Technische Universität München
Forschungsinstitut caesar in Bonn

GPU^{++} An Embedded GPU Development System for General-Purpose Computations

Thomas C. Jansen

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 26.04.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.09.2007 angenommen.

Zusammenfassung

Der Einsatz der Grafikkarte (GPU) zur Beschleunigung allgemeiner Berechnungen hat sich im Laufe der letzten Jahre zu einer wichtigen Technik im wissenschaftlichen Bereich entwickelt. Aufgrund der dabei verwendeten grafik-orientierten Konzepte und Werkzeuge, führt diese Methode jedoch zu einer spürbaren Steigerung der Entwicklungskomplexität im Vergleich zur klassischen Umsetzung für den Hauptprozessor. Die damit verbundene Kostensteigerung verhindert bis jetzt den breiten industriellen Einsatz dieser Technik.

Diese Arbeit beschreibt einen neuartigen Ansatz zur generellen Programmierung der Grafikkarte – das Gpu^{++} Entwicklungssystem. Dieses System ist nahtlos in die Programmiersprache C^{++} integriert und ermöglicht so die gleichzeitige Verwendung der CPU und GPU mittels einer bekannten Befehlssyntax. Zudem werden computer-grafische Konzepte wie die verschiedenen Berechnungseinheiten der GPU und deren Vektorarchitektur, effizient abstrahiert. So ermöglicht Gpu^{++} eine einfache und schnelle Integration GPU-beschleunigter allgemeiner Berechnungen in einen existierenden Entwicklungsprozess – und hebt damit die Unterschiede zwischen Grafik- und Hauptprozessor größtenteils auf.

Die Evaluierung des in dieser Arbeit vorgestellten Entwicklungssystems geschieht durch die CPU- und GPU-basierte Umsetzung bekannter Algorithmen, wie der Multiplikation großer Matrizen, Sortieren, Schnelle Fourier-Transformation und die gefilterte Rückprojektion zur Berechnung einer tomografischen Rekonstruktion. Obwohl die auf Gpu^{++} aufbauenden Implementationen die Geschwindigkeit der CPU-basierten Umsetzungen um teilweise mehr als drei Größenordnungen übertreffen, hat dies keine spürbare Steigerung der Softwarekomplexität und Entwicklungszeit zur Folge.

Diese Ergebnisse demonstrieren überzeugend, dass sich die Entwicklungskomplexität und -zeit für die Umsetzung allgemeiner Berechnungen auf der Grafikkarte durch den Einsatz des Gpu^{++} Entwicklungssystems auf ein Niveau senken lassen, welches mit der klassischen CPU-basierten Implementation vergleichbar ist – ohne dabei auf die enormen Geschwindigkeitsvorteile des Grafikprozessors verzichten zu müssen.

Abstract

Using the graphics processing unit (GPU) to accelerate general-purpose computations has become an important technique in scientific research. However, the development complexity is significantly higher than for CPU-based solutions, due to the mainly graphics-oriented concepts and development tools for GPU-programming. As a consequence, general-purpose computations on the GPU are mainly discussed in the academic domain and have not yet fully reached industrial software development.

This thesis presents a novel contribution to general-purpose GPU programming – the GPU^{++} development system. It features a seamless integration into the C^{++} programming language to address graphics hardware via a familiar syntax, an abstraction layer to efficiently hide the different computation frequencies, and a novel approach to relax the vector processor paradigm of the GPU. The GPU^{++} development system enables software engineers to embed GPU-based development into their existing software engineering workflow, thus, largely dissolving the differences between main and graphics processor.

The developed system has been evaluated by the implementation of well-known general-purpose algorithms on both processor platforms – including general matrix multiplication, sorting of values, the fast Fourier transform, and tomographic reconstruction via filtered back-projection. The GPU^{++} implementations outperform the CPU solutions by up to three orders of magnitude without a noticeable increase in source code complexity.

These results clearly demonstrate that the novel GPU^{++} development system significantly reduces the source code complexity and development time of general-purpose GPU applications to a level which is comparable to main processor implementations, while obtaining the tremendous performance advantages of today's graphics hardware architectures.

Acknowledgments

Over the last seven years I have had the privilege to work with a variety of people who have made my time at caesar an enjoyable and intellectually stimulating experience. I would like to thank all the people who have helped me and contributed to this dissertation.

First and foremost, I would like to thank my advisor Erwin Keeve who gave me the opportunity to work with him during my stay at the caesar research center. Having Erwin as an adviser has been an amazing experience. He was willing to take a chance on my research from the beginning, and has always pushed me to fill in that one last detail to elevate the level of my thinking and my work. I would also like to thank the other members of my reading committee, Prof. Karl-Heinz Hoffmann and Prof. Rüdiger Westermann, for their discussions on this dissertation. Their insights greatly improved this thesis.

One of the great things about being at caesar's *Surgical Systems Lab* working on a variety of different topics is that I have gotten to work closely with a lot of people. I am extremely grateful for the collaborations with Bartosz von Rymon-Lipinski and Nils Hanßen – their insights and fruitful discussions about computer graphics, software engineering and image processing greatly improved this thesis. I also would like to thank anyone I have worked with: Lutz Ritter, Fabio Fracassi, Khai Binh Duong, Timo Dreiseidler, Marco Wedekind, Kristian Raic, Andreas Zollorsch, and Alexander Winter. They have been a great sounding board for many of my ideas and they have spent countless hours reviewing my papers and talks, and my work is much better for it.

I am also deeply indebted to my family for believing in me and for giving me the inspiration to apply and complete a doctoral degree. Their love, guidance, and encouragement have been a constant source of strength for me.

And last but not least, my heartfelt thanks go out to my beloved better-half, Kathrin, who has been so patient and supportive since I started graduate school. She has been my inspiration and provided encouragement when research progress was slow, and when research felt like spiraling out of control, she brought serenity. It should be no surprise that this dissertation would be impossible without her. I dedicate this thesis to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Commodity graphics hardware has evolved tremendously over the last years – it started with basic polygon rendering via 3dfx's Voodoo Graphics in 1996, and continued with custom vertex manipulation four years later, the graphics processing unit (GPU) now has improved to a full-grown graphics-driven processing architecture with a speed-performance approx. 750 times higher than a decade before (1996: 50 mtex/s, 2006: 36,8 btex/s). This makes the GPU evolving much faster than the CPU, which became approx. 50 times faster in the same period (1996: 66 SPECfp2000, 2006: 3010 SPECfp2000) [69]. Figure 1.1 shows the GPU performance over the last ten years and how the gap to the CPU increases. Experts believe that this evolution will continue for at least the next five years [66, 76].

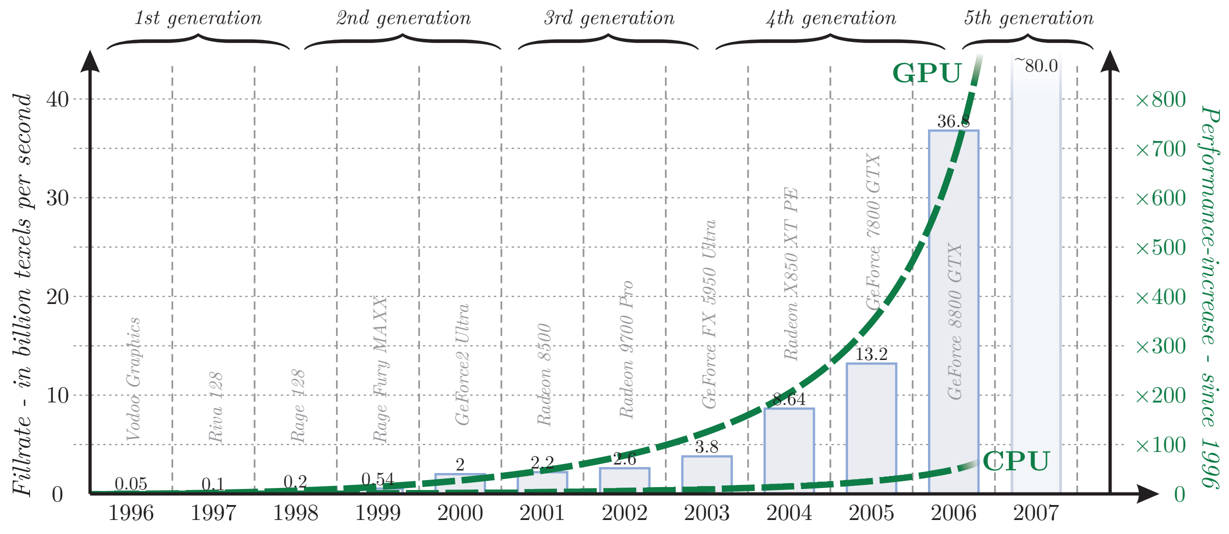


Figure 1.1: The performance-increase of computer graphics hardware over the last decade (using the "texels per second" metric). The green trend line shows that the GPU doubles its speed-performance every 13 months (i.e. GPUs of 2006 are approx. 750 times faster than GPUs of 1996). In contrast, the performance of the CPU doubles only every 22 months [69].

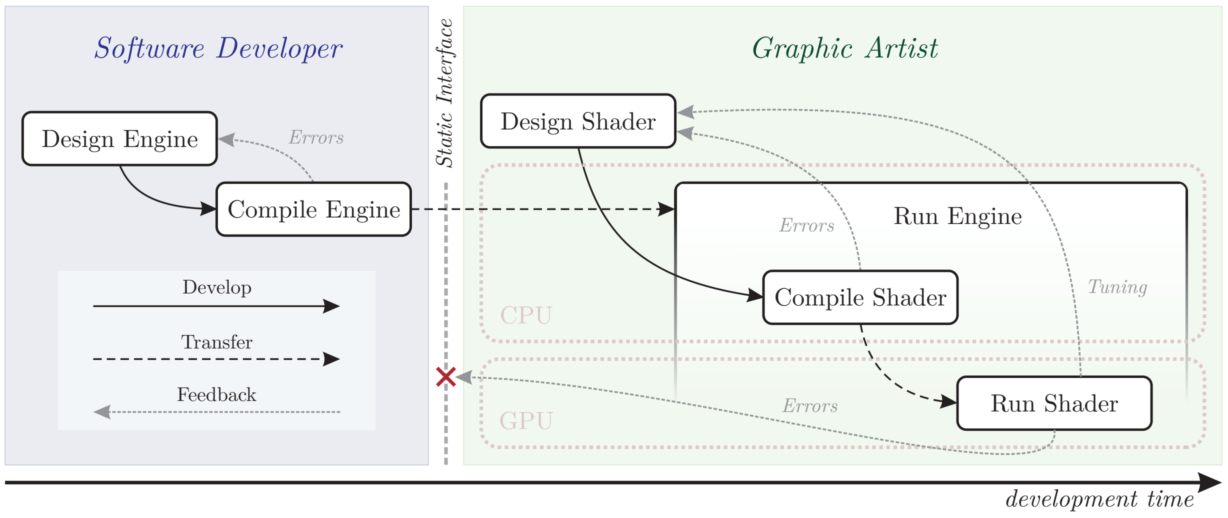


Figure 1.2: The "visual paradigm" knows two participants – the *software developer* and the *graphics artist* – each having its own workflow. The software developer creates the "engine" that is used by the graphics artist to create "shader" programs (that run as part of the engine).

As the name implies, the GPU was initially designed for accelerating graphical tasks. However, it was soon being exploited for performing non-graphical computations, for instance, the work of Lengyel et al. uses graphics hardware to compute robot motion, the CypherFlow project by Kedem and Ishihara exploits the graphics accelerator to decipher encrypted data, and a GPU-based computation of Voronoi diagrams has been presented by Hoff et al. [72, 60, 54]. Today, implementations for various general-processing algorithms exist for the graphics hardware, e.g. sorting, linear algebra, Fourier analysis, partial differential equations, and tomographic reconstruction [63, 70, 56, 113, 19]. Nevertheless, while more and more algorithms are implemented for graphics hardware, most of the work stays in the academic field and has not yet found its way into industrial software engineering.

The reason is that GPU-based application development is *much more complex*, mainly because the developer has to be an expert in two domains – in the application's domain, and in *computer graphics*. This means that changing the graphics-oriented paradigms and corresponding GPU development tools may significantly reduce development complexity.

Most of the existing GPU-based development systems are founded on the graphics-oriented paradigm that is illustrated in figure 1.2 – the *visual paradigm*. This approach has been influenced by the entertainment and special-effects industry, where the *software developer* creates the so-called "rendering engine" and the *graphics artist* uses the engine to create so-called "shader programs" that compute visual phenomena. For instance, a very popular rendering engine is the RenderMan software – while the software itself is developed by Pixar, the "visual effects" (i.e. shader programs) are created by individual special-effects companies [119]. In this case, the visual paradigm makes sense, because the *graphics artists* and the *software developers* usually do not know each other – and therefore, have separated workflows that are tailored to the specific needs of each group.

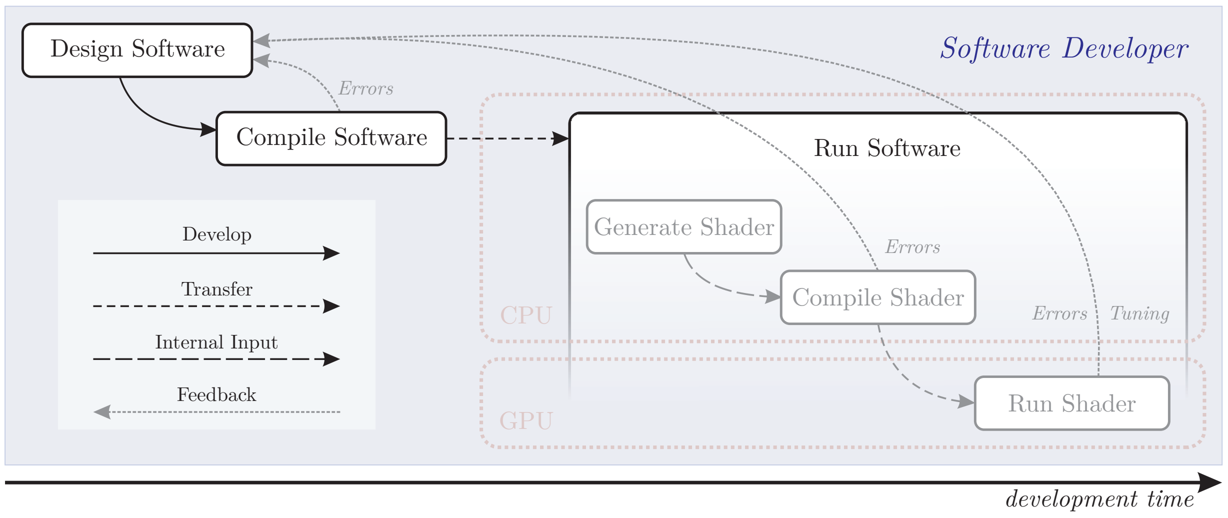


Figure 1.3: The "algorithmic paradigm" knows only the *software developer*, who creates the whole "software" including code for CPU and GPU. The GPU-based functionality of the software has to be extracted and transformed to the graphics hardware's internal format.

Examples for GPU-based development systems that are founded on the *visual paradigm* are RenderMan, Pfman, Interact-SL, RTSL, Cg, and GLslang [49, 91, 97, 99, 77, 104]. However, such development systems are inappropriate to implement general-purpose algorithms on the GPU, mainly because of the following two reasons:

- Due to the distinct specialized workflows, *broad knowledge* is required in both disciplines – software engineering (*developer*) *and* computer graphics (*artist*).
- The visual paradigm also forces the software developer to *leave his familiar development environment* and to develop in the graphics artist's programming language.

Furthermore, because CPU and GPU-based code are developed in different programming languages, additional *binding code* is required to "glue" the different functionality together.

Please note that there are GPU-based development systems that relax the aforementioned issues. For instance, Brook for GPUs, CUDA, and CTM are not graphics-oriented, but they still separate between CPU- and GPU-based code [18, 89, 106]. On the other hand, Sh allows mixing the code of both processor platforms in the same programming language, but still requires knowledge in the computer graphics domain [101].

As a matter of fact, to efficiently develop general-purpose applications that are accelerated by the GPU, a significantly different approach is required – the *algorithmic paradigm*. Figure 1.3 illustrated the paradigm: the *software developer* creates the complete "software" that contains code for the main *and* the graphics processor at the same time. This means that there is a single development environment, where the same programming language is used to define CPU- and GPU-based code side by side and no *binding code* is required that connects the variables of the different processor platforms. The "Gpu^{++} *development system*" that will be presented in this thesis, makes use of the algorithmic paradigm.

In other words, the *algorithmic paradigm* conceptually eliminates the separation between both processor platforms. In practice, GPU-based development systems that are founded on this paradigm have to deal with the following three challenges:

Compact Set of Generic Concepts – The development system has to present a *consistent set of generic concepts* to abstract techniques and terminology that are specific to computer graphics. For instance, *vertex* and *fragment* processing is hard to understand by the common software developer and need to be abstracted by a generic concept. The challenging task is that the new concepts have to be easier to learn, compared to the graphics-oriented concepts of the visual paradigm. While this is hard to evaluate, the rule-of-thumb is: the less concepts, the easier to learn.

Uniform Development Environment – Both processor platforms have to be accessible in the same development environment. In other words, CPU- and GPU-based code should be mixable in the same source code files, i.e. the code is specified in the same general-purpose programming language. Ideally, the software developer is able to use the same familiar syntax, the same compiler collection, and the same testing environment he is used to – for both processor architectures. Please note that "familiar syntax" means that mathematical expressions are specified in the same way, no matter of the target platform – i.e. that code should be interchangeable.

Seamless Data Interchange Between CPU and GPU – Development systems that follow the algorithmic paradigm have to eliminate the need of "binding code" – variables have to be accessed easily on both processor architectures. In other words, computation results of the graphics hardware have to be directly accessible in CPU-based code, and main processor variables have to be usable as GPU-based inputs. Please note that the prior challenges mainly addressed the syntax and style of source code and the software development workflow, while this is focused on the seamless interchange and automatic transfer of data between the main and graphics processor.

The aforementioned challenges are accomplished by using the following two-stage approach: On a *theoretical level*, the existing graphics-oriented concepts for programming the GPU are abstracted to generic concepts, e.g. pixel and fragment processing is abstracted via a "unified kernel definition", and the vector processor concept is abstracted via "vector fusion". Furthermore, on a *practical level*, graphics-hardware-based code is seamlessly integrated into the general-purpose C^{++} programming language, which is realized by using advanced object-oriented techniques like "ad-hoc polymorphism" and "generic programming". To further reduce the development complexity and to improve run-time performance, a variety of optimization strategies are automatically applied to the GPU-based code.

The GPU^{++} development system will be evaluated extensively by implementing a broad set of general-purpose algorithms. This includes the performance comparison of CPU-based solutions with implementations for the graphics processor, as well as an analysis of the source code complexity for both processors to verify the aforementioned design goals.

1.1 Contributions

This dissertation makes several contributions to the areas of computer graphics, graphics hardware programming, software engineering, and general-purpose computations on GPUs:

- An approach is presented that allows the implementation of general-purpose algorithms on computer graphics hardware by seamlessly extending the C^{++} language using *ad-hoc polymorphism*. This technique enables developers to use the same environment and language for both platforms – main *and* graphics processor. This approach significantly reduces development complexity for GPU-based applications.
- A consistent set of high-level *graphics-independent* concepts is presented, including the stream approach, vector processing, and computation frequencies. These concepts enable developers to efficiently exploit the computational power of today's graphics accelerators without the need of being an expert in computer graphics, computer graphics hardware, or the different graphics APIs for programming the GPUs.
- An abstraction scheme that hides the different computation units on the GPU (vertex and fragment processing) is presented: The *unified kernel definition* is partitioned automatically by the development system to efficiently exploit the different processing units. This technique enables the developer to specify all GPU-relevant functionality in a single place, resulting in a significant reduction of the source code complexity.
- The aforementioned GPU development system is extended to support on-the-fly optimizations, like dead-code elimination, constant folding, algebraic simplification, and back-end-dependent substitutions. These optimizations enable developers to focus more on algorithms than on implementations. The optimizations have proved to be useful for a variety of general-purpose computations, leading to a significant speed-up.
- A novel technique for vector component separation and efficient *vector fusion* is presented, i.e. scalar operations are efficiently combined automatically to exploit the vector processor architecture of the GPU. This leads to a much better granulation in the optimization stage, and furthermore, enables the developer to freely choose between vector and non-vector programming without a loss of run-time performance.
- A novel approach for efficient graph traversal and graph processing is presented, based on the *class traits* paradigm that features a *generic and extensible interface* and *optimal run-time performance*. Where the aforementioned optimizations lead to better run-time performance, the traversal approach significantly improves the GPU-based compile-time performance – and therefore the overall processing speed.
- Finally, a variety of general-purpose algorithms is implemented on both processor platforms, main and graphics processor, to extensively evaluate the GPU-based development system. In all cases, the GPU-based implementation out-performs the CPU-based version by one to three orders of magnitude. Furthermore, the evaluation demonstrates that both implementations are of similar source code complexity.

