIMD-parallelism on small-scale SIMD targets (CPU)..

- *Compatibility:* CGIS is available for several targets. A key component to this end is the choice of OPENGL and standardised assembly languages as the target platform. The dependencies on operating system and windowing system are confined to the CGIS runtime library. Other features necessary for compatibility are the transformations to cope with restrictions on memory access and control flow. This is a kind of compatibility with respect to various generations of similar hardware.

- *Adaptability:* During the course of this project, new architectures kept being created. Obviously, new characteristics require new considerations. For example, the inclusion of native integral types with the G80 generation lead to a host of changes to language, compiler and runtime system. For a large part, though, everything that could be kept in common for the various targets is internally abstracted away.

The pattern matching code generation system plays an important role in this, because it allows to *localise* changes to code generation to its smallest constituents, while keeping the main part of the code generation static. Thus, `cgisc` has passed this test.

- *Efficiency:* The efficiency of CGIS has been proven through a variety of applications. Naturally parallel algorithms can be efficiently implemented through CGIS on GPUs, and this lead to large, sometimes huge, performance benefits. It has been shown that the time spent to transfer data to the GPU and back again can render outsourcing unattractive. This happens when the actual computation takes so little time even on the CPU that the performance gain vanishes compared to the additional cost of transferring the data. For many algorithms, however, including data transfer times and even one-time setup costs still shows a considerable performance increase over contemporary CPUs. Efficiency has been thoroughly proven.
- *Controllability:* Programmers can direct the compiler by a variety of means. Hints let the programmer guide the code generation and optimisation process. This is a way to provide additional information about the algorithm to the compiler, and to adapt the code to characteristics of a particular hardware platform's combination of CPU and GPU. Also, packing specifiers enable programmers to dictate the data layout, offering further capabilities for performance increases and of interfacing with other data producers and consumers.
- *Visualisability:* CGIS offers a primitive, integrated visualisation for simulations implemented in CGIS. Applications desiring a more flexible access to the data can manipulate the data inside the GPU memory, thus offering a way to plug CGIS into existing visualisation environments.

All in all, CGIS meets its stated goals.

The means wherewith this is achieved is the CGIS compiler, `cgisc`, and the runtime system. CGIS is compiled down to the common assembly languages of OPENGL instead of leaving the actual code generation to a third-party compiler. This has detriments, in that it is not possible to benefit from vendor-optimised code generation tools, but it paves the way for the higher-level abstraction and preserves independence. Also, it is an exercise in adapting traditional code generation and optimisation algorithms to the new setting.

The CGIS compiler, `cgisc`, employs a variety of optimisations. Some of them are just standard transformations based on data flow analyses, such as constant propagation or dead code elimination. In a few cases, we have seen the demand for a modification of the traditional algorithms to cope with the characteristics of the language and the target.

A recurring theme was basing the algorithms on components of larger vectorial types. This was provoked by CGIS' capabilities in swizzling operands and masking targets and by the hardware's capabilities and requirements for these modifiers. Resulting from this were a more dense register allocation and more fine-grained data-flow analyses and optimisations in familiar fields such as dead code elimination.

Another problem to be tackled was the handling of data in streams. The hardware's restriction of memory accesses to non-overlapping arrays of primitive types and read-only/write-only memory formed one boundary of this optimisation. The desire for abstraction by offering streams of structs and smooth transition to structs of streams and

by offering a semantically strict implementation of read-write streams necessitated data packing and reordering algorithms.

The compiler `cgisc` is also the first real-life software implementing the `OORS` system. This device proved to be a useful tool in the code generation process. The experiences in `cgisc` and the needs of the compiler lead to changes in `OORS`, further enhancing its capabilities and usability.

CGIS and `cgisc` certainly are important developments and useful tools for a particular goal. As it happens, they are not the only tools developed for these purposes. Of particular importance in the related work is BROOK. Concurrently and independently developed, BROOK today presents a viable choice for GPGPU. It offers a similar view on GPUs and thus a similar abstraction. (This can be seen as an additional indication for the viability of the model.)