1.2 Outline

The thesis begins in chapter 2 with a discussion of the related work, including historical and modern shading languages dedicated to computer graphics, as well as, the latest innovations in general-purpose languages for graphics hardware.

A consistent set of graphics-independent concepts for GPU programming is presented in chapter 3. Furthermore, it is shown how the aforementioned concepts confine the family of algorithms that can be accelerated by the graphics processing unit.

The major internal data structure (that represents any GPU-based code in the GPU^{++} development system) is described in chapter 4. This includes the embedded creating of the graph via ad-hoc polymorphism, as well as its fast processing using the novel *visual traits* technique. This processing infrastructure is the foundation for a variety of generic and GPU-specific run-time optimizations of GPU-based computations that are presented in chapter 5. Chapter 6 presents the binding of the optimized kernel program to the graphics back-end using the OpenGL low-level shading language as an exemplar implementation.

The GPU^{++} development system is evaluated in chapter 7 by implementing algorithms from digital signal processing, medical processing, and computer science. The run-time performance and source code complexity of the resulting implementations are further discussed in section 8. Finally, the thesis is concluded and areas of future research are suggested in chapter 9.

Additional information is presented in two appendices: Appendix A is a brief reference of the new data types, managing classes, and function calls. Appendix B contains basic examples, including their full source code and a line–by-line description. Both appendices do not contain conceptual information relevant to the thesis and were only included for the sake of completeness.

Chapter 2

Related Work

The idea to implement general-purpose algorithms on computer graphics hardware has been introduced more than fifteen years ago, when Lengyel et al. used a rasterization device for robot motion planning in 1990 and Cabral et al. accelerated tomographic reconstruction in 1994 [72, 19]. However, most of the early work has been designed as a proof-of-concept and never meant to be used in industry-driven software development. But the introduction of programmable commodity graphics hardware in 2001 pushed the popularity of this approach even further, and shortly after, the abbreviation GPGPU ("general-purpose computations on GPUs") was coined for this new domain [73, 50]. At that time, the only way to program graphics hardware was via the detour of shading languages, because no explicit GPGPU development tools where available for quite a long time – but shading languages were mainly designed for the graphics and entertainment industry. As a consequence, most of the work and research were focused on the implementation instead of the algorithm. While this has changed lately when several companies released commercial GPGPU tools and languages by the end of 2006, most of the GPU-based programming is still done via shading languages. This chapter will present a broad overview of programming solutions including both – shading and general-purpose languages for computer graphics hardware.

2.1 Shading Languages

In computer generated imagery, the visual appearance of an object is influenced by both, shape and shading. The *shape* of an object is based on its surface geometry and its position relative to the viewpoint. In contrast, the *shading* depends on its optical properties and the illumination environment. Impressive images can be rendered from objects with a simple shape but a complex shading. Therefore, both attributes should are usually treated separately. In the very beginning of computer generated imagery, most of the shading was done by scanning textured images and attaching them to geometrical objects [20]. Soon after, *procedural shading* was introduced that evaluates the final color of particular points on the object's surface using a set of (sometimes very complex) mathematical equations.

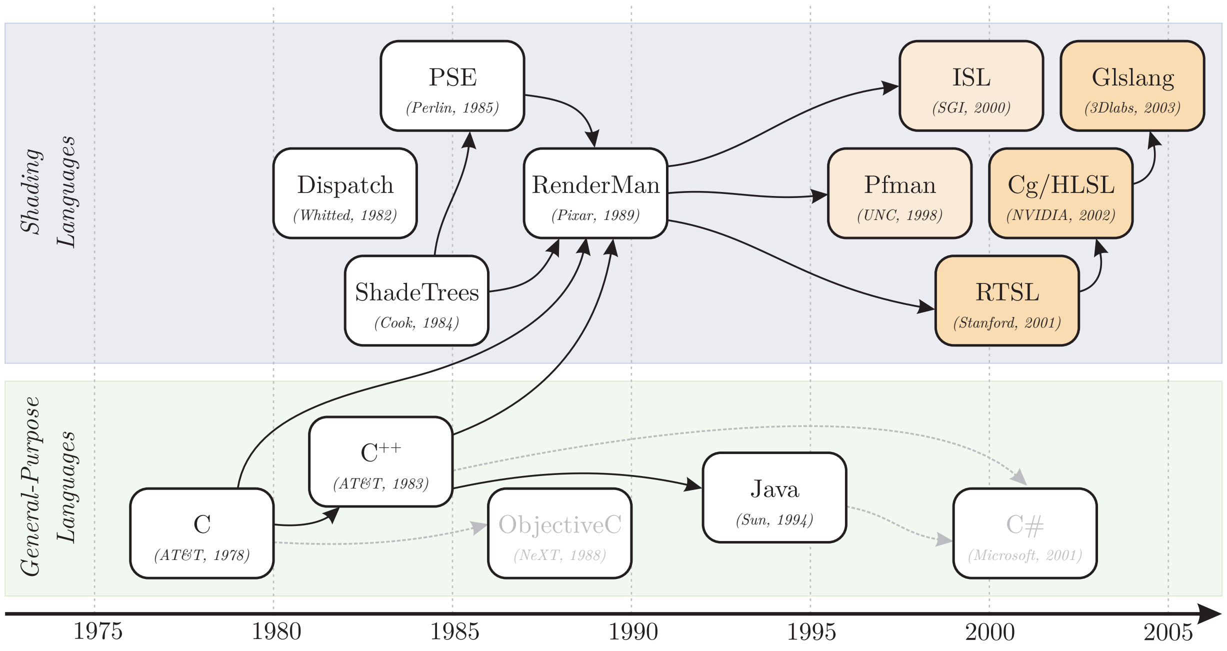


Figure 2.1: Evolution of shading languages. Most of them are inspired by the syntax and philosophy of C [61]. In the end of the 1990s, *interactive* shading languages entered the field, first for special hardware (lighter orange), then for consumer-market GPUs (darker orange).

Complex scenes usually contain many objects and each is associated to a different *shader* (i.e. the set of equations). Initially, shaders were hardwired in the rendering engine and selected via a *shader dispatch table* [125]. However, due to its *lack of portability* and the *required knowledge*, this technique was inappropriate for large productive environments like the movie industry, which resulted in a growing demand for higher-level languages. Figure 2.1 illustrates the evolution of these languages and how they influenced each other.

2.1.1 History

As part of his *shader tree* work, Robert Cook developed a system where shaders (given as a sequence of mathematical equations) were read in, parsed and executed at run-time by the rendering system [24]. Even if this was not a full shader language, it decoupled the shader description from the actual rendering architecture. This concept was extended by Perlin, who developed the Pixel Stream Editing Language (PSE) that included high-level constructs, such as conditionals, loops, functions, and a rich set of arithmetic and logical operators [98]. Perlin considered shading as a postprocess, so he rejected the concept of distinct shader types (as proposed by Cook) in favor of a singular "space function", i.e. a per-pixel shading of the already computed frame-buffer. However, such an approach is hard to apply within a global context, e.g. for ray-tracing and radiosity.

Based on the aforementioned work, Hanrahan et al. developed the RenderMan Shading Language (RMSL) as part of Pixar's RenderMan rendering system [49, 119]. The language is similar to "C" but tailored especially to shading calculations [61]. It offers a very small set of fixed data types (`color`, `point`, `float`, and `string`) and there is no way to define new types or high-level constructs like data structures or arrays. Many built-in functions are provided, e.g. for geometric computations, image processing and texture access. Shaders may be instantiated multiple times with possibly different values for its arguments and particular instances are attached to the objects in the scene. Today, RenderMan is the de-facto standard for photorealistic rendering in movies.

In 1998, Olano et al. introduced Pfman, the first real-time shading language (for the PixelFlow architecture). It is very similar to the aforementioned RenderMan shading language, but is slightly limited: only surface and light shaders are supported, fixed-point math has been added to avoid floating-point operations and C^{++} functions can be called directly from the shader [91]. Because of its scalability, the PixelFlow architecture was capable to render scenes at 30 frames per second, even for complex procedural shaders. Unfortunately, the PixelFlow hardware was too expensive and failed commercially.

A significantly different approach was introduced by Peercy et al.: Instead of using specific graphics hardware, the OpenGL software architecture itself is treated as a general SIMD computer, where the high-level shading description is translated into multiple rendering passes of the OpenGL library [97]. In combination with a small set of OpenGL extensions, a full implementation of the RenderMan shading language has been presented on top of this "SIMD model" approach – the so-called Interactive Shading Language (ISL). Unfortunately, the generality of this approach also is its major drawback – each command of the shader program is individually executed for all frame-buffer elements, hence, temporary values are written to the frame-buffer after one command and are read in for the next command. As a consequence, the ISL wastes most of its run-time for memory access and due to the large overhead of calls to the OpenGL programming interface.

Second generation GPUs started to provide (limited) programmability that was significantly extended in 2001 with the introduction of the "programmable vertex unit" [73]. Based on this new functionality, Proudfoot et al. developed the Stanford Real-Time Shading Language (RTSL), which was especially designed for interactive procedural shading [99]. The RTSL is based on a unified framework – the "programmable pipeline" – that provides the same syntax, types and operators for different "computation frequencies", i.e. computations done per scene, per object, per vertex and per fragment. However, this may result in a part-time CPU-based emulation of several computation frequencies. A computational model, based on the virtualization of several pipeline stages, is used to remove resource constraints of existing hardware architectures. Beside native operations supported by all GPUs (e.g. scalar and vector multiplication), the RTSL introduces so-called "canned functions" that are supported by special graphics hardware or are not supported yet, but will be in future (e.g. bump-mapping). Most of these techniques can be found in succeeding shading languages – the RTSL itself is no longer maintained, because it evolved into the Cg and HLSL shading languages that will be described in the next section.

2.1.2 Cg and HLSL

Some developers of the RTSL joined hardware vendor NVIDIA at the end of 2001, to define and implement a new shading language. This has been done in close collaboration with Microsoft, thus, the result is one language with two different names – NVIDIA's "C for graphics" (Cg) and Microsoft's "High-Level Shading Language" (HLSL) [77, 35, 45]. Even though the language is the same by syntax and semantic, they differ by *philosophy*: Cg was designed as an additional layer on top of both popular graphics APIs (i.e. OpenGL and DirectX) for a small performance penalty, while the HLSL offers a clean interface and avoids API overhead due to a tight integration into the DirectX framework.

Cg/HLSL is a further development of Stanford's RTSL. The language evolved broadly: many functions have been added to address the latest GPU functionality, control flow operators are supported, `half` and `fixed` have been added, vectors with up to four scalars and matrices with sizes up to 4×4 are supported, and some C^{++} techniques have been included. Changes can also be found in the architecture design: Although the concept of the "programmable pipeline" still exists, it is combined with the idea of a virtualized machine, leading to the concept of *language profiles*: subsets of the full language that are supported on particular hardware, computation frequencies, or APIs. Cg is in active development, even though most of the changes apply to the architecture, instead of the language. In contrast, Microsoft decided to break the compatibility by the release of DirectX10, introducing support for "geometry shaders", and many extensions to the CgFX format.

2.1.3 OpenGL Shading Language

The OpenGL shading language, GLslang, was designed and implemented by 3Dlabs in the end of 2002 [104]. However, the final specification was delayed till 2004, because the OpenGL consortium had to decide if Cg or GLslang should be included in the final OpenGL 2.0 specification [62]. At the end, GLslang made the race – mainly due to architectural purposes – and therefore became the first shading language that has been adopted in a truly cross-platform way by all major graphics hardware vendors.

Similar to most other shading languages, GLslang is based on "C" with some additional syntax from the RenderMan language. As a consequence, the important features are very similar to HLSL/Cg. The main advantage of GLslang, compared to the aforementioned choices, is not one of syntactic finesse, but rather a fundamental difference in the design of not the language itself, but the *process of using it* – RTSL, Cg, and HLSL all translate the source code to some kind of low-level assembler representation "outside" the underlying 3D API, using external tools (RTSL and Cg) or via the API itself (HLSL). Then, the low-level code is translated a second time "inside" the 3D API to the machine code of the actual graphics hardware. In contrast, GLslang compiles directly from high-level shader code to the machine code of the GPU, skipping the intermediate assembler representation. This leads to a significant performance gain, because graphics-driver developers know best how to optimize the generated instructions for their hardware.

2.2 Languages for General-Purpose Computations

All the aforementioned approaches are founded on the *visual paradigm*, i.e. they introduce a new programming language and they are focused on computer graphics. While the first property might be acceptable, the latter is a major limitation for a wider acceptance of GPGPU in industry-driven software development – additional graphics knowledge significantly increases development times and costs. However, for a long time, the aforementioned shading languages were the only way to develop applications for the GPU – but this has changed lately. This section presents GPU-based programming languages that are not focuses on computer graphics, but on general-purpose computations.

2.2.1 Brook for GPUs

Brook was developed at the Stanford University primarily as a programming language for "streaming processors", such as Stanford's Merrimac streaming supercomputer, and the Imagine processor [29]. Buck et al. adapted Brook to the capabilities of computer graphics hardware, making it the first general-purpose language for the GPU [18]. Brook extends the C programming language by *streams* – a "collection of elements" where each element is manipulated by the same computations. Streams are different to "arrays", because there is no index operation and element dependencies are not allowed. The functionality that is applied to each stream element is called *kernel* – which is comparable to a "shader".

Development with Brook is a two-stage process: Once the program is developed, its compilation results in a set of C^{++} files that can be added to the host application. Unfortunately, the target operating system, the target graphics vendor, and the target graphics API have to be specified in advance. Brook is not maintained since the end of 2004, except for some small progress to support ATI's CTM (presented in section 2.2.3).

2.2.2 CUDA – "Compute Unified Device Architecture"

The CUDA environment, introduced by NVIDIA in the end of 2006, is similar to Brook, i.e. the standard C programming language is extended to support streaming types and corresponding operations [89]. In contrast to Brook, the CUDA environment generates full executables (instead of intermediate C^{++} files). CUDA can be used as a *unified environment* to develop applications for both processor platforms – CPU and GPU. Furthermore, this language does not need a graphics back-end, because it accesses NVIDIA graphics hardware directly and therefore supports unique features in GPU-based development, like a full integer and bit instruction set, branching, looping, pointer support, large kernel programs of up to millions of instructions, and thousands of threads.

In addition to the programming language, the CUDA framework also includes libraries for linear algebra (BLAS) and digital signal processing (FFT), that can be used outside the CUDA language. Unfortunately, the software is in an early stage and has not yet been opened for the public, i.e. there is no official distribution and documentation available.

2.2.3 CTM – "Close-to-the-Metal"

At the same time NVIDIA release CUDA, ATI introduced the CTM platform – a data-parallel virtual machine that allows direct communication with ATI graphics devices alongside the graphics API [106]. Similar to the CUDA architecture, many constraints are imposed by this approach, including the ability to read, modify, and write memory in a single program, to directly access host memory, or to cast between formats without explicitly copying the data. CTM is distributed as a library that allows to open, use, and close "managed connections" to one of the three units of the graphics hardware: 1) The "command processor" is programmed via an architecturally independent language, 2) the "data-parallel processor" is programmed via a native (architecturally dependent) instruction set, and 3) the "memory controller" allows direct access to graphics and main memory.

As the name implies, CTM is used to access graphics hardware on a very low level. It has been designed for hand-optimized tuning the GPU-based functionality and not for every-day use. Furthermore, the application is responsible for all problems that occur in programming the graphics processor, which increases development complexity and costs.

2.3 Embedded Languages and Libraries

The aforementioned approaches all are stand-alone programming languages that are used in addition to the general-purpose programming language of the host application (except CUDA). This means that GPGPU developers have to switch between several languages to address the main and the graphics processor. Furthermore, additional code is required to "bind" GPU-based code to the CPU-based host application, e.g. to interchange data.

A different approach is based on the *expression template* technique (which is part of *template meta-programming*)exploits the compiler to pass expressions as function arguments [120, 121]. This usually means that *operator overloading* is used to overwrite the default behavior of the compiler to postpone the evaluation of an expression to a user-defined moment. In other words, instead of performing the actual evaluation of an expression, intermediate code is created that wraps the expression on a higher level of abstraction for later evaluation. The GPU^{++} development system also uses this approach extensively, hence, the technique is described in much more detail in chapter 4.

The idea of using "expression templates" for GPU programming was first mentioned by McCool and Heidrich in the context of *texture shaders* in 1999 [82]. From a technical point-of-view, the presented API was designed as an intermediate layer for shader tree generation, located between the high-level texture shader specification and the low-level rendering API. McCool and Heidrich presented several ideas for the high-level specification, including shader compilers and the "expression template" approach. Based on this work, McCool enhanced the API and created the SMASH library ("Simple Modeling And SHading") [79]. This API was a non-hardware-accelerated testbed for low-level single- and multi-pass graphics concepts. In parallel, the aforementioned concepts were improved and McCool introduced the first version of SH, based on the SMASH testbed, in 2002 [83].

2.3.1 Sh and the RapidMind Development Platform

The Smash testbed – developed at the University of Waterloo – was experimental, not hardware accelerated, and important features were missing, e.g. only a limited subset of OpenGL was supported and no "render-to-texture" functionality was available. This changed in 2004, when Sh addressed real graphics accelerators for the first time [81, 101]. As a result, the Sh language was slightly downgraded, due to the limited instruction set of the available GPUs (e.g. data-dependent control constructs, and missing data types). Note that the Smash API still existed as a thin intermediate layer in the Sh framework, but different targets were addressable using separate back-ends of the Smash layer.

While Sh pioneered embedded GPU programming, it still was graphics-oriented, i.e. it was in-between the visual and the algorithmic workflow of chapter 1 (it addressed a single developer with knowledge in both – graphics and software engineering). Furthermore, the integration into C^{++} has been criticized, due to the massive use of macros and the lack of object-oriented features (e.g. polymorphism and inheritance) [32]. The development of Sh has been stopped in 2006, mainly because it evolved into a commercial product.

The RapidMind Development Platform (RMDP) is the commercial successor of the Sh library that has evolved in two significant ways: It no longer is graphics-oriented and it also supports the Cell Broadband Engine as a target processor [80]. This makes it an attractive development platform for GPGPU programming. Unfortunately, most of the aforementioned C^{++} issues still exist, e.g. the use of macros to embed the language, the compilation of programs cannot be manually controlled, shader programs are not passed as classes – it still is not possible to derive shader programs from existing implementations or to use polymorphism – and the binding of parameters does not conform to common C^{++} argument passing. Furthermore, the developer is responsible for all optimizations – similar to Sh, because the RMDP does not include any strategies for run-time optimization.

2.3.2 PeakStream Platform

A comparable commercial product is the PeakStream Platform that also uses the "expression template" approach for embedded programming of the GPU [96]. However, in contrast to other solutions, graphics hardware is optional, as long as the main processor supports the streaming extension command sets SSE2/SSE3. Unfortunately, the software is not yet made public and therefore no further information is available.

The PeakStream Platform currently works on Linux and extends the GCC or Intel compilers. Such a coupling is required, because this software introduces own compiler `pragmas` and additional debugging information. Consequently, debugging and profiling the GPU-based code is fully supported. Furthermore, new types are natively built-in, which actually makes the PeakStream Platform an own programming language (section 2.2). The used programming model is based on "data parallelism" similar to the SIMD approach of ISL in section 2.1.1, i.e. special arrays (`Arrayf32` and `Arrayf64`) are combined with high-level operations to create new arrays. As a consequence, there is no explicit *kernel synthesis*.

2.3.3 Accelerator

A similar programming model is used in the ACCELERATOR library, which is the second programming library for graphics hardware from MICROSOFT – beside the graphics-oriented DIRECTX HLSL [116]. While ACCELERATOR is distributed as a library and does not actually use the "expression template" technique to embed into a host language, it integrates into most of the languages available as part of the .NET framework (e.g. C#). The common .NET just-in-time compilation generates CPU- or GPU-based code at run-time.

Similar to the PEAKSTREAM PLATFORM, ACCELERATOR does not support *kernel synthesis*, but uses special array types (so-called "data-parallel arrays") for GPU-based computations. Unfortunately, data-parallel arrays are different to normal memory arrays, because they lack of specific features (e.g. aliasing, pointer arithmetic, and the access of individual array elements). Therefore, a time-consuming explicit conversion is required each time the developer wants to share data between CPU and GPU. Currently, the ACCELERATOR library is a research project and does not allow to be used in commercial development.

2.4 Comparison Chart

Table 2.1 gives a detailed overview of the aforementioned approaches and the features they support – including architectural features, portability, and the used programming model. The more features the language supports, the better the GPGPU software development.

Features	CG	GLSL	BROOK	CUDA	CTM	SH	RMDP	ACC.	GPU^{++}
General-Purpose	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Embedded Lang.	No	No	No	No	No	Yes	Yes	n/a	Yes
Object-Oriented	n/a	n/a	No	No	No	No	No	No	Yes
Cross-OS	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes
Cross-GPU	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Direct Access	No	No	No	Yes	Yes	No	No	No	No
Kernel Synthesis	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Array Data-Type	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Optimization	Yes	Yes	Yes	Yes	n/a	No	No	No	Yes
Non-Commercial	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Active Develop.	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes

Table 2.1: Comparison of the presented GPU-based development systems (and GPU^{++}). PEAKSTREAM is not included, due to the lack of available information. *Object-Oriented* means, that inheritance and polymorphism work for kernel programs, and *Kernel Synthesis* means that kernel programs are specified explicitly (i.e. not using "data-parallel arrays").

Chapter 3

Concepts

The Gpu^{++} development system has been designed to simplify the implementation of general-purpose computations on the GPU. As shown in chapter 1, this also includes the abstraction of GPU-specific techniques, terms and patterns. Otherwise, despite a nice programming interface it would still be required to be an expert in computer graphics and graphics hardware to implement non-graphical algorithms on the GPU. In other words, "the fewer technical details about the GPU and computer graphics, the better for GPGPU".

But this cannot be done to the maximum, due to the differences that exist between GPU and CPU. Such differences are the main reason that some algorithms run much faster on the graphics hardware than on the main processor. A full abstraction of the GPU therefore inevitably leads to a theoretical construct, a proof-of-concept without a practical value. Therefore, the Gpu^{++} development system has not been designed to hide all differences between the CPU and the GPU architecture, but to abstract the most important features by generic concepts that common C^{++} developers are able to learn in a short time.

This chapter is about the five concepts that are relevant for GPU-based development using the Gpu^{++} system. Each of these concepts falls into one of the following categories:

Fundamental – Only two of the concepts are fundamental – the *application life-cycle* of section 3.1 and the *streaming paradigm* of section 3.2. Developers have to be familiar with these concepts, because they significantly affect the design of Gpu^{++} programs.

Optional / Optimization – The other concepts are optional – the *vector processor* in section 3.3, *computation frequencies* in section 3.4 and *array access* in section 3.5. In fact, developers are able to implement algorithms in Gpu^{++} without further knowledge of these concepts. While this leads to adequate results, there is significant potential for further optimizations. To create the most efficient code with the Gpu^{++} system it is inevitable for the developer to be familiar with all of these concepts.

The remaining section 3.6 clarifies the consequences of the aforementioned concepts by presenting a list of GPU-based requirements. Even though the reader might be familiar with programming concepts for the GPU, it is strongly recommended to read the sections of this chapter, due to new terms and definitions that are used in the Gpu^{++} system.

3.1 Application Life-Cycle

The life-cycle of a common application can roughly be divided into four stages – *design*, *implementation*, *compile-time*, and *run-time*. While the first and second stages both are independent of the target environment, the third and fourth stages are platform-specific, i.e. the compiler generates an executable for a dedicated target that can only be executed in this environment.[1] In the context of C^{++} applications, the CPU is "known" at *compile-time* and it is "used" at *run-time*.

But if also graphics hardware is used, there are actually two processors – the CPU and the GPU. While most parts of the application are compiled for and executed on the CPU, other parts are compiled for and executed on the GPU. Consequently, there are two compile-time stages and two run-time stages – one for the main processor and one for the graphics processor. The situation becomes even more challenging, because there is no unified GPU architecture, i.e. there is no unified GPU instruction set. In fact, each family of GPUs uses its own proprietary instruction format – which is encapsulated in the graphics library. This leads to the obscure situation that there is no way to fully compile a GPU-based program "in advance", because the target graphics hardware is unknown.

While multiple compile-time and run-time stages might be advantageous for the different developers of a *visual workflow*, they significantly hinder the single-developer *algorithmic workflow* (see chapter 1). There are two reasons that may lead to confusions – the absence of a clear nomenclature and the overlapping of the different processor stages. For instance, it *sounds* and it *feels* confusing that "a compile-time constant of the GPU can be a run-time variable of the CPU". To avoid irritation, the following definitions of the life-cycle stages are used (excluding the processor-independent "design" and "implementation") – figure 3.1 presents a graphical representation:

CPU-based compile-time – The full application is compiled to the specific CPU architecture, including "intermediate code" for embedded GPU processing. While CPU-based parts are optimized, no optimizations are applied to the GPU-based routines.

CPU-based run-time – The created executable is launched on the target mmain processor. While the CPU-based code cannot be changed anymore, the behavior of the GPU-based routines can be effected using CPU-based run-time variables.

GPU-based compile-time – This stage is performed at CPU-based run-time, when GPU-based code is processed for the first time. In this stage the GPU code is created, optimized, translated and uploaded to the available graphics processing unit.

GPU-based run-time – This stage is also performed at CPU-based run-time, when a specific GPU routine is executed. It launches the GPU-based code (that has been created and uploaded before), which then performs all GPU-based computations.

[1]The *target environment* is dependent of the programming language – e.g. Java generates code for a virtual machine, while the target environment for C^{++} programs usually is a specific CPU. For the sake of simplicity, the term "CPU" is used as a synonym for all target environments in the remainder of this work.

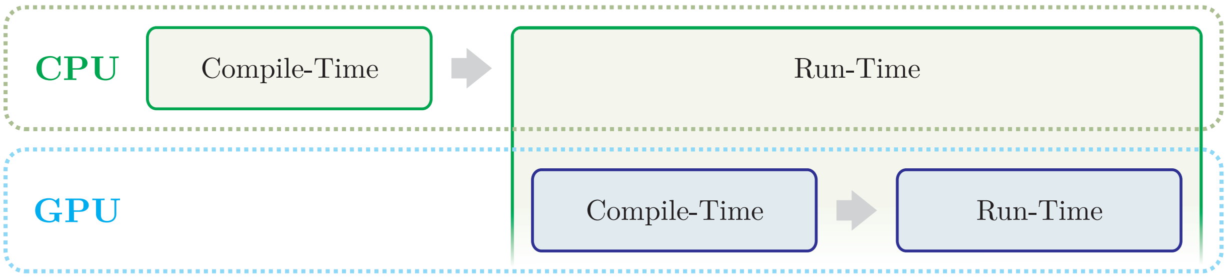


Figure 3.1: The life-cycle stages of an application that works on both – CPU and GPU. The first two stages (*design* and *implementation*) are not shown, due to their hardware independence. Note that *compile-time* and *run-time* overlap for the different processors.

3.2 Streaming Paradigm

Graphics hardware has impressively evolved over the years, as shown in figure 1.1. This leads to the question, why all the optimizations on the graphics processor cannot also be done on the main processor. Where is the main difference between the CPU and the GPU?

Computations on the GPU can be *parallelized* much more efficient than CPU-based programs. In fact, state-of-the-art graphics hardware contains more than 100 unified computation units [90]. Graphics hardware initially was designed to accelerate the color computation of pixels on the screen. For common visual algorithms, like texture mapping or local illumination, such computations are independent from other pixels, i.e. a pixel can be computed without the (intermediate) results of other pixel computations. As a matter of fact, this independence of data allows the graphics hardware to compute a large set of pixels simultaneously – current GPUs are able to compute up to 24 pixels in parallel [90].

Such a parallelization scheme is known as *stream processing*, which means in general: given a set of input data (the so-called *stream*), a series of compute-intensive operations (the so-called *kernel*) is applied to each element of the stream. On the graphics hardware the stream is the “vertex list” or “input texture”, and the kernel is known as “shader” [122]. Stream processing has been known in scientific computing before – a very popular streaming approach is the IMAGINE stream architecture developed at the Stanford University [103]. Furthermore, the streaming paradigm slowly finds its way to the design of next generation CPUs, as can be seen for the latest CELL and ITANIUM microprocessors [52, 84].

This concept is crucial, because it leads to most of the performance-boost of the GPU compared to the CPU. As a consequence, the streaming paradigm is fundamental in the GPU^{++} development system. However, because the term “stream” may be misunderstood as a one-dimensional data container, the following definitions are used:

Array – Arrays are equivalent to “streams” – they contain a set of elements. However, developers might be confused by this term, because streams are usually limited to one-dimensional data sets, while arrays can be also of higher dimensions. Furthermore, reading from such containers looks and feels like reading from memory arrays.

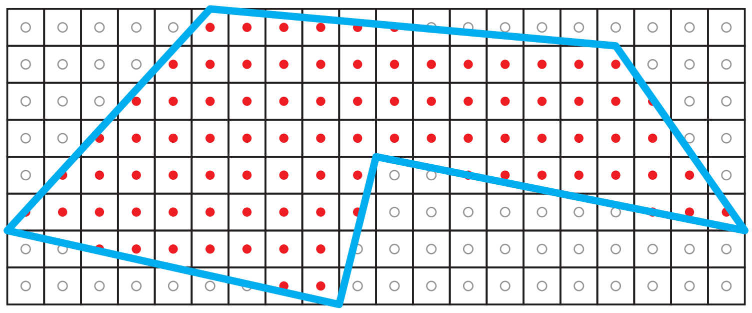


Figure 3.2: The *region* (blue) specifies the elements of an *array* (red dotted squares) that are computed by a *kernel*. Elements outside the region (gray circles) are not touched.

Kernel – A kernel is a series of commands that compute a value, which is then written to an output array. The computation may depend on the individual elements of several input arrays. Note that a kernel program is executed *on* an output array, i.e. the specified kernel is applied to *each* element of the array it has been executed on.

Region – A region confines the elements of an array, i.e. it is a “mask” that determines the elements that are computed by the given kernel. Regions also can be used for input arrays, to specify the array elements that should be used as inputs for the kernel.

Figure 3.2 illustrates, how a kernel is applied to some array elements that were specified by a region. The default case applies the kernel program to *all* elements of an array.

3.3 Vector Processor

Computer graphics is about *vectors*. The processing of geometries, the computing of colors, such computation heavily involves vectors – in the three-dimensional space, on the two-dimensional screen, by addressing one-, two- or three-dimensional textures, computing three- or four-dimensional colors (with transparency as the fourth component). Hardware that accelerates graphics computations has to address this special algebra.

As a consequence, a major difference between GPU and CPU is its “data scalarity”, i.e. while a single instruction on the CPU usually manipulates a single data value, a single instruction of the GPU usually acts on multiple data entities simultaneously. This technique is classified as *single instruction, multiple data* (SIMD), while the CPU instruction set is classified as *single instruction, single data* (SISD) [37].[2] Processor architectures based on the SIMD approach – so-called *vector processors* – are known in scientific computing since the 1960s. The GPU is a very small vector processor, because an instruction is performed on four scalars at most. In contrast, the CONNECTION MACHINE of the early 1980s computed hundreds and thousands of scalars with a single instruction [118].

[2]Today almost all CPU designs include SIMD extensions, e.g. Intel’s MMX, SSE, and SSEx, AMD’s 3DNow, or IBM’s AltiVec. However, the core processing of these CPUs is still done via SISD commands.

In other words, for a main and a graphics processor with the same core and memory speed, a dedicated algorithm theoretically runs up to four times faster on the GPU due to its vector architecture. While this is a rare case that cannot be generalized, many algorithms can be restructured to make partial use of the SIMD approach on the GPU.

This concept is optional, because GPU^{++} tries to find an optimal assignment of the vector components for a given kernel program using the *vector fusion* approach of section 6.2.1. It does not make a difference, if the developer uses the vector processor abilities of the system or not – internally, each vector component is treated as a separate equation that is merged with the other equations in a final stage. This results in acceptable performance. However, there are three reasons why the "vector processor" concept is recommended in GPU development anyway: 1) the vector processing paradigm may change the design of the kernel program, which leads to an implementation that can be optimized more efficiently, 2) to reduce GPU-based compile-time, a heuristic is used for merging vector instructions, which works best if the developer gives some "hints", and 3) the implementation of a kernel might be easier to understand and to maintain when using the vector processor concept.

It should be noted that there is an alternative way to interpret the GPU as a vector processor: On a higher level of abstraction, a single call of a graphics API function invokes the recomputation of the complete frame-buffer [97]. However, this thesis uses the term "vector processor" to describe the small scale parallelism of the most basic GPU vector type of today's graphics hardware – the four-component vector. The aforementioned large scale parallelism has been explained in section 3.2 using the "streaming paradigm".

3.4 Computation Frequencies

A *computation frequency* indicates "how often" a computation is performed in a larger context – the more often a task is done, the higher its computation frequency. This term usually is used in combination with the "direction" of computations, where the output of a *single* low-frequency computation is used as input in *multiple* high-frequency computations.

Developers are accustomed to this concept in their everyday work, e.g. using *precalculations*, where a complex sub-expression is computed once to be reused many times in the actual algorithm. While computations at lower frequencies might take more time, they also reduce the execution time of higher frequencies that are performed more often. Hence, computation frequencies may lead to optimized time and memory requirements for the algorithm, e.g. computations done at compile-time are available for free at run-time.

The paradigm of computation frequencies has been introduced to graphics hardware and shading languages by Proudfoot et al. as part of STANFORD's REAL-TIME SHADING LANGUAGE [99]. A typical dependency-chain of computation frequencies in computer graphics is: SCENE → OBJECT → $\underline{\text{VERTEX}}$ → $\underline{\text{GEOMETRY}}$ → EDGE → SCANLINE → $\underline{\text{PIXEL}}$.[3] Note that there might be less or more computation frequencies, depending on the context.

[3]Underlined frequencies are programmable on today's GPUs: *vertex* is programmable since 2001, *pixel* became programmable in 2002, and *geometry* programming started in 2006 with the NV80 chip set [90].

The context of the aforementioned frequency-chain is computer graphics, which seems to be inappropriate for general-purpose computations. As a consequence, the GPU^{++} development system defines its own chain of the following four computation frequencies:[4]

Compile – Expressions that can be evaluated at GPU-based compile-time belong to the COMPILE computation frequency. The inputs are taken from CPU-based run-time (i.e. constants and variables that have been evaluated by the CPU-based parts of the application) and the result of this computation is passed to following frequencies.

Execute – Expressions that can be evaluated *before* the GPU program is executed fall into the EXECUTE computation frequency. The input arguments can change between distinct executes, but they stay constant for all array elements. This includes all CPU-based constants and variables, as well as results from the COMPILE frequency.

Region – The REGION computation frequency evaluates all expressions that depend on the passed regions, e.g. any linear expression that uses the array element position as its input. As before, the inputs are taken from lower frequencies (i.e. COMPILE and EXECUTE), while the expression result is passed to the last frequency.

Element – Finally, the ELEMENT computation frequency evaluates the remaining expressions. Input arguments are the results of previous computation frequencies, as well as elements of input arrays. The result of these expressions is the final result of the overall computation, and therefore, it is stored in the elements of the output array.

The concept of "computation frequencies" is optional, because many algorithms can be implemented on the GPU without further knowledge if this paradigm. However, especially the use of "type qualifiers" (section 4.2.3) requires a brief understanding of this concept.

3.5 Array Access

Similar to CPU-based arrays, where an *index* addresses a specific element, GPU-based arrays can be read within a kernel program by using a *position*. However, while the index for CPU-based arrays is an *integral*, the GPU-based position is a *floating-point value*. This leads to the following two differences: First, each array dimension is scaled to the range $[0 \dots 1]$, e.g. the fifth element of a one-dimensional array of size 20 is accessed by using position $5/20 = 0.25$. Second, it is possible to access an *interpolated element* that lies "in-between" two array elements, e.g. position 0.251 results in a different values than 0.249.

By scaling the array dimensions to the range $[0 \dots 1]$ and using interpolation for "in-between elements", the kernel program has been made independent of the actual array size. In fact, GPU-based arrays are abstracted to *continuous* functions that are defined for the

[4]Note that these computation frequencies are defined for kernel programs only. As mentioned before, there are additional CPU-based computation frequencies that are not included in the above list.

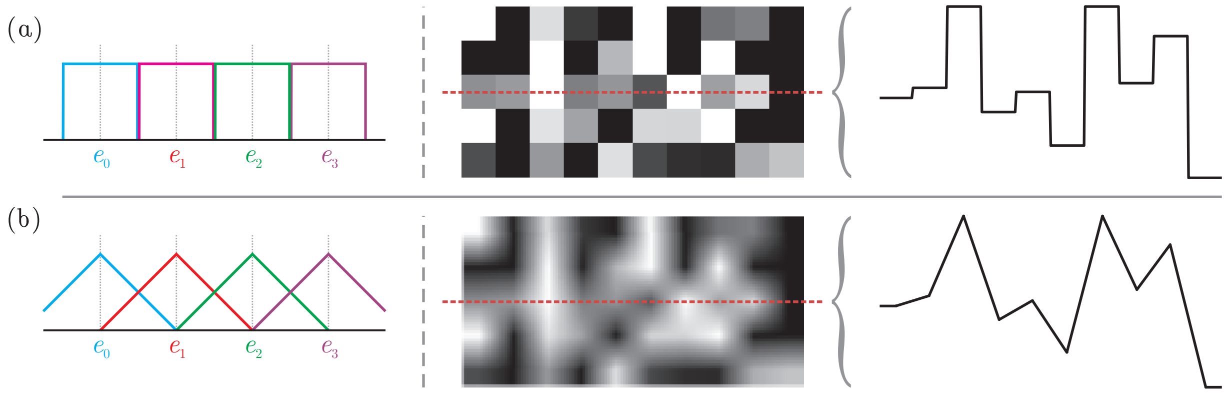


Figure 3.3: Available interpolation schemes to read array elements within a kernel program: a) *nearest neighbor* and b) *linear interpolation*. The left illustrates the weighting functions for neighboring elements. The same 10×5 array is shown in the middle for the different interpolation schemes. The right shows the the intensity values of the center row as a graph.

unit range. This forces the design of kernel programs to be generic, i.e. kernels can be used with arrays of arbitrary resolutions. However, this concept leads to two issues:

Elements outside the range – Developers can choose between *clamping* the position to the range $[0 \ldots 1]$ (i.e. positions larger than the upper bound are set to 1.0 and positions smaller than the lower bound are set to 0.0) or infinitely *repeating* the range (i.e. only the fraction of the position is used).

Interpolation scheme – Developers can choose between *nearest neighbor* that returns the element that is nearest to the given position, or *linear interpolation* that computes a new value by linearly interpolating the array elements (e.g. if position p lies 40% on the way between element e_1 and element e_2, the value $e_p = 0.6e_1 + 0.4e_2$ is returned). See figure 3.3 for a comparison of the schemes.

Please note that these issues are relevant only for reading from an array. In contrast, writing array elements is always done *within* the unit range and WITHOUT applying interpolation.

It might be useful to access the *same* array in the *same* kernel with *different* read-access properties. As a consequence, the properties are stored in an object separately from the array, which is called the *sampler*. Multiple samplers can be associated to the same array, where each sampler may have different strategies for handling the range and interpolation. As a matter of fact, an array is always accessed *through* a sampler object within the kernel.

Please note that the linear interpolation scheme can be exploited for memory access optimizations, because a single read command returns the linear combination of two array elements. This approach is noticeably faster than performing two distinct reads of neighboring pixels (without interpolation), due to its native support in graphics hardware. There are numerous general-purpose computations, where this technique leads to a noticeable increase of speed-performance – examples can be found in sections 7.1.2 and 7.1.3.

3.6 Requirements

While there are plenty of algorithms that benefit from their implementation on the graphics processing unit, others may not. The algorithm and its computational workflow need to fulfill the architectural qualifications of the graphics hardware to result in a significant performance gain. Some of the requirements are rooted in the architectural design of the graphics hardware, while others are founded in the GPU^{++} development system.

Ideally, all GPU-based requirements are accomplished by the GPU-based implementation, however, some prerequisites can be evaded, i.e. algorithms often can be redefined to cooperate with the aforementioned concepts. For instance, the classic sorting algorithms (like QUICKSORT) are usually inappropriate to be implemented on the GPU, but stream-friendly sorting perfectly fits into the algorithmic requirements – see chapter 7.3.