As mentioned in Section IV.6, the CGIS programming model allows for a cleaner separation of CPU and GPU work. By expressing the whole sub-algorithm to be implemented on the parallel target in a single language in a single source file, the programmer has a better overview over the algorithm than in the mixed kernel/C model of BROOK. Other benefits of CGIS' choice on syntax and semantics for single kernels and their parallel executions have been mentioned in Section IV.6. All in all, the CGIS model does not offer a *fundamentally new* approach to GPGPU, but higher comfortability for certain tasks.

And what remains to be done in the future? What are the key points for future advances in the computing environments and in CGIS?

Let us first consider GPUs and GPGPU. At all times, the development of CGIS has been hampered by driver problems. Curious performance changes from one version to the next, and not completely implemented specifications are a continuous problem for GPGPU, not only for CGIS. Vendor initiatives such as CUDA provide a much more restricted set of functions to access GPUs detached from graphics APIs, and a more usual abstraction than these APIs. The abstraction mechanisms of CUDA also remedy one of the problems present in customary GPGPU languages. If it just comes to GPU programming on NVIDIA GPUs, compiling from CGIS to CUDA certainly offers less of an advantage than compiling to OPENGL in the pre-CUDA GPGPU world.

The GPU backend will be developed further. It is likely that a more detailed knowledge about quantitative characteristics of GPUs can lead to better optimisations. A first step in this direction would be to perform tests on ATI GPUs, which also in the most recent generation offer a unified shading model. A further, perhaps much deeper change presents itself in the future with the upcoming OPENGL 3.0, which will offer a quite radical update getting rid of legacy features: *"Get back to the bare metal"* [G07b]. The official presentation of OPENGL 3.0 is scheduled for end of September 2007.

But the future of CGIS is going to lie not solely in the area of GPGPU, just as it is not only a GPGPU language. A key feature of CGIS is the abstraction of a range of completely different target platforms. The explicit parallelism and the restriction of global data to bounded arrays and singulars result in a common, unified abstraction. Starting from this high-level algorithm representation, transformations can create code which efficiently runs on one of a variety of targets. Although the actual platforms of GPUs and SIMD CPUs are radically different, CGIS programs form a common starting point from which a reasonable implementation can be derived.

It is this versatility which enables CGIS to give an impact on future developments. With future CPU architectures incorporating more cores and a convergence of GPUs

and CPUs on a single chip [I07, H07, S07b], a unified abstraction mechanism is the only way to handle the increasing complexity of the emerging architectures. CGIS offers such an abstraction mechanism. Practically speaking, its use for efficient implementations on GPUs and on SIMD CPUs has been shown in this work and in [FLW07]. From a broader design perspective, it shows how to arrive at two quite different approaches to parallelism from a common language. At its very heart, CGIS simply prescribes a programming model of a multitude of independent, sequential computations with (statical) uniform instruction stream. It shows its roots as a GPGPU language evidently, but just by the specification of parallelism on multiple levels, also SIMD CPUs can be exploited. These two levels of parallelism in CGIS (SIMD parallelism on vectorial types inside procedures, SPMD parallelism with `forall` loops) are beneficially transformed into target-specific parallelism. This shows that just by an appropriate *specification* of *logical* parallelism, quite different kinds of *physical* parallelism can be exploited.

Just like CGIS was not simply an extension of an existing, sequential language but was implemented as a new language to better express the key features of parallelism and data dependencies, the languages arising in the next years might not directly build upon CGIS. But the experiences gathered in designing and implementing CGIS form a good starting point for future developments. CGIS might play its part as a precursor for the unified, parallel languages of the future.

Bibliography

[A00] AMD. *3DNow! Technology Manual*, March 2000.

[A06] AMD. *ATI CTM Guide*, 2006.

[A07a] AbsInt. Graph Description Language in a Nutshell. `http://www.aisee.com/gdl/nutshell/`, 2007.

[A07b] AMD. ATI Radeon HD 2900 Series – GPU Specifications. `http://ati.amd.com/products/Radeonhd2900/specs.html`, August 2007.

[A07c] ATI. Radeon X1950 Graphics Technology – GPU Specifications. `http://ati.amd.com/products/RadeonX1950/specs.html`, 2007.

[AG99] Anthony A. Apodaca, Larry Gritz. *Advanced RenderMan*. Morgan Kaufmann, 1999.