Conceptional Requirements

The following list of requirements directly derives from the aforementioned fundamental concepts, i.e. the stream paradigm and vector processing. Relaxing such requirements would lead to a (partial) architectural redesign of the graphics processing unit:

Non-Sequential Output / Scattering – This means that *all* elements of the output array are computed (except the elements not covered by the output region) – there is no easy way to specify an “output modulo” that forces GPU^{++} to compute only every second or third output element. However, such a static pattern can usually be avoided by restructuring the algorithm or using regions. Far more problematic is the data-dependent case, where the kernel result is written to an output position that depends on the result itself (this is called *scattering*). For instance, when computing a histogram the element spectrum is divided into buckets and the algorithm computes the element count for each bucket. The sequential algorithm goes through all n elements e_i and increases the value at output position $Q[e_i]$ via scattering (where Q is the quantization of the input spectrum). Unfortunately, there is no efficient implementation for histogram computations on graphics hardware [88, 36].

Limited Element Dependencies – Computations that rely on previous computations may cause a massive performance loss. As a rule-of-thumb it can be said that: “the more dependencies, the slower the implementation”. For instance, imagine the *all-prefix-sums* algorithm that computes the sum of all previous array elements, i.e. the first output element in a row is $r_0 = 0$, while the second is $r_1 = 0 + r_0$, and the nth output element is $r_n = r_0 + r_1 + \ldots + r_{n-1}$. The sequential implementation uses a temporary variable $t = 0$ that is raised by r_i for all elements i in a row, but such variables are not available on the GPU. While there are GPU-based implementations for similar algorithms, CPU-based solutions usually run noticeably faster [53]. However, there are a variety of techniques to overcome this restriction, e.g. *reductions* or *loop swapping*, where inner loops are flipped with outer loops – see section 7.2 [18].

Data-Dependent Output Size – This means that the output size of an algorithm depends on its input arguments. In fact, the size of the output is specified in advance by creating the output array. Executing the kernel program on a given array will *not* change its size, therefore, it cannot depend on the input content. For instance, imagine an algorithm that collects all n input values above a certain threshold, i.e. the output size varies from zero (all elements below the threshold) to n (all elements above the threshold). This problem is usually solved by introducing *empty numbers* that are returned instead of "nothing" [92]. Please note that the introduction of *geometry shaders* eliminates this restriction for future GPUs [90].

Data-Dependent Workflow – This is similar to the previous property, but means that different code is executed, dependent of the input content. While this is awkward within a single kernel program, it becomes almost impossible when spanning over multiple kernels, i.e. the output of kernel A effects the code that is executed in kernel B (e.g. different array elements are read). For instance, most sequential sorting algorithms fall into this category, because the result of a comparison effects the input that is used in the next comparison (which repeats over and over) – see section 7.3.

Hardware Requirements

The upcoming list of requirements is rooted in practical considerations of the graphics hardware vendors, such as the reduction of production costs. Relaxing such requirements only leads to small changes in the architectural design of the graphics processing unit:

Data Size – One of the major differences between the CPU and GPU is the amount of memory that they can access. The benchmark system used for the performance evaluation in chapter 7 is equipped with 16 GiBytes of main memory and 768 MiBytes of graphics memory. Furthermore, the arrays can have a size of at most 8192 elements for each dimension. Both restrictions hinder the use of graphics hardware for processing large data, like in the medical domain, where a medical data set with a size of 512^3 single-precision floating point intensity values requires 512 MiBytes of memory – and "processing" usually means that there are at least two arrays (one for the input and one for the output). However, such restrictions can be avoided via *bricking*, where the data set is divided into subsets of moderate size [68].

Precision – Computational precision is a crucial issue in scientific research and there is a vast number of algorithms that require special care in this context. Current graphics hardware supports floating point values of up to 32 bits, which is equivalent to the C^{++} type `float`. However, this does not automatically mean that all computations are done with this precision. In fact, there are operations that might be performed with less accuracy, like trigonometric functions – but this differs between GPUs. Tt is recommended to create precomputed look-up-tables if high precision is crucial, but this might drastically decrease performance. In addition, there are techniques that virtually extend the precision of floating-point values to 64 or 128 bits [117].

GPU++-Based Requirements

The following list contains requirements that derive from the software architecture of the GPU^{++} development system. While some GPU-based features can be used in low-level or high-level shading languages, GPU^{++} hides them to simplify its programming model:

Basic Run-Time Flow-Control – There are several flow-control approaches used in today's graphics hardware: The most basic approach for conditions is *predication*, where both sides of the branch are evaluated and then one of the results is discarded based on the conditional boolean value [93]. In *MIMD branching*, different computation units can follow different paths through the kernel program. Finally, in *SIMD branching*, all active computation units must execute the same instruction at the same time, resulting in a hybrid solution of *predication* and *MIMD branching*. The GPU^{++} development system does not support MIMD or SIMD branching, because the conditional C^{++} statements (e.g. `if` and `while`) cannot be overridden. But *predication* is supported by the commands `min()`, `max()`, `select()` and `threshold()`.

No Graphics-Dependent Features – While computer graphics hardware was designed to accelerate computer graphics computations, the GPU^{++} development system abstracts this graphical paradigms for the sake of generality. As a result, some of the graphics-dependent features cannot be addressed via GPU^{++}, like the z-Buffer, the early z-Test and occlusion queries [12]. GPU-based implementations that require such features may run significantly slower with the GPU^{++} development system.

There are numerous other restrictions, e.g. amount of temporary registers that can be used in a kernell program, maximum size of shader programs, the amount of textures that can be accessed by a kernel program, and the maximum number of environment variables. However, in contrast to the aforementioned restrictions, such limitations change radically between generations of graphics hardware and will become obsolete in the near future.

Chapter 4

The Expression Graph

Many of the GPU-based programming languages that have been presented in chapter 2 (e.g. GLslang, Brook and Cg) are "string-based", i.e. the kernel program definition is specified in a text-string or file that is transformed into an internal representation using a dedicated compiler. Because such programming systems fully control the complete compilation process, the internal data structure is usually designed to integrate all language- and hardware-specific features and enables fast processing during compilation, e.g. for optimizing the GPU-based code. However, string-based languages slow down the development of GPGPU applications, due to the separation of CPU- and GPU-based source code.

A different approach is the "embedded definition" of the kernel program, like in Sh and the RapidMind Development Platform, where the C^{++} language is "extended" to create the internal data structure. While there is no theoretical limit in the design of the internal data structure, the aforementioned embedded approaches use compile-time-efficient data structures that lack support for on-the-fly optimizations. Other "embedded approaches", like the Accelerator library or the PeakStream software, apply algebraic operations on complete arrays, avoiding explicit *kernel synthesis* [96]. While this is an intuitive scheme, it does not allow to exploit the full power of today's GPUs.

The internal data structure of the Gpu^{++} development system was designed to combine the aforementioned advantages of an ***embedded***, ***graphics-independent*** and ***explicit*** kernel program definition that supports ***on-the-fly optimizations*** and ***exploits all computation stages*** of the graphics processing unit.

This chapter is about the resulting internal data structure, which is a *directed acyclic graph* that represents the complete kernel program. The novel features of this data structure are described in detail in section 4.1. However, because the kernel program is *embedded* in the normal C^{++} source code of the host-application, the creation of the data structure is of special interest and is extensively explained in section 4.2. In contrast to other GPU-based programming approaches, several advanced optimization strategies are performed on the internal data structure – which requires a very efficient processing infrastructure for the graph: thus, section 4.3 presents novel improvements of the well-known *visitor pattern*.

4.1 Internal Representation

In computer science, it is common to represent mathematical expressions, and even complete computer programs, by using a graph. The most simple graph type that can be used is a tree, where all the leaf nodes represent *constant values*, while all the inner nodes represent *operations* that are performed on these constant values. Figure 4.1a shows a simple expression tree. By recursively traversing the tree, from the leaf nodes to the root node, each inner node can be replaced by the result of the operation on its child nodes, until there is only one node left – which represents the result (4.1b-e). This is called *evaluation*.

In 1984, Robert Cook was the first who used trees to represent shading equations during rendering – since then, most shading languages have used similar data structures for internal shader representation [24]. However, while a tree can be used to represent complete programs, it is a sub-optimal for programs that generate intermediate results and store them in temporary variables – "temporaries" cannot be represented by a tree.

More generally, a graph that represents a complex expression (i.e. a complete program) needs to be a *directed acyclic graph* (DAG) – for any vertex v, there are no non-empty directed paths beginning and ending on v [27]. This definition allows temporary variables, but also avoids "circularities", i.e. expressions cannot use their own result as input. In fact, a directed acyclic graph is a generalization of a tree, where subtrees can be shared in a global context, which means that partial results are used as input more than once in the expression (the root nodes of such subtrees are the aforementioned temporary variables).

Most of the existing GPU-based shading languages use a DAG (or similar data structures) as their internal representation of the kernel program (except SH and the RAPIDMIND architecture that both use a treelike structure with "a list of tokens" at each inner node [101]). However, the DAG used in the GPU^{++} system differs by two unique and novel concepts – *vector decoupling* and the *unified kernel definition* – that will be explained in sections 4.1.1 and 4.1.2. Furthermore, while features like "embedded creation" (section 4.2) and automatic "type qualifiers" (section 4.2.3) have been introduced separately by other GPU-based languages, only the GPU^{++} development system contains them all.

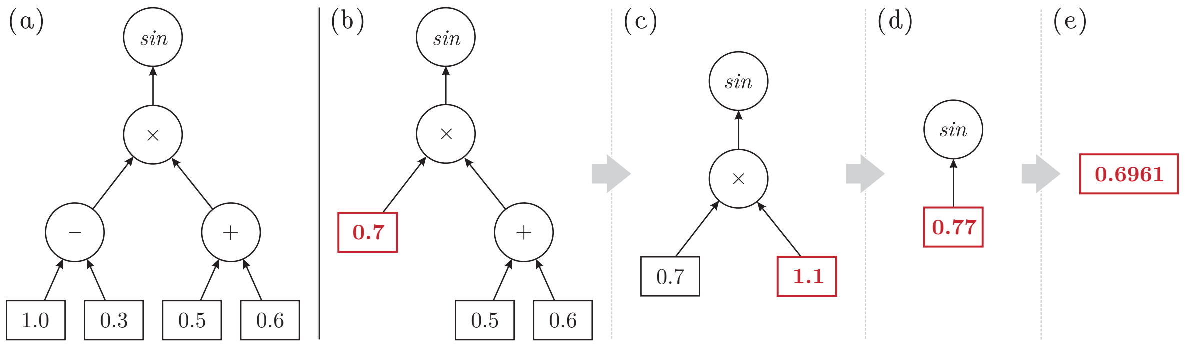


Figure 4.1: The mathematical expression "$sin((1.0 - 0.3)(0.5 + 0.6))$" represented by a tree (a), and the stepwise evaluation of the expression using a bottom-to-top traversal (b)-(e).

4.1.1 Vector Decoupling

The expression graphs of other GPU-based programming languages usually map the underlying graphics hardware architecture directly to the graph nodes, i.e. that an addition of two three-component vectors is represented as a single `add` node with two three-component vectors as its inputs. While this significantly simplifies the compilation processes, it has two major drawbacks: 1) Developer have to be experienced in using a *vector processor*, and 2) there is no automatic combination of individual operations – or it works sub-optimal.

In contrast, the GPU^{++} development system decouples the component-wise operations in the expression DAG, e.g. the aforementioned sum of two three-component vectors is translated to three individual `add` nodes, each adding a single component of the first vector with the corresponding component of the second vector. Even further, vector operations like `cross()` result in three nodes, each computing a single component of the result vector. This novel *vector decoupling* approach enables optimizations on a significantly finer granulation – optimizations can be applied to single vector components instead of complete vectors. Furthermore, it does not make any difference whether the developer is familiar to a *vector processor* or not, because the expression DAG only represents scalar operations.

This novel approach only works in combination with an efficient *vector fusion* algorithm that reverses the vector decoupling at the end of the optimization stage – while *vector decoupling* is trivial, the *vector fusion* of individual instructions is an advanced task. The GPU^{++} system uses an efficient approach that is explained in section 6.2.1.

4.1.2 Unified Kernel Definition

The expression DAG in the GPU^{++} development system contains expressions for all computation stages of the GPU, i.e. the represented kernel program may span over several computation frequencies (like the vertex and the fragment unit).

This is significantly different to most of the other GPU-based programming approaches, where the kernel program is explicitly bound to a specific computation frequency. For instance, the shader code in SH is either associated to “`gpu:vertex`” or “`gpu:fragment`” and GLSLANG and CG work similar. Other GPU-based programming languages (like BROOK, PEAKSTREAM, and ACCELERATOR) make only use of the graphics hardware’s fragment processing stage. Both approaches lead to sup-optimal results: Either the computational power of the GPU is not fully exploited, or the developer is forced to split his GPU-based code into parts for fragment and vertex processing, which is nontrivial in many cases.

In contrast, the *unified kernel definition* of the GPU^{++} development system abstracts the low-level graphics hardware architecture by hiding the specific computation stages. Instead to burden the developer with the decision of which kernel operation is executed on which GPU computation stage, the expression DAG is automatically partitioned during the optimization stage – see section 5.2.2. As a consequence, all relevant GPU-based code is clustered in a single kernel program, resulting in significantly reduced code complexity.

4.2 Embedded Creation

As mentioned before, the internal kernel representation is usually created using a dedicated parser. Similarly, the GPU^{++} development system also uses a parser: the one from the C^{++} compiler. This is achieved by *embedding* own language constructs into the C^{++} language.

Note, that the word "embedding" is used in the meaning of "seamlessly integrating", hence, "embedded expression graph creation" actually means "the creation of the expression graph is seamlessly integrated into the C^{++} language". The developer does not see a difference between writing expressions that will be computed by the CPU and writing expressions that will be processed on the GPU. Unfortunately, the compiler does not know the GPU nor the DAG structure. To exploit its parser the following detour is taken:

> *While the C^{++} compiler cannot be used directly to create the expression graph, it can be used to create intermediate code that will creates the graph when executed.*

Ad-hoc polymorphism[1] is used to create such "intermediate DAG creation code", i.e. new class types are introduced and *operator overloading* is used to change the implementation of arithmetic operations that are applied to these types (such as "+", "*", and "<") [111, 112]. In fact, this technique is "syntactic sugar", i.e. it is an addition to the C^{++} syntax that does not affect its expressiveness – but makes it "sweeter" to use: operator overloading allows a familiar notation, where custom data types look like native built-in types.

For instance, consider a simple system that stores values as strings in the class `value`. Furthermore, consider the following overloading of the sum operator ("`operator +`"):

```cpp
value operator + (value const & a, value const & b)
{
   std::strstream out;                                          // use string-stream
   out << "(" << a.get_string() << "+" << b.get_string() << ")";     // concatenate
   return value(out.str(), out.tellp());                // create output from string
}
```

While the C^{++} expression "`value r = value(3) + value(4);`" looks like a typical algebraic equation to the software programmer, the resulting instance `r` encapsulates the string "`(3+4)`" instead of the string "`7`". In fact, the expression was not "evaluated" at compile-time, but intermediate code was created that generates the resulting string at run-time.

This approach is common to create internal data representations in many domains, such as in *compiler design* or *linear algebra* [86, 65]. While the technique was adopted to computer graphics and shading languages by McCool and Qin as part of the SH language, it has not been used to create an expression DAG but a plain instruction list [83, 81, 101].

[1] There are three types of polymorphism in the C^{++} language: First, *static polymorphism* is used in templates that are created at compile-time. Second, *dynamic polymorphism* is based on inheritance at run-time, where a function call may differ dependent on the derived classes. Finally, *ad-hoc polymorphism* is used to execute different functionality for different input argument types – usually done via *overloading*.

4.2.1 Custom Data Types

From a conceptual viewpoint, the new custom data types represent the data primitives available on the GPU, like vectors (section 4.2.1) and matrices (section 4.2.1). In practice however, the custom data types encapsulate DAG nodes and the overloaded operators connect such nodes together. Therefore, whenever the developer applies operations on the following custom data types, the expression DAG is "silently" created in the background.

Vectors

The major data type natively supported by the GPU is a *vector* of up to four *vector components* (or *scalars*) that usually represents colors, positions in space, and texture coordinates (see section 3.3). As a consequence, the major class types of the GPU^{++} system are the vector classes "`vec[1|2|3|4]`" (i.e. "`vec1`" represents a one-component vector, "`vec2`" is a two-component vector, etc.).[2] Operators have been overloaded as follows:

```
class vec2 {
   vec2   operator - () const;                               // unary subtraction
   vec2   operator + (vec2 const & v2) const;                  // binary addition
   vec2   operator * (vec2 const & v2) const;            // binary multiplication
   ...                        // same for unary +, and binary -, /, +=, -=, *=, /=
};
```

Please remember that the implementation of an overloaded operator *does not* compute the result of the operation, but inserts a new node to the expression DAG that represents the operation together with its operands. The mathematical operators (i.e. "`+`", "`-`", "`*`", "`/`", "`+=`", "`-=`", "`*=`", and "`/=`") are overloaded for all reasonable vectors and vector combinations. Additionally, the assignment operator "`=`" duplicates all components of a vector. A complete list of the overloaded operators and the supported constructors can be found in the language reference in appendix A.

Note that, until now, all mentioned mathematical operations are performed *component-wise*, which is plausible for addition and subtraction, but might be confusing for multiplication and especially for division. For instance, the component-wise multiplication of vectors $v = (2, -4, 3)$ and $u = (-1, 2, 3)$ leads to $u \times v = (-2, -8, 9)$.

In addition to such component-wise operations, *vector operations* are also supported. However, vector operations are not implemented via operator overloading, but using function calls, e.g. "`dot()`" performs a vector dot product and "`cross()`" computes the cross product of two three-component vectors. The function reference in appendix A.4 gives an overview of all function call operations available in the GPU^{++} development system.

[2]For the sake of simplicity, only scalars of type "`float`" (*32-bit floating point*) are supported. This is not a limitation, because on current and near-future GPUs, computations are only performed in this precision. However, GPU^{++} can easily be extended to support additional scalar types, like "`int`" and "`bool`".

A very popular GPU-based scheme to address vector components is *swizzling*, which stands for the extraction, rearrangement, and optional duplication of vector components. This operation can be used without speed-costs on most graphics processors (or at least it is very "cheap"). Therefore, it is heavily used in GPU-based programming. While swizzling might lead to more efficient programs, it may also result in code that is hard to understand, test, and maintain. Nevertheless, swizzling is an excellent optimization technique (when used efficiently) and therefore is supported by the GPU^{++} development system.

Because swizzling is not natively supported by the C^{++} language and non-graphics developers may be unfamiliar with the concept, its language integration has to be "compatible" to the C^{++} specification. In fact, this is not the case for the existing syntaxes:

Cg, GLslang – Swizzling is indicated by appending a period ("`.`"), followed by up to four symbols, each representing a component of the source vector, e.g. `pos.xy` returns a vector containing the first (`x`) and second (`y`) component of the vector `pos`. This syntax can be emulated in C^{++} by exploiting *member fields* of special types with according names – reading and writing to the fields is bypassed to the vector object they belong to. While this approach is compatible to most shading languages, it has major drawbacks: complex intermediate classes and further operator overloading need to be introduced and each vector instance requires at least 1.4 Kbytes of memory[3].

Sh – Swizzling is implemented by overloading the function operator "`()`" that can be invoked with up to four component indices, e.g. `pos(0,1)` is equivalent to `pos.xy` in CG/GLSLANG. While this is easy to implement, the approach lacks an important C^{++} feature – it is not type-safe. For instance, it does not make sense to access the third component of a two-component vector, but even commands like `bad=pos(8)` are compiled without compile-time errors (but they are detected at run-time). Furthermore, the syntax leads to ambiguities when *writing* to a swizzled vector, e.g. the statement "`v4pos(0,0)=v2pos`" is ambiguous, because it is undefined whether the first component of `v4pos` is replaced by the first *or* the second component of `v2pos`.

The swizzling syntax of the GPU^{++} development system is a combination of the aforementioned approaches: While *class methods* are used to safe memory and keep the system simple, a specific class method exists for each swizzling combination, named after the scheme of GLSLANG. For instance, the aforementioned example is written as `pos.xy()`. This approach is type-safe (`pos4d.zz()=pos2d` leads to an error at CPU-based compilation), space efficient (no run-time memory allocation), and time efficient (optimized at compile-time). Furthermore, this syntax is intuitive for by both – non-graphics C^{++} developers (due to the strict compatibility to the C^{++} specification style) and those with graphics hardware experience (due to its similarity to the swizzling syntax of other shading languages).

[3]There are up to four destination components that address one of the four source components, which makes a maximum of $\sum_{n=1}^{4} 4^n = 340$ member fields for a four-component vector. A member field consumes at least four bytes of memory, leading to approx. 1.4 Kbytes for each vector instance. In contrast, using class methods instead of member fields leads to a *single* function-table that is only required at compile-time.

An implementation detail for swizzling regards *temporary objects*, i.e. unnamed objects that are silently created on the stack by the compiler when evaluating expressions (during standard type conversion, argument passing, and function return) [112]. Consider a naive vector class implementation that stores pointers to the DAG nodes directly in the class instance. The following source code manipulates vector components via swizzling:

```
1 vec3 v3one(1.2, -2.0, 3.14);                  // v3one = (1.2, -2.0, 3.14)
2 vec3 v3two(0.0, 0.1, 0.2);                       // v3two = (0.0, 0.1, 0.2)
3 v3two.zx() = v3one.xy();                           // What happens in here?
```

Unfortunately, the first and third component of `v3two` have not changed in line 3, because `v3one.xy()` was written to the "temporary object" that was returned by `v3two.zx()`. Because this temporary object is a deep copy of the first and third component of `v3two`, nothing actually happens. This issue has been solved by introducing a new data structure – a *container* – that can be shared between vector instances, even of different diversity (i.e. `vec1` and `vec3` can share the same container instance).

Matrices

In addition to vectors, *matrices* are natively supported by the GPU^{++} development system through the `mat[2|3|4]` classes – i.e. 2×2 matrices are represented by `mat2`, 3×3 matrices by `mat3`, and 4×4 matrices by `mat4`. In fact, the matrix classes are implemented as lightweight wrappers around the aforementioned vector classes, e.g. `mat4` encapsulates four `vec4` vectors.

The wrapped vectors represent the matrix columns and can be accessed directly using the *array subscript* operator `[]`. Combined with the *array subscript* operator of the vertex class, individual components can be accessed using the syntax "`m[`i`][`j`]`", where i is the column vector of the matrix and j is the row within that column vector. Unfortunately, this syntax does not allow to check vector and matrix dimensions at CPU-based compile-time, but *only* at GPU-based compile-time (i.e. an exception is thrown). Nevertheless, this syntax is recommended, due to the high compatibility to most other matrix APIs.

Similar to vectors, the common mathematical unary and binary operators are overloaded for the matrix classes – *negation*, *addition*, *subtraction*, and *multiplication*. However, in contrast to the aforementioned vector classes, matrix multiplication uses the correct mathematical definition, i.e. it does *not* work component-wise. Furthermore, the operators for *addition*, *subtraction*, and *multiplication* of vector-with-matrix and matrix-with-vector have been implemented (where vectors and matrices need to have the same dimension). Finally, *addition*, *subtraction*, *multiplication*, and *division* with scalars (i.e. one-component vectors of type `vec1` and `floats`) are supported for $n \times n$ matrices.

Note that GPU^{++} provides additional matrix functions, such as the component-wise `matrixCompMul()` function. A list of all overloaded operators can be found in section A.2 of the language referenc and all matrix functions are presented in section A.4.

4.2.2 Data Managing Classes

While custom data types (that enable the embedded creation of a directed acyclic graph using “ad-hoc polymorphism”) have been introduced, additional classes are required that manage these data: the resulting expression DAG is wrapped in the `kernel` class and the input and output arrays are accessed using the `array` and `sampler` classes.

Kernel Programs

The `kernel` template class encapsulates an expression DAG (that represents a kernel program). It covers the DAG’s life-span from creation to GPU-based execution:

Creation – A custom kernel program is *created* by deriving an own class from the `kernel` template class and overriding the pure virtual method `kernel::element()`. From a conceptual viewpoint, this method is called for each element of an array (where the kernel is executed on). However, from the implementation point-of-view, this method is called *just once* – to create the expression DAG as part of the compilation process. Thus, changing input variables may require a recompilation of the kernel (except, if the input variables are qualified as “uniform” or “attribute” – see section 4.2.3).

Optimization – The next step of GPU-based compilation is that the previously created expression DAG is automatically *optimized*. This includes advanced optimization techniques such as algebraic simplifications, substitution of complex or unsupported operations, elimination of “dead code”, and elimination of common sub-expressions. See chapter 5 for a full list of optimization strategies using in the GPU^{++} system.

Translation – The final step of the compilation process *translates* the optimized DAG to a “back-end specific format”. While this is not done in the `kernel` class directly, it provides the *connection* to the back-end via its template parameter. For instance, to use the OpenGL low-level back-end to access the GPU, the developer derives its own kernel program from “`kernel<opengl>`”. Depending on the selected back-end, further optimizations (e.g. vector fusion and register renaming) are applied.

Execute – After the DAG is compiled and transferred to the graphics hardware, it only can be executed. In fact, executing the kernel program is triggered by calling an array class method – this will be explained in the next section. However, the main logic of this operation is located in the `kernel` class, including: the synchronization of all input parameters (i.e. uniform variables, attribute variables, samplers, and input arrays), activation of the kernel program, binding of the output array, and rendering the region into the output array.

The `kernel` class can be used like any other class in the C^{++} language – which includes object-oriented features like polymorphism and inheritance. For instance, the developer is able to derive an intermediate class that provides further GPU-based functionality, like advanced operations for linear algebra or an implementation of the Fast Fourier Transform.

Arrays and Samplers

Kernel programs are executed on the elements of an array, i.e. the computations are performed on each element in the same way. The `array[1|2|3]` class can have up to three dimensions, where each element of the array can have up to four components of a native C++ type, e.g. `array2<float,3>` specifies a two-dimensional array that contains elements of three floating-point values, and `array1<int,1>` represents a one-dimensional array of integers. Arrays can have arbitrary resolutions[4] that can be changed at any time. Note that currently, only two-dimensional arrays are supported as *output arrays*.[5]

Conceptually, an array is a single block of memory that can be accessed by the main and the graphics processor, e.g. output arrays are usually computed by the GPU (using a kernel), and then read out by the CPU. In fact, an array instance manages *two* memory blocks (one in the main memory and one in the graphics memory) and synchronizes them when needed. Note that this may involve time-consuming data transfer between the different memories. To guarantee *efficient memory handling* and *optimal transfer rates* between the CPU and GPU, several speedup techniques are used in the GPU++ `array` classes:

Delayed Allocation – Blocks of main memory are allocated when they are accessed on the CPU for the first time. This is optimal for "temporary" arrays that are used only for GPU-based computations and that are never accessed on the main processor.

Shared Memory – Blocks of memory are shared between multiple arrays (via the assign operator) as long as the developer performs read-only access. If there is CPU-based write access to such shared memory, it is deep-copied to a new main memory block.

Lazy Up- and Download – Data is only transferred between the main and the graphics memory if the following is true: 1) the data is "dirty", i.e. the content on the CPU and the GPU is not synchronized, *and* 2) the data is accessed by the developer.

Late Binding – Back-end binding of main memory is performed at GPU-based run-time, because the used back-end is not known in advance. This allows to use the same array instance with different back-ends in parallel – see section 6.3.

Kernel programs are executed on arrays by calling the class method `array2::execute()`, passing the kernel program as the first argument. Please note that the kernel programm will be implicitly compiled, if `kernel::compile()` was not called before.

In addition, the stream operator "«" is overloaded and has the same effect as calling `execute()` – which means "`a « k`" equals to "`a.execute(k)`". This syntax simplifies the handling of arrays and kernel programs and visually demonstrates their relationship.

[4]Actually, the resolution of arrays is limited by the used graphics back-end and the graphics accelerator that is available at run-time – in fact, most graphics hardware does not support a resolution larger than 4096 elements. Furthermore, a resolution that is not a two-exponent *may* slow down program execution.

[5]In theory, it is also possible to execute kernel programs on one- and three-dimensional arrays, however, in practice, the `ARB_framebuffer_object` extension of the OpenGL back-end does not work properly.

Arrays are used as *output arrays* to store the computation result of a kernel program. Furthermore, they also can be used as *input arrays*, which means that array elements can be read from inside a kernel program. However, array elements cannot be read directly due to the issues described in section 3.5 – an intermediate class, the *sampler*, is required.

The `sampler[1|2|3]` template class stores three properties: First, an `array` instance is attached (note that `array1` corresponds to `sampler1`, `array2` to `sampler2`, etc.). Second, the *wrapping property* specifies how read positions "outside" the range $[0 \ldots 1]$ are handled (*clamp* vs. *repeat*). And third, the *interpolation property* effects how positions "in-between" elements are interpreted (*nearest* vs. *linear*) – see section 3.5 for detailed information.

The same array can be attached to multiple samplers, and the same sampler can be used in multiple kernel programs. The only sampler functionality, that can be used within a kernel program, is the *array subscript* operator "`[]`", i.e. `s2input[v2pos]` accesses the array that is attached to sampler `s2input` at position `v2pos`. Note that the sampler parameters (i.e. wrapping and interpolation) can be changed between several GPU-based execute calls.

There is a major limitation with arrays: It is not allowed to use the same array as input and output for the same kernel program instance. However, this might be necessary from time to time, e.g. for complex computations or so-called "feedback effects". The common approach is the *ping-pong technique*, where one array is used for the input and one is used for the output and that arrays are swapped after each kernel program execute [46].

Regions

Regions were introduced in section 3.2 as a "sequence of coordinates" that forms a closed polygon in the output array. Kernel programs are applied to all elements that are covered by such an "output region", while all other elements of the output array are left untouched.

The `region` class is used to store the aforementioned sequence of coordinates, where each coordinate is a vector of up to four native C^{++} types. For instance, an object of type `region<float,2>` store two-dimensional vectors of `float` scalars and `region<int,1>` encapsulates a list of (one-dimensional) integer values. New coordinates are appended to the back of the existing sequence via the `push()` method that allows up to four parameters.

Similar to `array` objects, `region` objects can also be shared between multiple kernel programs and graphics hardware back-ends. In addition, most of the speedup techniques mentioned in the prior subsection have also been implemented for region objects, including:

- *Shared Memory* – Memory is shared between multiple regions (for read-only access).
- *Lazy Upload* – The region data is transferred if GPU is not synchronized with CPU.
- *Late Binding* – Back-end binding of main memory is performed at kernel execution.

Please note that – in contrast to an array – a region cannot be created or manipulated by the graphics processing unit. Therefore, "delayed allocation" has not been implemented and the "lazy mechanism" does not include *lazy download* of region data.

4.2.3 Type Qualifiers

Computation frequencies have been introduced as a "tool" for better understanding the data flow of a program over its life-span – see section 3.4. In addition, computation frequencies are the foundation of an efficient variable classification scheme that leads to automatic code optimizations at compile-time. Experienced C^{++} developers are familiar with this concept: *Type qualifiers* – such as `const` and `volatile` – are used as a "hint" to bind variables and expressions to specific computation frequencies. For instance, the `const` type qualifier associates the addressed variable with the COMPILE computation frequency, i.e. expressions that only contain such constants can be evaluated at compile-time.

While the C^{++} programming language knows two computation frequencies (i.e. CPU-based COMPILE and EXECUTE), the GPU^{++} development system knows four additional: GPU-based COMPILE, EXECUTE, REGION, and ELEMENT. Therefore, additional type qualifiers are requires to bind variables to the these new computation frequencies. The syntax for the new type qualifiers has been inspired by the built-in C^{++} casting operators. For instance, a 3×3 matrix that is associated to the EXECUTE computation frequency, is declared as "`uniform< mat3 >`" – where the `uniform` keyword is the GPU^{++} type qualifier.

While the concept of type qualifiers is also known in other GPU-based programming languages, they are only used for *type-safety* and *external bindings*, but *not* for optimization. The reason is that other GPU-based development systems usually do not allow kernel programs to span over multiple computation frequencies. As a consequence, the developer has to decide what variables (and therefore which part of the code) belong to each computation frequency. For instance, while there are type qualifiers in SH, they are "local", i.e. they are the same for each computation frequency: *temporary*, *constant*, *input*, and *output*. The same approach can be found in other languages, like CG and GLSLANG.

In contrast, the GPU^{++} development system features a *unified kernel definition* (see section 4.1.2) that uses "global" type qualifiers to perform automatic optimizations to the kernel program – across all the aforementioned computation frequencies.

Uniforms

The `uniform` type qualifier is used to bind custom data types to the EXECUTE computation frequency, i.e. uniforms have not to be specified at GPU-based compile-time, but have to be prepared when the compiled kernel program is executed on the graphics processing unit. This also means that the variable can be changed between distinct GPU-based executes – without the need for recompilation – by assigning a compatible vector or `float`.

As all other custom data types in the GPU^{++} development system, "uniform variables" can be shared between multiple kernel program instances, i.e. assigning a new value to a specific uniform variable will change the behavior of all kernel programs that make use of the variable. However, uniform variables cannot be "connected", i.e. assigning a uniform variable to other uniform variables will lead to a compile-time error. Note that the `uniform` type qualifier can be used in combination with vector and matrix classes of arbitrary sizes.

Attributes

The `attribute` type qualifier is used to bind data to the REGION computation frequency, i.e. attributes change "linearly" between elements of the output array. The aforementioned linearity depends on a `region` object (see section 4.2.2). Whenever a kernel program is executed on an array, the *output region* specifies the elements that are affected, i.e. only the elements inside this region are computed. Note that the output region is attached to the *position attribute*, which is the only input parameter of "`kernel::element()`" – the "position attribute" represents the position of the currently processed output array element.

Regions are attached to attribute variables to control their content over the array, i.e. the value of an attribute at a specific output position is the linear interpolation of the region coordinates for that position. Developer can attach custom regions (using the `region` class) to custom attribute variables (using the `attribute` keyword). The `attribute` type qualifier can only be specified for vector classes (matrices do *not* work). Note that the coordinate count of a region has to be the same for all regions that are used in a kernel program.

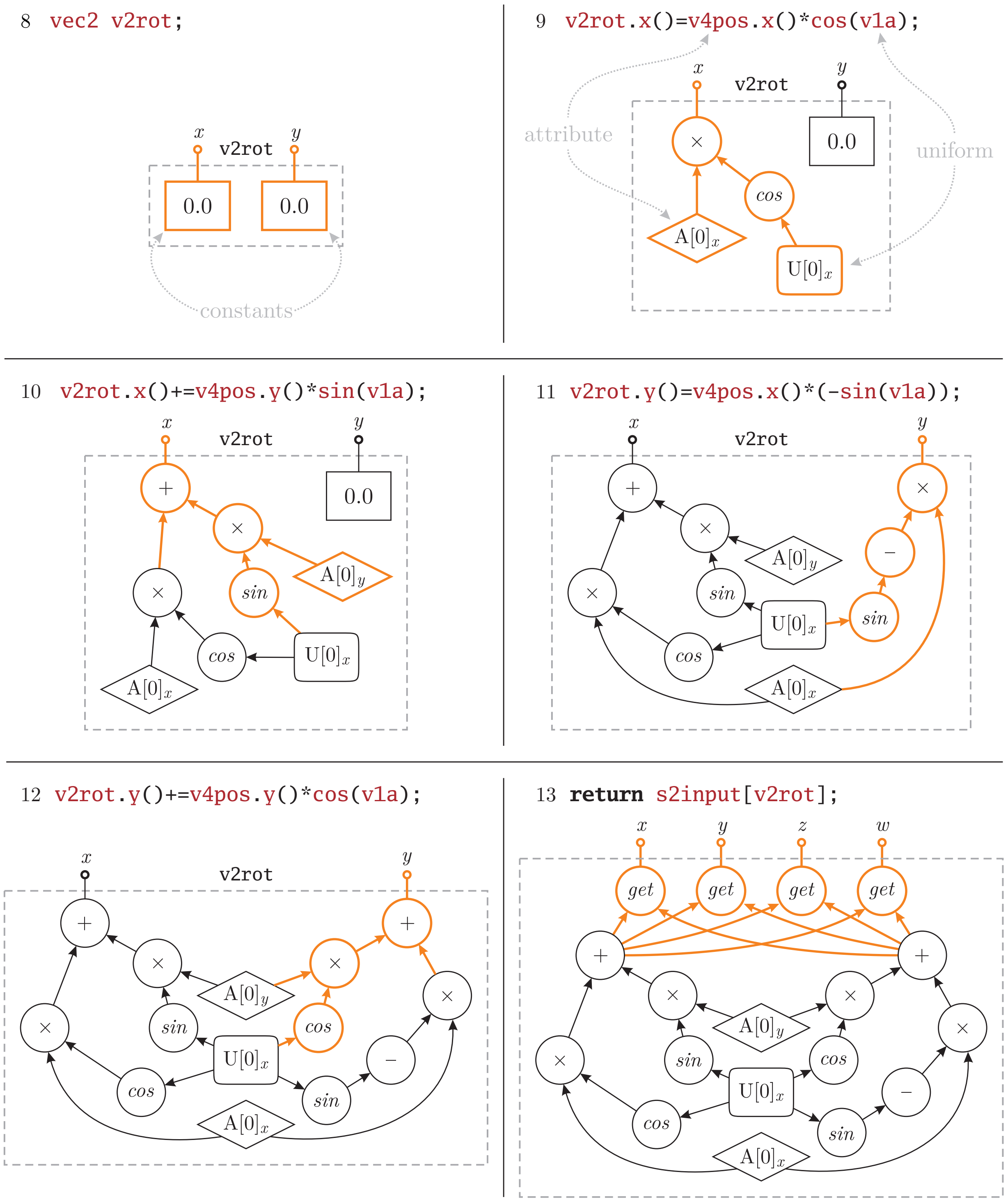
4.2.4 Example

The use of the aforementioned custom data types and data managing classes is demonstrated in listing 4.1 that contains a complete GPU^{++} kernel program – a rotated version of an input image (that is attached to sampler `s2input`) is created by rotating the output element position `v4pos` by an angle of `v1a` (uniform variable). While this listing only shows the kernel program, the full source code can be found in section B.3. The embedded creation of the corresponding expression DAG is illustrated line-by-line in figure 4.2.

```
struct foobar : public kernel< opengl >                        // the kernel uses the OpenGL back-end
{
   sampler2< float, 3 > s2input;                           // sampler to access an attached input array
   uniform< vec1 > v1a;                                              // the rotation angle (in radiants)

   virtual vec4 element(attribute< vec4 > const & v4pos) const
   {
      vec2 v2rot;                                            // instance to store the rotated position
      v2rot.x()  = v4pos.x() * cos(v1a);
      v2rot.x() += v4pos.y() * sin(v1a);                                // compute the rotate x-component
      v2rot.y()  = v4pos.x() * (-sin(v1a));
      v2rot.y() += v4pos.y() * cos(v1a);                               // compute the rotated y-component
      return s2input[v2rot];                                // return the element at the computed position
   }
};
```

Listing 4.1: A simple kernel program that accesses an array element (via sampler `s2input`) at position `v2rot`, which is the element position `v4pos` rotated by angle `v1a`. The full source code is presented in section B.3. The expression DAG creation is illustrated in figure 4.2.



```
vec2 v2rot;
v2rot.x()=v4pos.x()*cos(v1a);
v2rot.x()+=v4pos.y()*sin(v1a);
v2rot.y()=v4pos.x()*(-sin(v1a));
v2rot.y()+=v4pos.y()*cos(v1a);
return s2input[v2rot];
```

Figure 4.2: Embedded creation of an expression DAG is illustrated for the kernel program of listing 4.1 (full source code is shown in listing B.3), where an input texture is rotated by a specific angle. Parts of the DAG, that have changed between source lines, are shown orange.

4.3 Fast DAG Processing

In contrast to other embedded GPU-based languages (like Sh and the RapidMind Development Platform) a fundamental element of Gpu^{++}'s compilation process is the optimization of the expression DAG. This includes algebraic simplifications, constant folding, elimination of common sub-expressions, and back-end specific substitutions – see chapter 5 for details. Applying such automatic DAG optimizations enables the developer to focus stronger on the actual algorithm, instead of (GPU-) specific implementation details.

Because "optimization" means *processing the DAG* the Gpu^{++} development system offers a generic processing infrastructure that is the foundation for all optimizations, but also for the translation to the back-end specific instruction format. The expression DAG class hierarchy (together with its node classes) is a "static" structure, i.e. it is unlikely that it will be extended by other developers. On the other hand, the set of algorithms that is performed on the DAG is manifold and may be changed by others. Thus, an adaptation of the "visitor" design pattern is used as the base for DAG traversal and processing [40].