[AK02] Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[AMW95] Martin Alt, Florian Martin, Reinhard Wilhelm. Generating Analyzers with PAG. Technical Report A10/95, Universität des Saarlandes, 1995.

[B65] J. E. Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[B68] Kenneth E. Batcher. Sorting Networks and their Applications. In *Proceedings of AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[B75] W. H. Burge. Stream Processing Functions. *IBM Journal of Research and Development*, pages 12–25, January 1975.

[B06] David Blythe. The Direct3D 10 System. In *Proceedings of SIGGRAPH 2006*, pages 724–734, 2006.

[BFHSFHH04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of SIGGRAPH*, pages 777–786, 2004.

[BFRR01] Thomas Bräunl, Stefan Feyrer, Wolfgang Rapf, Michael Reinhardt. *Parallel Image Processing*. Springer-Verlag, 2001.

[BHK03] Mauricio Breternitz Jr., Herbert Hum, Sanjeev Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *Proceedings of the 12th International Conference on Parallel Architecture and Compilation Techniques 2003 (PACT'03)*, 2003.

[BR96] R. Baldwin, Ronald L. Rivest. The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. RFC 2040, 1996.

[C82] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Reprinted as [C04].

[C04] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 39(4):66–74, April 2004. Reprinted version of [C82].

[CT98] Tom Christiansen, Nathan Torlington. *Perl Cookbook*. O'Reilly, 1998.

[D25] Albrecht Dürer. *Underweysung der messung mit dem zirckel und richtscheyt (German)*. Hieronymus Andreae, 1525.

[D02] Jack Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *International Journal of High Performance Applications and Supercomputing*, 16(1/2):1–111/115–199, 2002.

[DHEKLAJKDGB03] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, Ian Buck. Merrimac: Supercomputing with Streams. In *Proceedings of the International Conference on Supercomputing 2003 (SC'03)*, November 2003.

[E03] Wolfgang F. Engel, editor. *Shader X2: Tips & Tricks with DirectX*. Wordware Publishing, 2003.

[E04] Bruce Eckel. *Thinking in Java (4rd ed.)*. Prentice Hall, 2004.

[EVG04] Manfred Ernst, Christian Vogelgsang, Günther Greiner. Stack Implementation on Programmable Graphics Hardware. In Bernd Girod, Hans-Peter Seidel, Marcus Magnor, editors, *Proceedings of the 9th International Workshop "Vision, Modeling, and Visualization" (VMV'04)*, pages 255–262, 2004.

[F06] Freescale. *AltiVec Technology Programming Environments Manual (Rev. 3)*, April 2006.

[F08] Nicolas Fritz. *Exploiting SIMD Parallelism with the CGiS Compiler Framework*. PhD thesis, Universität des Saarlandes, 2008. To appear.

[FLW07] Nicolas Fritz, Philipp Lucas, Reinhard Wilhelm. Exploiting SIMD Parallelism with the CGiS Compiler Framework. In Vikram Adve, María Jesús Garzarán, Paul Petersen, editors, *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, LNCS. Springer-Verlag, October 2007.

[G70] Martin Gardner. The Fantastic Combinations of John Conway's new Solitaire Game "life". *Scientific American*, pages 120–123, October 1970.

[G71] Henri Gouraut. Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers*, 20(6):623–629, June 1971.

[G95] Andrew S. Glassner. *Principles of Digital Image Synthesis 2*. Morgan Kaufmann, 1995.

[G02] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Morgan Kaufmann, 2002.

[G05] Gernot Gebhard. CGiS Implementations. Fortgeschrittenenpraktikum (German), Universität des Saarlandes, April 2005.

[G06] Gernot Gebhard. A Pattern Matcher Generator for Retargetable Code Generation and Optimisation. Diplomarbeit, Universität des Saarlandes, 2006.

[G07a] GNU. Autoconf. http://www.gnu.org/software/autoconf/, 2007.

[G07b] Michael Gold. OpenGL 3 Overview. Presentation at Siggraph, August 2007.

[GL07] Gernot Gebhard, Philipp Lucas. OORS: An Object-Oriented Rewrite System. *Computer Science and Information Systems (ComSIS)*, 4(2):1–26, December 2007.

[GST07] Daniel Göddeke, Robert Strzodka, Stefan Turek. Performance and Accuracy of Hardware-Oriented Native-, Emulated- and Mixed-Precision Solvers in FEM Simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, January 2007.

[H63] George Hutchinson. Partitioning Algorithms for Finite Sets. *Communications of the ACM*, 6:613–614, October 1963.

[H07] Phil Hester. Presentation at AMD Technology Analyst Day, July 2007.

[HP03] John L. Hennessy, David Patterson. *Computer Architecture: A Quantitative Approach (3rd edition)*. Morgan Kaufmann, 2003.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, Pat Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of I3D*, pages 167–174, 2007.

[I85] IEEE. Standard for Binary Floating-Point Arithmetic. Standard 754-1985, 1985.

[I06a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, November 2006. 5 volumes.

[I06b] Intel. *Intel Itanium Architecture Software Developer's Manual*, January 2006. 3 volumes.

[I07] Intel. Intel Provides Details On New Products, Initiatives For Higher-Performing, More Efficient Computers. `http://www.intel.com/pressroom/archive/releases/20070416comp_b.htm`, April 2007.

[IMR83] Oscar H. Ibarra, Shlomo Moran, Louis E. Rosier. On the Control Power of Integer Division. *Theoretical Computer Science*, 24(1):35–52, 1983.

[IST05] IBM, SCEI, Toshiba. *Cell Broadband Engine Architecture*, 2005.

[K05] Donald E. Knuth. *The Art of Computer Programming, Vol. 4, Fascicle 3*. Addison-Wesley, 2005.

[K06] John Kessenich. *The OpenGL Shading Language, Version 1.20*, September 2006.

[K07] Khronos. OpenGL Extension Registry. `http://www.opengl.org/registry/`, August 2007.

[KBR04] John Kessenich, Dave Baldwin, Randi Rost. *The OpenGL Shading Language, Version 1.10*, April 2004.

[KW03] Jens Krüger, Rüdiger Westermann. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In *Proceedings of SIGGRAPH*, pages 908–916, 2003.

[L68] P. J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-notations. *Communications of the ACM*, 8(2/3):89–101, 158–167, 1968.

[LA00] Samuel Larsen, Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, June 2000.

[LKSSO06] Aaron E. Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, John D. Owens. Glift: Generic, Efficient, Random-Access GPU Data Structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[LM01] E. Scott Larsen, David McAllister. Fast Matrix Multiplies using Graphics Hardware. In *Proceedings on the ACM/IEEE Conference on Supercomputing*, 2001.

[LMB92] John R. Levine, Tony Mason, Doug Brown. *lex & yacc (2nd edition)*. O'Reilly, 1992.

[M97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[M04] Michael Macedonia. The GPU Enters Computing's Mainstream. *Computer*, 36(10):106–108, October 2004.

[M07a] Microsoft. DirectX Resource Center. `http://msdn.microsoft.com/directx`, August 2007.

[M07b] Microsoft. Windows Vista Enterprise Hardware Planning Guidance. `http://technet.microsoft.com/en-us/windowsvista/aa905075.aspx`, August 2007.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proceedings of SIGGRAPH*, pages 896–907, 2003.

[MHC04] Henrique S. Malvar, Li-wei He, Ross Cutler. High Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'04)*, 2004.

[MQP02] Michael D. McCool, Zheng Qin, Tiberiu S. Popa. Shader Metaprogramming. In *Proceedings of the Eurographics Workshop on Graphics Hardware*. ACM, 2002. *Revised version*.

[MS68] T. H. Myer, I. E. Sutherland. On the Design of Display Processors. *Communications of the ACM*, 11(6):410–414, June 1968.

[MT97] Tomas Möller, Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[MTPCM04] Michael McCool, Stefanus Du Toit, Tiberiu S. Popa, Bryan Chan, Kevin Moule. Shader Algebra. In *Proceedings of SIGGRAPH*, pages 787–795, 2004.

[N00] NVIDIA. NV_register_combiners. OpenGL Extension 191, 2000.

[N05] NVIDIA. *Cg Toolkit User's Manual, Release 1.5*, September 2005.