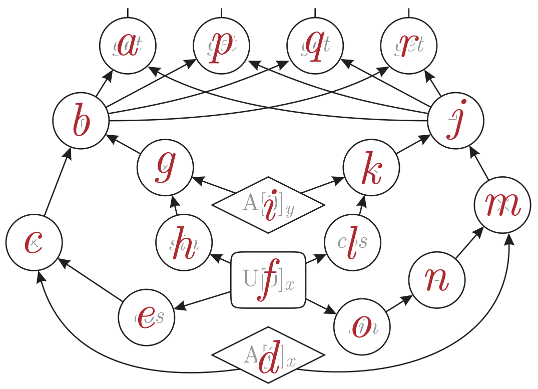
The *visitor design pattern* is used, when many distinct and unrelated operations are performed on the same heterogeneous aggregate structure – this avoids "polluting" the node classes with such operations [3]. By using the visitor pattern, the abstract algorithmic functionality is moved from the "element" classes to the `visitor` base class. New algorithms are implemented by deriving from `visitor` and overriding so-called *element dispatch functions*, where each function is responsible for a specific element type of the aggregate structure. Furthermore, an `iterator` class traverses the aggregate structure and calls the appropriate dispatch functions of a given visitor instance. In summary, `iterator` encapsulates the actual data structure and how it is traversed, while `visitor` wraps the actual functionality.

In the case of Gpu^{++} the aforementioned "aggregate structure" is the expression DAG. Therefore, the `iterator` class implements a depth-first recursive traversal of the DAG and the "elements" are the DAG nodes – see section 4.1. In contrast to the generic design pattern, the `visitor` class contains dispatch functions for specific node types that are called *before* and *after* the recursion. While this leads to a stronger coupling between `visitor` and `iterator`, there are no practical limitations. Figure 4.3 illustrates the traversal (and its dispatch function invocation) of the known expression DAG from section 4.2.4.

Because some of the DAG optimizations are applied multiple times, the `visitor` and `iterator` implementation significantly effects the run-time performance of the compilation stage (the DAG is processed a total of 14 times during optimization). The Gpu^{++} development system uses two techniques to increase the performance for DAG processing:

- *Custom RTTI* – The `dynamic_cast<>` operator is replaced by a more efficient version.
- *Visitor Traits* – Static polymorphism (i.e. templates) is used for the visitor pattern.

Both speedup approaches are explained in the following sections. Please note that the Gpu^{++} implementation of the visitor pattern has the ability to *detect changes* in the expression DAG and automatically traverses the new DAG nodes – which allows on-the-fly optimizations that replace sub-expressions with new nodes.



1. preBinaryVec(a)
2. preBinarySlot(b)
3. preBinarySlot(c)
4. attribute(d)
5. preUnarySlot(e)
6. uniform(f)
7. postUnarySlot(e)
8. postBinarySlot(c)
9. preBinarySlot(g)
10. preUnarySlot(h)
11. postUnarySlot(h)
12. attribute(i)
13. postBinarySlot(g)
14. postBinarySlot(b)
15. preBinarySlot(j)
16. preBinarySlot(k)
17. preUnarySlot(e)
18. postUnarySlot(e)
19. postBinarySlot(k)
20. preBinarySlot(m)
21. preUnarySlot(n)
22. preUnarySlot(o)
23. postUnarySlot(o)
24. postUnarySlot(n)
25. postBinarySlot(m)
26. postBinaryVec(a)
27. preBinaryVec(p)
28. postBinaryVec(p)
29. preBinaryVec(q)
30. postBinaryVec(q)
31. preBinaryVec(r)
32. postBinaryVec(r)

Figure 4.3: The expression DAG of section 4.2.1, its traversal by the iterator class in the GPU^{++} development system, and how the dispatch functions of the visitor class are called. Note that the traversal starts from the "output nodes" a, p, q, and r (i.e. the "get" nodes).

4.3.1 Custom RTTI Mechanism

The common implementation of the aforementioned visitor pattern uses function overloading to distinguish between the different element types of the aggregate structure. But this does not work for GPU^{++}'s expression DAG, because the class type of the visited DAG node is not known at compile-time [112]. While the class type of a specific DAG node can easily be detected at run-time using the built-in `dynamic_cast<>` operator, the native C^{++} *run-time type information* (RTTI) mechanism is known to be slow in many cases [59]. As a consequence, the GPU^{++} development system has implemented its own RTTI mechanism.

The DAG node class hierarchy is very simple – one root class (i.e. `node`), nine first-level category classes (e.g. `nodeValue`, `nodeBinarySlot`, and `nodeTransitVarying`) and 42 second-level operation classes (e.g. `opAdd`, and `opCross`). Such a sparse and static class hierarchy allows efficient RTTI by providing the following information for each DAG node:

- The static constant `class_id` is created as follows: Bits 0–7 represent the first class hierarchy level (i.e. `node`) and are set to `0x01`, bits 8–15 represent an index for the category classes that are directly derived from `node` (e.g. `nodeBinarySlot`), and bits 16–23 represent an index for the operation classes (e.g. `nodeBinarySlot<opSub>`).
- The static constant `class_mask` is created accordingly: For `node` bits 0–7 are set, for category classes bits 0–15 are set, and for the operation classes bits 0–23 are set.
- The virtual method `id()` for each DAG node class returns the static constant `class_id` of that class, for instance, `nodeFoobar::id()` returns `nodeFoobar::class_id`.

Finally, to test at run-time if pointer `pNode` is derived from DAG node class `NODE`, the conditional statement "`(pNode->id() & CLASS::class_mask) == CLASS::class_id)`" needs to be `true`. This statement is encapsulated in the template class `node_cast<>` that can be used as a direct replacement for the built-in `dynamic_cast<>` operator. Figure 4.4 represents the class hierarchy of all expression DAG nodes using the *Unified Modeling Language* [105]. The resulting speed-performance improvements are illustrated in table 4.1 on page 43.

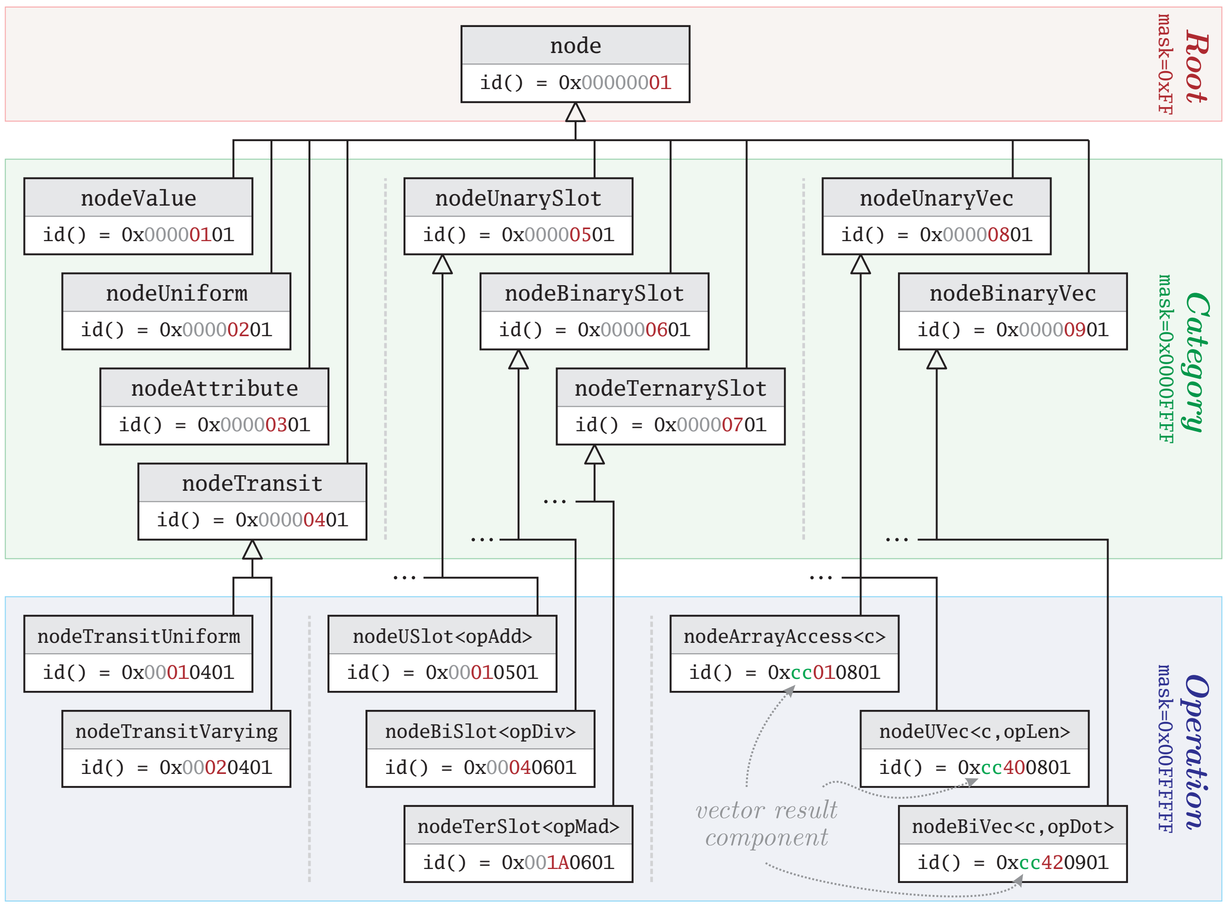


Figure 4.4: A partial UML class diagram of the expression DAG node inheritance hierarchy. The hierarchy consists of only three inheritance levels (i.e. root, category, and operation) and therefore allows a very efficient mechanism for custom run-time type information.

4.3.2 Efficient Use of Templates Using Visitor Traits

The visitor pattern is implemented via template classes, i.e. a `visitor`-derived class is the template parameter for the `iterator` template class. As a consequence, the supported dispatch functions are known at compile-time. In other words, the compiler knows what DAG node types are important in the DAG traversal. While this results in fairly optimized code (because the compiler can inline the dispatch function definitions that are provided by the `visitor`-derived class), there is more potential for compile-time optimizations.

By analyzing the algorithms that are performed on the expression DAG, a crucial observation can be made: Most of the `visitor`-derived classes override just a small subset of dispatch functions. Furthermore, the `iterator` class performs the same operations for all node types: the custom RTTI mechanism is used to determine a specific DAG node type and the according (inlined) dispatch function is called – whether it contains the default implementation or has been overridden in the `visitor`-derived class. In other words, due to the small subset of overridden dispatch functions, the default implementation (which actually does nothing) is called most of the time. As a matter of fact, the `iterator` class handles ten distinct node types, and calls fifteen according dispatch functions of the `visitor` class – even if a only single dispatch function has been overridden by the `visitor`-derived class.[6] Obviously, much of the run-time is "wasted" for handling expression DAG nodes with "empty" dispatch functions.

The aforementioned issue could be solved by a compile-time approach that detects vital DAG node types and the corresponding dispatch functions of the `visitor`-derived class. Ideally, non-overridden dispatch functions should result in exclusion of any relevant code, like the RTTI-based node type detection. The GPU^{++} system uses a novel approach to achieve such "visitor-dependent compilation" by extending the "type traits" technique.

In brief, the *type traits* technique is used to establish associations between pieces of metadata [3]. A prominent example is the "BOOST Type Traits Library" that mainly provides boolean-valued meta-functions the answer questions about the fundamental properties of native C^{++} types at compile-time, e.g. "`boost::is_pointer<TYPE>::value`" is "`true`" if "`TYPE`" is a pointer, and "`false`" otherwise [1]. In other words, "type traits" provide the compiler with type-specific properties. Such an approach can be extended to shift the dynamic polymorphism of the visitor pattern to static polymorphism that is available at compile-time: the `visitor`-derived class contains the dispatch functions together with a set of *visitor traits* that answer the question if specific functions are available or not. As a consequence, there is no need for a virtual base class anymore, because all important information is stored in the visitor traits, which is part of the `visitor`-derived class.

To get a better understanding of the approach, the classic implementation of the visitor pattern is compared to "visitor traits". As an example, the source code that handles the `nodeBinarySlot` node is examined (this node represents a *binary slot operation*). First, the `visitor` base class defines the default dispatch functions – that actually do nothing:[7]

```
class visitor {
   ...
   virtual node * preBinarySlot (nodeBinarySlot * pNode) { return pNode; }       // before recursion
   virtual node * postBinarySlot(nodeBinarySlot * pNode) { return pNode; }        // after recursion
   ...
}
```

The presented dispatch functions – default or overridden – are called by the `iterator` template class, whenever a DAG node of type `nodeBinarySlot` is visited during the expression DAG traversal. The relevant code in the `iterator` class looks like this:

[6]The node types are: unary/binary/ternary slot operation, unary/binary vector operation, uniform and varying transition, constant, attribute, and uniform. Each of the nodes types result in a dispatch function call, where each "operation node" results in two calls (one call *before* and one *after* the recursion).

[7]In fact, the default implementation does something: it returns the node pointer that was passed as an argument. This tells the `iterator` class to keep the expression DAG as it is. In contrast, if the dispatch function returns a different node pointer, that node replaces the original node in the expression DAG.

```
template<class VISITOR> class iterator {
   VISITOR   m_visitor;                                           // the visitor is a class member
   ...
   void recursive(node * pNode) {                             // recursive expression DAG traversal
      ...
      nodeBinarySlot * pBiSlot = node_cast<nodeBinarySlot>(pNode);          // custom RTTI mechanism
      if(pBiSlot) {

         node * pPreNewNode = m_visitor.preBinarySlot(pBiSlot);      // call "pre" dispatch function
         if(pNode != pPreNewNode) { /* replace the old node by a new one */ }

         recursive(pBiSlot->access1stOperand());             // invoke recursion for the 1st operand
         recursive(pBiSlot->access2ndOperand());             // invoke recursion for the 2nd operand

         node * pPostNewNode = m_visitor.postBinarySlot(pBiSlot);   // call "post" dispatch function
         if(ptrNode != pPostNewNode) { /* replace the old node by a new one */ }
      }
   ...
};
```

Finally, consider class **visitorTest** that is derived from **visitor** and encapsulates the actual algorithm, i.e. that the **postBinarySlot()** dispatch function has been overridden.

This example now is adapted to use the *visitor traits* approach: The major difference is that the base class **visitor** becomes obsolete, because all necessary polymorphism information is stored in the "traits structure" **traits** – that is part of the "derived" visitor class. For instance, the aforementioned dispatch function requires two traits – **has_binary_slot_pre** and **has_binary_slot_post**.[8] This leads to the following **visitorTest** class implementation:

```
class visitorTest {
   ...
   struct traits {
      ...
      static bool const has_binary_slot_pre  = false;     // function "preBinarySlot()" NOT provided
      static bool const has_binary_slot_post =  true;         // function "postBinarySlot()" provided
      ...
   };
   ...
   node * postBinarySlot(nodeBinarySlot * pNode) { /* finally, the actual implementation */ }
}
```

In addition, the **iterator** class has to be adapted to make use of the polymorphism information stored in the *visitor traits structure* – in here, the information is used to decide whether a dispatch function is called or not. The relevant parts of the source code are:

[8]The GPU^{++} development system makes use of a third trait, **has_binary_slot_recursive**, that is used to avoid an unnecessary recursion for the operands of the binary slot operation. This leads to a further speedup if all aforementioned visitor traits are **false**, because the RTTI detection code is skipped completely.

```
template<class VISITOR> class iterator {
   ...
   void recursive(node * pNode) {                          // recursive expression DAG traversal
      ...
      nodeBinarySlot * pBiSlot = node_cast<nodeBinarySlot>(pNode);          // custom RTTI mechanism
      if(pBiSlot) {

         if(VISITOR::traits::has_binary_slot_pre) {
            node * pPreNewNode = m_visitor.preBinarySlot(pBiSlot);   // call "pre" dispatch function
            if(pNode != pPreNewNode) { /* replace the old node by a new one */ }
         }
         ...
         if(VISITOR::traits::has_binary_slot_post) {
            node * pPostNewNode = m_visitor.postBinarySlot(pBiSlot);// call "post" dispatch function
            if(ptrNode != pPostNewNode) { /* replace the old node by a new one */ }
         }
      }
   ...
```

As can be seen, the *visitor traits* features the same functionality, but leads to optimal code, due to the compile-time polymorphism. In fact, a `visitor` class, that does not provide any dispatch functionality leads to an `iterator` class that does not perform any processing.

4.3.3 Performance Results

The aforementioned acceleration techniques for DAG processing have been benchmarked using the "Smear × 4" kernel program of the filtered back-projection (see section 7.5.2). Table 4.1 illustrates the basic implementation of the visitor pattern (using native C^{++} RTTI) in comparison to the *custom RTTI* approach and the novel *visitor traits* technique.

`visitor` Implementations	**Classic impl.**	**Custom RTTI**	**RTTI & Traits**
Embedded Creation	204.7 ms	135.4 ms	106.2 ms
Algebraic Simplifications	409.1 ms	100.0 ms	78.40 ms
Algebraic Reassociation	180.4 ms	74.62 ms	58.53 ms
Com. Sub-Expression Elimination	396.7 ms	338.0 ms	265.1 ms
Frequency Transitions	187.5 ms	111.5 ms	87.46 ms
Substitutions (OpenGL)	130.7 ms	26.08 ms	20.46 ms
Overall	**1509 ms**	**785.6 ms**	**616.2 ms**

Table 4.1: Run-time performance for expression DAG processing of the "Smear × 4" kernel program (see section 7.5.2) using the C^{++} run-time type information, *custom RTTI*, and the novel *visitor traits* approach. The optimization techniques are presented in chapter 5.

Chapter 5

Kernel Optimizations

The aforementioned embedded DAG creation leads to a one-to-one representation of the kernel program's source code. As a consequence, the DAG is *not* optimized. While the C^{++} compiler has optimized the CPU-based code, there are two issues that disable optimizations for the GPU-based kernel program: First, the ad-hoc polymorphism of the new vector and matrix types restricts arithmetic simplifications (even for compile-time constants), i.e. the sum of the two compile-time constants in "`vec1(3)+vec1(5)`" *does not* leat to "`vec1(8)`" at compile-time. Second, the "application life-cycle" concept of section 3.1 draws a distinction between CPU- and GPU-based constants, i.e. CPU-based run-time variables may be used as GPU-based constants. Therefore, kernel optimizations based on these variables cannot be evaluated at CPU-based compile-time by the C^{++} compiler.

While the aforementioned optimization issues do not exist in "string-based" shader languages like Cg, GLslang and Brook (such systems feature dedicated compiler tools for the GPU-based kernel program source code), they could lead to sub-optimal kernel programs in embedded shading languages like Sh and the RapidMind Development Platform [77, 101, 80]. However, no embedded shading language has addressed these issues before. Instead, optimizations of any kind have been imposed to the developer.

The Gpu^{++} development system differs significantly from other embedded approaches by performing many DAG optimizations automatically. This includes well-known arithmetic simplifications, like *algebraic reassociation* (section 5.1.3), *common sub-expression elimination* (section 5.1.4) and *substitution* (section 5.2.1). In addition, section 5.2.2 describes the efficient partitioning of computation frequencies, which is a novel approach to abstract the different processing units of the GPU (i.e. vertex and fragment unit). This technique allows a single kernel program to be distributed on the GPU's processing stages.

Note that most of the optimizations in section 5.1 are optional and can be disabled to speedup the GPU-based compile-time performance. However, manual optimization of CPU-based run-time variables is very difficult and leads to inefficient code in most cases. This could result in a drastic slowdown of the kernel program's run-time performance, which then might lead to an overall performance-decrease of the application.

5.1 Expression Optimization Strategies

The optimization of expression graphs is known from disciplines like compiler design and computer algebra systems [86, 128]. While the optimization strategies of this section are well known in other domains, they are novel in the context of embedded shading languages. Hence, a brief description of each optimization strategy is given, as well as references that contain more detailed information. Please note that more advanced techniques exist to optimize the expression DAG [28]. However, they have not been integrated, due to their minimal benefit in comparison to the slowdown of GPU-based kernel program compilation.

5.1.1 Basic Optimizations

One of the most evident optimizations is **dead code elimination** (DCE), where sub-expressions that are irrelevant to the result of the complete expression are removed. This happens very often in practice. For instance variables are overwritten in conditional statements or a node is replaced by another node during optimization. DCE is automatically performed via *reference counting* and *smart pointers*: whenever a "reference count" of a DAG node becomes zero the node is "dead code" and is removed from memory [23].

A further basic optimization is **constant folding** (CF), which has been illustrated for a tree in figure 4.1. Ideally, the full expression DAG can be evaluated – which is the case for kernel programs that only uses constant variables. Nevertheless, even each evaluated sub-expression increases the run-time performance. For instance, the expression DAG that represents the computation "`3.0 * (2.0 - (b + sin(0.5))) * 2.0`" is automatically optimized to "`6.0 * (1.52057446 - b)`" by the Gpu^{++} development system.