[N06] NVIDIA. GeForce 8800 Architecture Overview. Technical Report TB-02787-001_v1.0, NVIDIA, November 2006.

[N07a] NVIDIA. *CUDA Programming Guide Version 1.0*, June 2007.

[N07b] NVIDIA. NV_fragment_program4. OpenGL Extension 335, 2007.

[N07c] NVIDIA. NV_gpu_program4. OpenGL Extension 322, 2007.

[N07d] NVIDIA. *PTX ISA Version 1.0*, June 2007.

[NNH99] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[O02a] OpenGL ARB. ARB_fragment_program. OpenGL ARB Extension 27, 2002.

[O02b] OpenGL ARB. ARB_vertex_program. OpenGL ARB Extension 26, 2002.

[OLGHKLP07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[P75] Bui Tuong Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18:311–317, June 1975.

[P04] Timothy John Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, March 2004.

[P07] Matthias Parbel. Nvidia rückt dichter an die Spitze des Grafikchipmarktes (German). `http://www.heise.de/newsticker/meldung/93543`, July 2007.

[PDCJH03] Timothy John Purcell, Craig Donner, Mika Cammarano, Henrik Wann Jensen, Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 41–50, 2003.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In *Proceedings of Eurographics*, 2007.

[PM03] Craig Peeper, Jason L. Mitchell. Introduction to the DirectX 9 High-Level Shader Language. In Engel [E03].

[R94] Ronald L. Rivest. The RC5 Encryption Algorithm. In *Proceedings of the Leuven Workshop on Fast Software Encryption*, pages 86–96, 1994. Revised version of 1997.

[R97] Raúl Rojas. How to Make Konrad Zuse's Z3 a Universal Computer. *IEEE Annals of the History of Computing*, 20(3):51–54, 1997.

[R06] RapidMind. *RapidMind Development Platform Reference Manual*, October 2006.

[R07] Justin R. Rattner. Tera-scale Computing – A Parallel Path to the Future. `http://softwarecommunity.intel.com/articles/eng/1275.htm`, May 2007.

[RWP05] Gang Ren, Peng Wu, David Padua. An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 89b, 2005.

[S97a] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[S97b] Bjarne Stroustrup. *The C++ Programming Language (3rd edition)*. Addison-Wesley, 1997.

[S99] Robert W. Sebesta. *Concepts of Programming Languages (4th ed.)*. Addison Wesley Longman, 1999.

[S07a] Bjarne Stroustrup. Bjarne Stroustrup's FAQ. `http://www.research.att.com/~bs/bs_faq.html`, April 2007.

[S07b] Sun Microsystems. UltraSPARC T2 Processor. `http://sun.com/t2`, August 2007.

[SA06] Mark Segal, Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.1)*, July 2006.

[SAGMRSTW05] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen R. Marschner, Erik Reinhard, Kelvin Sung, William B. Thompson, Peter Willemsen. *Fundamentals of Computer Graphics, 2nd Edition*. A K Peters, 2005.

[SR71] R. Stefanelli, Azriel Rosenfeld. Some Parallel Thinning Algorithms for Digital Pictures. *Journal of the ACM*, 18(2):255–264, 1971.

[T99] Simon Thompson. *Haskell. The Craft Of Functional Programming*. International Computer Science Series. Addison-Wesley, 2nd edition, 1999.

[THO02] Chris J. Thompson, Sahngyun Hahn, Mark Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *Proceedings of the 35th IEEE/ACM International Symposium on Microarchitecture (MICRO'35)*, pages 306–317. IEEE, 2002.

[TL00] Claes Thornberg, Björn Lisper. Elemental Function Overloading in Explicitly Typed Languages. In Markus Mohnen, Pieter Koopman, editors, *Proceedings of the 12th International Workshop of Implementation of Functional Languages*, pages 31–46, September 2000.

[TPO06] David Tarditi, Sidd Puri, Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 325–335, 2006.

[W05] Eric W. Weisstein. Mandelbrot Set. `http://mathworld.wolfram.com/MandelbrotSet.html`, November 2005.

[W06] Ben Woodhouse. GLee 5.21. `http://elf-stone.com/glee.php`, November 2006.

[WM95] Reinhard Wilhelm, Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

Index

Entries written **in bold** represent more important sources of information for the particular entry.