5.1.2 Algebraic Simplifications

Some expressions can be simplified by replacing them with an equivalent but more efficient expression, e.g. "`1*i+0`" can be simplified to "`i`" [86]. While developers generally do not write such expressions, they may silently be generated after macro expansion or conditionals. Simplifications can be classified in one of the following three categories (see table 5.1):

Sign Beautifying – Each *negation* node (i.e. unary subtraction) is checked for children that also represent negations, e.g. "`-(-i)`". In this case, both negations are removed from the DAG. Additionally, all *positive* nodes (i.e. unary addition) are also removed.

Identity Values – Whenever the visitor reaches a basic arithmetic operation, it checks if one of the two children is an *identity value* (i.e. "`0.0`" or "`1.0`"). Many of these operations can be simplified to a constant value or a single equivalent DAG node.

Sign Uplifting – For basic arithmetic operations, where one (or both) of the children is a "negation", the *sign* can be *uplifted*, i.e. it is made the new parent of the operation. This may lead to node reductions, either directly or in subsequent traversal passes.

$\circ$	$\circ a$	$\circ(\circ a)$	$0 \circ a$	$a \circ 0$	$1 \circ a$	$a \circ 1$	$-a \circ b$	$a \circ -b$	$-a \circ -b$
$+$	a	a	a	a	$1+a$	$a+1$	$b-a$	$a-b$	$-(a+b)$
$-$	$-a$	a	$-a$	a	$1-a$	$a-1$	$-(a+b)$	$a+b$	$b-a$
$\times$	undefined		0	0	a	a	$-(a \times b)$	$-(a \times b)$	$a \times b$
$\div$			0	$a \div 0$	$1 \div a$	a	$-(a \div b)$	$-(a \div b)$	$a \div b$

Table 5.1: Arithmetic simplifications for operator "$\circ$". Some expressions cannot be optimized (gray), while others may lead to an instruction count reduction in subsequent stages (blue). While the first two columns relate to *sign beautifying*, the next four columns represent the "identity value" optimizations. Finally, the remaining columns relate to "sign uplifting".

A further algebraic simplification relates to the `select()` command (which is defined component-wise as "`select(a,b,c)=(a<0)?b:c`" – see section A.4): if the 1st parameter represents a GPU-based constant value and it is negative, the operation is replaced by the 2nd parameter, otherwise the operation is replaced by the 3rd parameter. However, if the 1st parameter is `uniform` or `attribute` the `select()` command cannot be optimized.

5.1.3 Algebraic Reassociation

Expressions and sub-expressions that only contain constant values are usually handled by "constant folding" that has been explained in section 5.1.1. But CF does not work for expressions with "partially constant" children, e.g. the expression "$(a + C_1) - C_2$" contains the constant values C_1 and $C_2$and the variable `a`. This expression cannot be evaluated via CF, because C_1 and C_2 do not share the same parent node in the expression graph.

Such expressions are simplified by *algebraic reassociation*, where the expression is rearranged to gather constants on the same DAG level [16]. In fact, this "grouping of constants" rearranges the aforementioned example to "$a + (C_1 - C_2)$", which can be further simplified by a subsequent constant folding step, leading to "$a + C_3$". Without loss of generality, only constants that are spread over two levels in the expression DAG need to be checked, because the recursive nature of the DAG traversal resolves more levels automatically.[1] All possible two-level constellations are presented in table 5.2.

While the same technique may be applied to expressions that mix addition/subtraction with multiplication/division, this does not lead to any reduction of DAG nodes. For instance, the expression "$(a + C_1) * C_2$" could be rearranged to "$(a * C_2) + (C_1 * C_2)$". However, constant folding leads to "$(a * C_2) + C_3$". There is no optimization, because the operation count has not decreased and the constant value count stays the same.[2]

[1]This does not work if the constant folding of the "group of constants" is postponed to a subsequent traversal. Instead, the constants are evaluated directly within the implementation of the visitor class.

[2]Even worse, such a rearrangement could be disadvantageous; C_1 might not be eliminated via DCE, because it is used somewhere else in the DAG – hence, three constants need to be stored, instead of two.

$\circ$	$\bullet$	$(a \circ \mathcal{B}) \bullet \mathcal{C}$	$(\mathcal{B} \circ a) \bullet \mathcal{C}$	$\mathcal{C} \bullet (a \circ \mathcal{B})$	$\mathcal{C} \bullet (\mathcal{B} \circ a)$
$+$	$+$	$a + (\mathcal{B} + \mathcal{C})$	$a + (\mathcal{B} + \mathcal{C})$	$a + (\mathcal{B} + \mathcal{C})$	$a + (\mathcal{B} + \mathcal{C})$
$-$	$+$	$a + (\mathcal{C} - \mathcal{B})$	$(\mathcal{B} + \mathcal{C}) - a$	$a + (\mathcal{C} - \mathcal{B})$	$(\mathcal{B} + \mathcal{C}) - a$
$+$	$-$	$a + (\mathcal{B} - \mathcal{C})$	$a + (\mathcal{B} - \mathcal{C})$	$(\mathcal{C} - \mathcal{B}) - a$	$(\mathcal{C} - \mathcal{B}) - a$
$-$	$-$	$a - (\mathcal{B} + \mathcal{C})$	$(\mathcal{B} - \mathcal{C}) - a$	$(\mathcal{B} + \mathcal{C}) - a$	$a - (\mathcal{B} - \mathcal{C})$
$\times$	$\times$	$a \times (\mathcal{B} \times \mathcal{C})$	$a \times (\mathcal{B} \times \mathcal{C})$	$a \times (\mathcal{B} \times \mathcal{C})$	$a \times (\mathcal{B} \times \mathcal{C})$
$\div$	$\times$	$a \times (\mathcal{C} \div \mathcal{B})$	$(\mathcal{B} \times \mathcal{C}) \div a$	$a \times (\mathcal{C} \div \mathcal{B})$	$(\mathcal{B} \times \mathcal{C}) \div a$
$\times$	$\div$	$a \times (\mathcal{B} \div \mathcal{C})$	$a \times (\mathcal{B} \div \mathcal{C})$	$(\mathcal{C} \div \mathcal{B}) \div a$	$(\mathcal{C} \div \mathcal{B}) \div a$
$\div$	$\div$	$a \div (\mathcal{B} \times \mathcal{C})$	$(\mathcal{B} \div \mathcal{C}) \div a$	$(\mathcal{B} \times \mathcal{C}) \div a$	$a \div (\mathcal{B} \div \mathcal{C})$

Table 5.2: Grouping of constant values in two cascading binary expressions for the operations "$\circ$" and "$\bullet$". The evaluation of grouped constants (shown in blue) leads to a reduction of the instruction count and enables additional optimizations in higher levels of the graph.

5.1.4 Common Sub-Expression Elimination

An expression is a *common sub-expression* (CSE) if it has been computed previously for the same DAG and the values of the operands have not changed since then. Recomputing the expression can be eliminated by using the value of the previous computation [22].

Usually, developers eliminate the CSE manually while writing the source code. However, the greatest source of CSEs is intermediate code generated by the compiler, e.g. array indexing calculations, CPU-based run-time variables that become GPU-based compile-time constants, and macro expansions that result in CSEs not apparent in the original source code. Furthermore, constants, uniforms, and attributes are also common sub-expression that are automatically reused to save memory resources on the graphics hardware.

GPU^{++} uses *fingerprints* to identify CSEs. A "fingerprint" is a signature that represents the content of a DAG node without actually evaluating the node. If two fingerprints are equal, they compute the same expression. As a consequence, two nodes with identical fingerprints, represent the same CSE and all references to the second node can be replaced by references to the first node. While many fingerprint functions exist that fulfill the aforementioned "equality" property, two additional attributes have been considered:

Commutativity – Fingerprints of commutative operations are invariant of the children's order. For instance, the fingerprint of "`a+b`" is the same as fingerprint of "`b+a`". In contrast, "`a-b`" results in a different fingerprint than "`b-a`".

Run-Time Efficiency – The fingerprint function of the GPU^{++} development system has been designed to create fingerprints in constant time. Furthermore, the identity check is very efficient, because fingerprints differ significantly even for similar expression, thus, only small parts of the fingerprints need to be compared to check their identity.

5.2 GPU-Driven Optimizations

While the aforementioned optimization strategies are known from other domains, the techniques in this section are dedicated to the graphics processing unit. Furthermore, the following techniques are not optimizations by the common meaning of "reducing the count of expression DAG nodes to speedup the kernel program's run-time", but by the meaning of "transformation of the expression DAG to gain better performance". This includes the substitution of unsupported operations, because otherwise this would result in a time-consuming CPU-based emulation.

5.2.1 Substitutions

The substitution optimization replaces "unsupported" operations by (a sequence of) equivalent but supported operations. See figure 5.1 for a graphical illustration of substituting the "modulus" node in the DAG. Please note that "equivalent" is a theoretical statement. In practice, the replacement of a single operation by a sequence of operations may lead to the accumulation of inaccuracies. Furthermore, some operations are replaced by approximations – e.g. using Taylor series – which leads to additional deviations in precision.

Unsupported operations are specified via *unit traits*, which is a set of CPU-based compile-time constants that specify for each DAG operation node whether it is supported by the back-end or not. Once more, templates are used to create efficient code: the C^{++} compiler is able to eliminate all managing and substitution code for operations that are natively supported by the back-end. Consequently, back-ends that support all GPU^{++} operations result in empty substitution classes that do not consume memory or time.

There are two sets of traits, because the instructions differ for the GPU's vertex and fragment unit. For instance, the OpenGL assembly language supports `sin()` and `cos()` operations in the fragment unit, while they need to be substituted for vertex processing.

All available substitutions for component-wise operations are shown in table 5.3, where the DAG operation nodes are entitled with the internal GPU^{++} name (e.g. "`opAdd(`a`, `b`)`" represents the "addition of node a with node b"). All operations are defined in section A.4 of the language reference. Furthermore, substitutions are available for most vector operations. Note that some fundamental operations cannot be substituted, e.g. "`+`", "`min()`", and "`pow()`".

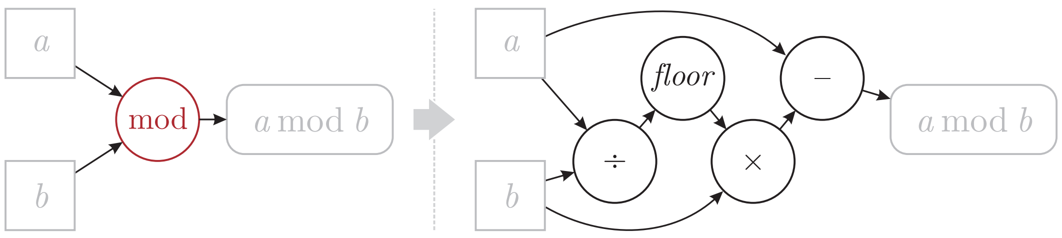


Figure 5.1: The node representing the "modulus" operation is substituted by a sequence of new nodes that compute an equivalent result, using the relation "$a \bmod b = a - b \times \lfloor a/b \rfloor$".

Name of operation	Original node	Substituted by this sequence of nodes
Division	`opDiv(`a`, `b`)`	`opMul(`a`, opRcp(`b`))`
Reciprocal	`opRcp(`a`)`	`opDiv(`1.0`, `a`)`
Floor	`opFlr(`a`)`	`opSub(opCil(opSub(`a`)))`
Ceil	`opCil(`a`)`	`opSub(opFlr(opSub(`a`)))`
Fraction	`opFrc(`a`)`	`opSub(`a`, opFlr(`a`))`
Modulus	`opMod(`a`, `b`)`	`opSub(`a`, opMul(`b`, opFlr(opDiv(`a`, `b`))))`
Linear Interpolation	`opMix(`a`, `b`, `c`)`	`opMad(opSub(`b`, `a`), `c`, `a`)`
Multiply and Add	`opMad(`a`, `b`, `c`)`	`opAdd(opMul(`a`, `b`), `c`)`
Compare	`opCmp(`a`, `b`, `c`)`	`opMad(opSge(`a`, `0.0`), opSub(`c`, `b`), `b`)`
Greater-or-Equal	`opSge(`a`, `b`)`	`opCmp(opSub(`a`, `b`), `0.0`, `1.0`)`
Radiant-to-Degree	`opRad(`a`)`	`opMul(`a`, `$\pi/180.0$`)`
Degree-to-Radiant	`opDeg(`a`)`	`opMul(`a`, `$180.0/\pi$`)`
Tangent	`opTan(`a`)`	`opDiv(opSin(`a`), opCos(`a`))`
Natural Exponent	`opExp(`a`)`	`opPow(`e`, `a`)`
Natural Logarithm	`opLog(`a`)`	`opMul(opLog2(`a`), `$\ln(2.0)$`)`
Binary Exponent	`opExp2(`a`)`	`opPow(`2.0`, `a`)`
Binary Logarithm	`opLog2(`a`)`	`opMul(opLog(`a`), `$1.0/\ln(2.0)$`)`
Square Root	`opSqrt(`a`)`	`opRcp(opISqrt(`a`))`
Inverse Square Root	`opISqrt(`a`)`	`opRcp(opSqrt(`a`))`

Table 5.3: All substitutions that are available in the GPU^{++} development system. Operations that are approximated using Taylor series (i.e. all trigonometric functions), or the Henron algorithm (i.e. square root function) are not shown – see section 5.2.1 for a discussion.

Approximation of Complex Operations

If `sin()`, `cos()`, or `tan()` (together with their inverse versions) are not natively supported by the back-end, they are approximated by partially evaluating of infinite sum based on *Taylor series* [2]. To get adequate results, at least the first six terms of the series need to be taken into account – table 5.4 gives detailed information about the terms used and the resulting accuracy. As can be seen, for some operations (especially the inverse functions), the approximation's variance is nearly 0.04% compared to the exact definition. Therefore, the developer is able to increase the number of the Taylor series terms up to 32. However, if the context demands even better accuracy it is recommended to use an array-based approach that precomputes the trigonometric values to a look-up-table.

The *square root* is approximated by evaluating a few iterations of *Henron's algorithm* – sometimes also called *Babylonian algorithm* [2]. The algorithm converges very fast, hence, six iterations are used by default (but this can be adapted by the developer).

Operation	**Approximation**							**Variance**
Sine	x	$-\frac{1}{3!}x^3$	$+\frac{1}{5!}x^5$	$-\frac{1}{7!}x^7$	$+\frac{1}{9!}x^9$	$-\frac{1}{11!}x^{11}$	$+\frac{1}{13!}x^{13}$	4.1×10^{-8}
Cosine	1	$-\frac{1}{2!}x^2$	$+\frac{1}{4!}x^4$	$-\frac{1}{6!}x^6$	$+\frac{1}{8!}x^8$	$-\frac{1}{10!}x^{10}$	$+\frac{1}{12!}x^{12}$	3.5×10^{-10}
Tangent	x	$+\frac{1}{3}x^3$	$+\frac{2}{15}x^5$	$+\frac{17}{315}x^7$	$+\frac{62}{2835}x^9$	$+\frac{1382}{155925}x^{11}$	$+\frac{21844}{6081075}x^{13}$	4.2×10^{-4}
Arcsine	x	$+\frac{1}{6}x^3$	$+\frac{3}{40}x^5$	$+\frac{5}{112}x^7$	$+\frac{35}{1152}x^9$	$+\frac{63}{2816}x^{11}$	$+\frac{231}{13312}x^{13}$	3.5×10^{-4}
Arccosine	$\frac{\pi}{2} - x$	$-\frac{1}{6}x^3$	$-\frac{3}{40}x^5$	$-\frac{5}{112}x^7$	$-\frac{35}{1152}x^9$	$-\frac{63}{2816}x^{11}$	$-\frac{231}{13312}x^{13}$	3.5×10^{-4}
Arctangent	x	$-\frac{1}{3}x^3$	$+\frac{1}{5}x^5$	$-\frac{1}{7}x^7$	$+\frac{1}{9}x^9$	$-\frac{1}{11}x^{11}$	$+\frac{1}{13}x^{13}$	5.6×10^{-5}

Table 5.4: The expressions used to approximate the trigonometric operations, based on the first six terms of the corresponding Taylor series. The last column shows variance σ^2 in comparison to the exact evaluation [2]. The input domain ranges are: $[-\pi, +\pi]$ for *sine* and *cosine*, $[-2\pi/5, +2\pi/5]$ for *tangent*, and $[-1, +1]$ for *arcsine*, *arccosine*, and *arctangent* [2].

5.2.2 Frequency Transitions

As aforementioned in section 4.2.3, *type qualifiers* are used to mark a variable as constant input for a specific *computation frequency* – i.e. `uniform` variables are constant in the EXECUTE computation frequency, and `attribute` variables are constant in the REGION frequency. In addition, computation frequencies are assigned to "processing units": while the COMPILE and EXECUTE frequencies are both attached to the CPU, the REGION frequency is linked to the vertex unit and the ELEMENT frequency is attached to the fragment unit.

But which *operation* of the kernel program belongs to which computation frequency? Or in other words, how is the expression DAG separated into the different frequencies?

Obviously, all operations could be processed in the highest frequency, i.e. the fragment unit of the GPU. But this is sub-optimal, due to recomputations of values for each array element. Alternatively, most of the available GPU-based programming languages (e.g. CG, GLSLANG, SH or BROOK) impose the decision to this questions to the developer. For instance, SH avoids type qualifiers for different computation frequencies by introducing the so-called "compilation targets" `gpu::vertex` and `gpu::fragment`. Similarly in GLSLANG, where shader programs are attached to the different processing units by using different OPENGL extensions – `ARB_vertex_shader` and `ARB_fragment_shader`. While such an approach allows finer control of the processing units, it complicates GPGPU significantly.

In contrast, the GPU^{++} development system uses the novel "unified kernel definition" approach (see section 4.1.2) that **automatically** finds the optimal *frequency transitions*, i.e. those edges in the DAG that separate two computation frequencies. Such a "transition scheme" has been used in the "constant folding" technique to separate the graph into operations that can be performed immediately on the CPU and operations that need to be done on the GPU. The used strategy has been very simple: do as much as possible in low frequencies. For "constant folding" this means that if the operation's children are constant, the operation is replaced by the result of its evaluation (which again is a constant value).

The same strategy can be used for all other computation frequencies: For a frequency F, traverse "bottom-to-top" from the leaf nodes of the expression DAG to its root and check for each node if all children are part of F *and* if the node itself fulfills some criteria specific to F. If both conditions are true, the node also becomes part of F, otherwise the edges to the node's children that belong to F are marked as "frequency transitions".

Transition From the CPU to the GPU

This section is about *uniform transition*, i.e. the transition of variables between the EXECUTE computation frequency (attached to the CPU) and the REGION computation frequency (attached to the GPU). Note that a "uniform transition" is affected by the `uniform` type qualifier, but "affected" does not mean that all edges emanating from "uniform nodes" automatically become transitions. For instance, if $\mathtt{U}_1$ and $\mathtt{U}_2$ are both qualified as `uniform`, the result of "$(\mathtt{U}_1+\mathtt{U}_2)$" is uniform as well: while the edges emanating directly from $\mathtt{U}_1$ and $\mathtt{U}_2$ are no "uniform transitions", the edge emanating from operator "+" is.

The strategy to find "uniform transition" edges is based on the aforementioned idea: By traversing "bottom-to-top" from the DAG's leaf nodes to its root, each node n_i is checked to be part of the EXECUTE frequency and is added to F_X in this case. Otherwise, a "uniform transition" is added to each children c_i of n_i that is part of F_X. The algorithm is:

Algorithm 5.1: Find the set of transition edges T_X for the EXECUTE frequency F_X

```
 1 $F_X \leftarrow \emptyset$;
 2 $T_X \leftarrow \emptyset$;
 3 forall nodes $n_i$ on a "bottom-to-top" traversal of the expression DAG do
 4     if $n_i$ = constant variable or $n_i$ = uniform variable then
 5         $F_X \leftarrow n_i$;
 6     else if $n_i$ = operation node then
 7         if $n_i \neq$ opGet and $\forall(c_i$ is child of $n_i) : c_i \in F_X$ then
 8             $F_X \leftarrow n_i$;
 9         else
10             forall children $c_i$ of node $n_i : c_i \in F_X$ do
11                 $T_X \leftarrow c_i$;
12             end
13         end
14     end
15 end
```

Please note the implementation detail in line 7: The `opGet` operation (representing array access) is handled with special care, because it cannot be evaluated on the CPU at GPU-based compile-time even if all children are constant values. While `opGet` could be evaluated in theory, it leads to time- and memory-consuming array transfers between CPU and GPU. Figure 5.2a shows the *uniform transitions* for the expression DAG of section 4.2.4.

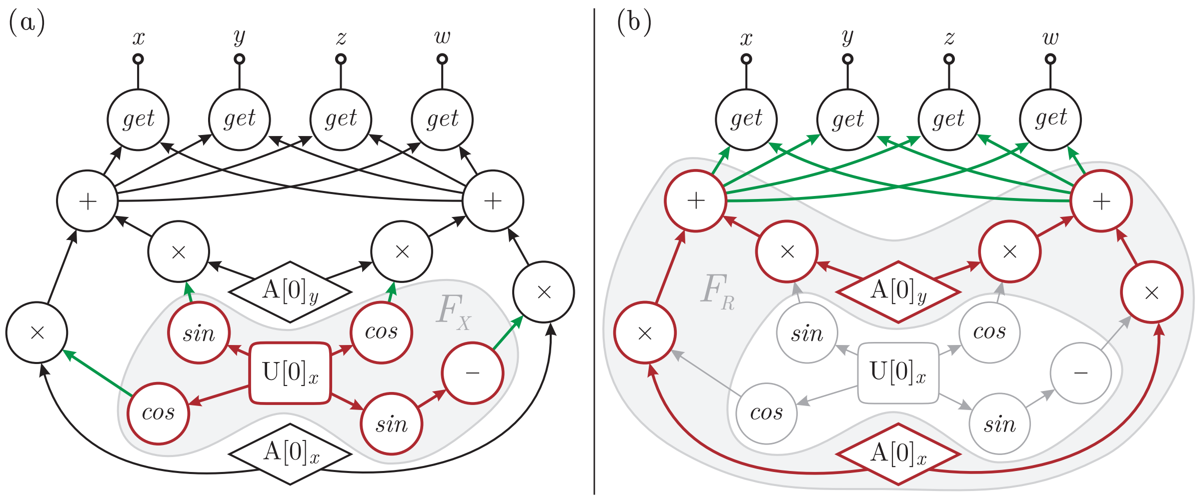


Figure 5.2: The expression DAG from section 4.2.4 is partitioned into the three computation frequencies and processing stages – CPU, vertex unit of the GPU, and fragment unit of the GPU. (a) shows the "uniform transition" edges (green) that separate the sub-DAG relevant to the CPU (red) from the rest (black), while (b) shows the "varying transition" edges (green) that separate the remaining DAG into vertex unit (red) and fragment unit (black).

Transition From the Vertex Unit to the Fragment Unit

Similar to the aforementioned transition from the CPU to the GPU, the GPU^{++} development system supports a transition between the different processing units on the graphics hardware – the so-called *varying transition* indicates the outputs of the vertex unit used as inputs for the fragment unit. Such transitions are affected by the `attribute` qualifier, but again this does not mean that all attributes in an expression DAG necessarily emanate "varying transitions" – similar to uniform transitions in the prior subsection.

Please remember that `attribute` variables are attached to a *region*, which is a "list of array coordinates" that are linearly interpolated over the covered area (section 4.2.3). For instance, the element position `v4pos` is the linear interpolation of the kernel's input region. However, instead of computing the linear interpolation "by hand", the following property of the graphics hardware can be exploited: output values of the vertex unit are *automatically* linearly interpolated in hardware before they are passed as inputs to the fragment unit.

Therefore, a simple transition strategy is to compute all attribute values on the vertex unit and let the GPU apply the linear interpolation. However, this strategy can be enhanced, because some mathematical operations are *invariant* to linear interpolation: For instance, let $L_t(a, b) = a + t(b - a)$ then it follows that $L_t(a_1, b_1) + L_t(a_2, b_2) = L_t(a_1 + a_2, b_1 + b_2)$. In other words, it does not make a difference if the results of two linearly interpolated attributes are added or the sum of the attributes is linearly interpolated – *addition is invariant to linear interpolation*. This property can be exploited: Instead of adding two linear interpolated vertex unit results in the fragment unit, the sum can be directly computed in the vertex unit and the result is passed as input to the fragment unit.

Beside *fully invariant operations* (FIO) – like "addition" and "subtraction" – there are also *partially invariant operations* (PIO). PIOs are invariant if some of the input parameters are constant or uniform. For instance, *multiplication* is PIO, because it becomes invariant to linear interpolation if one of its parameters is constant. Another PIO is "division" that becomes invariant to linear interpolation if the second parameter is constant, but it is not invariant for a constant first parameter. The other PIOs are "linear interpolation", "multiply-and-add", "comparison", "vector dot product" and "vector cross product".

The strategy to find "varying transition" edges needs to keep track of all the aforementioned issues. This works as follows: There are three set of DAG nodes – F_R stores all nodes that belong to the REGION frequency, C_R stores all constant or uniform transition nodes (these are "constant" from the fragment unit's point-of-view) and T_R stores the transition edges. The pseudo-code for the "varying transition" approach works as follows:

Algorithm 5.2: Find the set of transition edges T_R for the REGION frequency F_R

```
$F_R \leftarrow \emptyset$;
$C_R \leftarrow \emptyset$;
$T_R \leftarrow \emptyset$;
forall nodes $n_i$ on a "bottom-to-top" traversal of the expression DAG do
    if $n_i$ = attribute variable then
        $F_R \leftarrow n_i$;
    else if $n_i$ = uniform transition node then
        $F_R \leftarrow n_i$;
        $C_R \leftarrow n_i$;
    else if $n_i$ = operation node then
        if $n_i$ = FIO and $\forall(c_i$ is child of $n_i) : c_i \in F_R$ then
            $F_R \leftarrow n_i$;
        else if $n_i$ = PIO and $\forall(c_i$ is child of $n_i) : c_i \in F_R$ and specific $c_i \in C_R$ then
            $F_R \leftarrow n_i$;
        else
            forall children $c_i$ of node $n_i$, where $c_i \in F_R$ do
                $T_R \leftarrow c_i$;
            end
        end
    end
end
```

Figure 5.2b illustrates the varying transition strategy for the known expression DAG of section 4.2.4. It can be seen, that the complete "coordinate rotation" is performed as part of the REGION frequency on the vertex unit of the GPU, because the used addition is a FIO and the used multiplication is a PIO, where one parameter is a constant or a uniform transition. All "varying transitions" have been detected automatically using algorithm 5.2 and the computed configuration is optimal (i.e. it cannot be further optimized manually).

Chapter 6

Back-End

The expression DAG has been created (chapter 4) and optimized (chapter 5) without any access to the actual graphics hardware. In contrast, the *back-end* compiles the graph to the specific instruction set of the target GPU, i.e. to the shader format of the API.

The back-end design of other GPU-based development systems usually is based on a "monolithic" approach: The internal data structure is passed to the back-end that encapsulates the creation of the back-end-specific output. While this is reasonable for lightweight back-ends of non-optimizing systems (e.g. Sh and the RapidMind development platform) and vectorized approaches (e.g. Cg and GLslang), the back-end stage of the Gpu^{++} development system performs additional *optimizations* on the low-level instruction stream. In this case, a monolithic design leads to massive code duplication for multiple back-ends. Therefore, back-end processing is separated into the following three stages:

Stream Creation – The expression DAG is traversed to create a "stream" of graphics API instructions, where each instruction computes an individual vector component.

Optimization of the Stream – The stream of instructions is automatically optimized, which includes register renaming and the "fusion" of separate vector components.

Transfer Stream to GPU – The optimized stream is used to create a back-end specific output, which is then uploaded and attached to the GPU through the graphics API.

Only the first and last stages depend on the actual graphics API. Therefore, only these stages have to be implemented in the Gpu^{++} back-end, while the stream optimization stage is reused in each back-end. This approach simplifies implementation of new back-ends, because only the following is required: 1) a set of custom *instructions* and *registers* to operate on, 2) the *unit traits for substitution* that have been presented in section 5.2.1, 3) a dedicated *visitor class* to traverse the expression DAG and create the stream of custom instructions, and 4) a thin *communication layer* dedicated to the specific graphics API.

Sections 6.1 through 6.3 correlate to the aforementioned three stages. The OpenGL back-end is used as an exemplar implementation, which is based on the OpenGL low-level/assembler language and therefore is independent of the operating system and the graphics hardware [12]. However, the aforementioned separation facilitates the fast implementation of additional back-ends, like the DirectX low-level shader API.

6.1 Creation of the Instruction Stream

While the expression DAG can efficiently be used for processing (e.g. optimization of the represented expression) it cannot directly be used as input for the "plain" instruction architecture of a processor platform. In other words, the graphics processor expects a sequential stream of instructions, instead of a graph-like structure. Furthermore, the generic node types that are used in the expression DAG usually do not match to the native instruction set of the GPU (or the API used by the back-end). As a consequence, two tasks are crucial for the back-end: First, a custom instruction set has to be defined, and second, a stream of such native instructions needs to be created, using the expression DAG as input.

6.1.1 Custom Instruction Set

The *custom instruction set* directly maps the native GPU-based shader instructions of the graphics API to C^{++} classes. Because the instructions operate on graphics API dependent variables – so-called *registers* – the custom instruction set also includes the mapping of registers. A sequence of instructions is concatenated to build an *instruction stream*.

From the implementational point-of-view, the aforementioned instruction stream is stored in an instance of the template class `cmdstream<CMD,REG>` – where `CMD` represents the base class of all custom instructions and `REG` represents the base class of the register types natively supported by the back-end. Instructions and registers can be of any kind, as long as the following methods have been implemented to incorporate with the Gpu^{++} system:

Vector Fusion – Two methods have to be implemented to support the vector fusion approach of section 6.2.1: First, `CMD::isCompatible()` returns `true` if two instructions can be combined, and second, `CMD::combine()` returns the combined native instruction.

Register Renaming – To support the register renaming technique of section 6.2.2, two further methods have to be implemented: `CMD::registers()` returns a list of used `REG` objects and finally, register renaming is applied using the `CMD:mapRegisters()` method.

By implementing this very "light" interface, the custom instruction set is fully integrated into the back-end architecture, i.e. optimizations are automatically applied to the instruction stream and vector component are fused automatically. In fact, most of the complexity in back-end implementation is wrapped in "`isCompatible()`" for "vector fusion". The most basic approach is to return `false` for all instructions, which disables vector fusion.

Example – The OpenGL Assembly Language

The OpenGL graphics library includes an own powerful string-based high-level shading language: GLslang (see section 2.1.3). However, a high-level shading language performs its own compilation and applies own optimizations, which leads to a sub-optimal overall GPU-based compilation performance. Furthermore, GLslang requires a fourth-generation graphics processor and is not supported on prior hardware. Therefore, the Gpu^{++} back-end is based on the widespread OpenGL assembler language[1], which is accessed through two OpenGL extensions (“`ARB_vertex_program`” and “`ARB_fragment_program`” [8, 9]).

The aforementioned extensions provide a very limited set of instructions, and some of the instructions are not applicable for general-purpose computations or are only partially supported. Table 6.1 shows the set of available OpenGL assembler instructions and whether they are used in the back-end. Some operations are not fully supported on both processing units of the GPU. However, the substitution stage (see section 5.2.1) ensures that such instructions are replaced by equivalent supported instructions.[2]

The assembler instructions are separated into two main categories: *scalar* and *vector* operations. While vector operations compute all scalars of a vector differently, scalar operations perform the same computation to all vector components. Both main categories are further divided into five subcategories: instructions that use one or two scalar inputs (`gl::cmdScalarUnary` and `gl::cmdScalarBinary`) and those who use one, two or three vector inputs (`gl::cmdVectorUnary`, `gl::cmdVectorBinary` and `gl::cmdVectorTernary`). The assembler instruction for a texture fetch (`TEX`) is emulated as an unary vector instruction.

Furthermore, the OpenGL assembler language supports different types of register variables: `gl::regConstant` encapsulates GPU-based compile-time constants, `gl::regParameter` represents “uniform transitions”, `gl::regInputVertex` represents `attribute` variables, “varying transition” are encapsulated in `gl::regInputFragment`, `gl::regOutput` wraps the output of a computation frequency, and `gl::regTemporary` represents intermediate results in the shader program. While the first five register variable types are affected by “type qualifiers” and “frequency transitions”, temporary registers are used to store computation results that stay in the same computation frequency.

There is a missing link between registers and instructions. Instead of raw registers, instructions use “parameters” as their inputs and outputs. In fact, a *parameter* is a register with two (optional) additional information: First, a sign-flag is used for negating the register, and second, a component mapping is provided to swizzle the register components. Additionally, parameters are dependent of the instruction category and whether they are used for input or output, i.e. `gl::paramScalarIn` represents scalar inputs, `gl::paramVectorIn` is used for vector inputs, and `gl::paramVectorOut` represents all results of an instruction.

[1]Please note that OpenGL’s low-level shading language is not identical to the native instruction set of the specific graphics processing unit. The term “assembler language” might be misleading in this case, because the “low-level” shader code is again transformed (which may also include further optimizations).

[2]In fact, such instructions are `SIN`, `COS`, `LRP`, `CMP`, and `TEX` – that are all supported on the fragment unit only. However, the only critical instruction is `LRP`, because it is linear – and therefore may not be shifted to the fragment unit in the frequency transition stage of section 5.2.2. All other commands are non-linear.

		Unary Scalar			
ARL	Address Register Load	V	SIN	Sine	F
COS	Cosine	F	SCS	Sine and Cosine	F
EX2	Base-2 Exponential	V+F	EXP	Base-2 Exponential (approx.)	V
LG2	Base-2 Exponential	V+F	LOG	Base-2 Exponential (approx.)	V
RCP	Reciprocal	V+F	RSQ	Reciprocal Square Root	V+F
		Binary Scalar			
POW	Exponentiate	V+F			
		Unary Vector			
ABS	Absolute Value	V+F	FLR	Floor	V+F
FRC	Fraction	V+F	KIL	Kill Fragment	F
LIT	Light Coefficients	V+F	MOV	Move	V+F
SWZ	Extended Swizzle	V+F			
		Binary Vector			
ADD	Addition	V+F	SUB	Subtraction	V+F
MUL	Multiplication	V+F	DP3	3D Dot Product	V+F
DP4	4D Dot Product	V+F	DPH	Homogeneous Dot Product	V+F
DST	Distance Vector	V+F	MAX	Maximum	V+F
MIN	Minimum	V+F	SGE	Set on Greater or Equal Than	V+F
SLT	Set on Less Than	V+F	XPD	3D Cross Product	V+F
		Ternary Vector			
CMP	Compare	F	LRP	Linear Interpolation	F
MAD	Multiply and Add	V+F			
		Texture Fetch			
TEX	Texture Sample	F	TXB	Texture Sample with Bias	F
TXP	Projected Texture Sample	F			

Table 6.1: The instruction set of OpenGL's assembler language. Most commands work for vertex ("V") and fragment ("F") unit, but some are only supported by one computation frequency. Note that grayed instructions are not used by the Gpu^{++} development system, because of their strong computer graphics focus, such as lighting computations using LIT.

6.1.2 Stream Creation

The instruction stream is created by traversing the expression DAG and converting each node into a single instruction, which then is appended to the stream. This is done individually for each computation frequency: The traversal begins with the output nodes and creates the instruction stream for the ELEMENT frequency (i.e. fragment unit). Whenever a "varying transition" is reached, the instruction stream for the REGION frequency is created. Consequently, the traversal stops at "uniform transition" nodes, because everything below such DAG nodes is computed on-the-fly by the cpu when the kernel program is executed.

Example – OpenGL Peculiarities

The OpenGL assembler back-end uses a dedicated visitor class (`gl::visGenerateCode`) to create instruction streams for the different processing units. However, there are some situations that requires special care.

Registers in the OpenGL assembler language are rather limited, e.g. registers cannot be declared as input and output at the same time, and output registers cannot be used for further computations. To overcome such limitations, the back-end may insert *dummy nodes* into the expression DAG, i.e. nodes that actually do nothing but copying its input to a new output register. Dummy nodes are inserted in one of the following situations:

Negation – The assembler language does not provide a "negation" operation, but all instructions support implicit negation of input registers – like in `MUL r0,r1,-r2` which represents "$r_0 = r_1(-r_2)$". The negation is "for free" in this case, which leads to further optimizations of the kernel program. However, this does not work for output nodes[3]. More precisely, if an output node of the expression DAG represents a "negation", a *dummy node* has to be inserted, which leads to the *simulated negation* "`MOV r0,-r1`".

Bypassing – There are situations, where all kernel computations are done in just one of the two processing units, and the other unit is *bypassed*. For instance, writing the element position to the output array causes only the vertex unit to perform computations. This means that no instruction is performed in the fragment unit and the input register of the fragment program is bypassed to its output register. Unfortunately, in the assembler language, registers cannot be declared as input and output at the same time. Thus, a *dummy node* needs to be inserted again.

Illegal Output Slot – The expression DAG can have up to four output nodes, one for each component of the output vector. This might lead to the situation that one output node is used to compute a second output node. For instance, in the expression "`return vec4(t,t*2,t+1,t-3)`", the output node of the first component is used as input for all other output nodes. Unfortunately, a register in the assembler language that is declared as output, cannot be used as input for subsequent instructions. Therefore, a *dummy node* is required that replaces the first output node.

The aforementioned criteria are checked in the `preOutput()` method of the visitor class, because all problems can be solved by inserting a *dummy node* as a new root node. This is done for output nodes that wrap a *negation*, a *transition*, or an *illegal output*.

A final implementation detail regards the vertex unit: The vertex stage requires some initialization, i.e. the input vertex position is transformed from world to screen space. The required matrix multiplication is automatically inserted in front of the instruction stream.

[3]Remember that expression DAGs do not have unique *root nodes* that represent the final result of a computation, like in an expression tree. Instead, there are up to four *output nodes* that represent the result of each component of the output vector. While at least one of the output nodes is also a root node, there might be other output nodes that are used for further processing and therefore are no root nodes.

6.2 Instruction Stream Optimization

A major difference of the GPU^{++} development system in comparison to other shading languages is that back-end optimizations are done in a generic way, so that they can be reused in all back-ends. As a consequence, the optimizations that are performed on the instruction stream are not dependent on a specific graphics API, but they are dedicated to the graphics hardware architecture in general (e.g. the vector processor concept). Such a generic approach for final optimizations significantly simplifies back-end development.

6.2.1 Vector Fusion

The GPU is a vector processor that performs its computations on up to four vector components in parallel (see section 3.3). However, the expression DAG does not completely reflect this architectural design. Instead, it represents computations on individual scalars and no further "vector component information" is associated to the graph. The advantage is that developers are not forced to deal with the concept of a vector processor, which is one of the major difficulties in understanding the GPU. In other words, an expression DAG that represents a kernel program with massive use of the vector approach (i.e. using vectors with multiple components and utilizing the swizzling technique) can also be created without this concept using only the custom GPU^{++} type "`vec1`" as a replacement of "`float`".

This novel approach differs significantly from all other shading languages, where the vector processor design is an integral part of the internal data structure. While this simplifies the language's design and makes sense in the computer graphics domain, optimal run-time performance can only be achieved by the use of the vector processor paradigm. Developers who are not familiar with this concept are unable to create speed-efficient GPU-based code.

But the aforementioned approach of decoupled vectors only "virtually" eliminates the vector processor paradigm. In fact, the need of "vectorization" is postponed to the back-end, where individual instructions have to be combined to reduce the shader program size. For instance, the sequence "`ADD t2.x,t0.xxxx,t1.xxxx;ADD t4.x,t3.xxxx,t4.xxxx;`" can be combined to a single command "`ADD tC.xy,tA.xyxx,tB.xyxx`" with the according register aliasing (i.e. `tA.x=t0.x`, `tA.y=t3.x`, `tB.x=t1.x`, and `tB.y=t4.x`). This is called *vector fusion*.

Shortest Edit Script

The GPU^{++} development system uses a novel algorithm to fuse vector computations, i.e. combining scalar operations to exploit the vector processor architecture. The approach is based on the problem of "finding the *shortest edit script* for transforming A into B", which means, to find the minimum "script" of symbol deletions and insertions that transform one symbol sequence into the other [123]. For instance, to transform sequence "`ABCABBA`" into sequence "`CBABAC`" the shortest edit script is "$\underline{\mathtt{AB}}\mathtt{C}\overline{\mathtt{B}}\mathtt{AB}\underline{\mathtt{B}}\mathtt{A}\overline{\mathtt{C}}$" (where "$\underline{\mathtt{X}}$" means that `X` has to be *deleted*, "$\overline{\mathtt{X}}$" means that `X` has to be *inserted*, and "`X`" means that `X` has to be *taken over*).

The shortest edit script (SES) problem is most popular to its use in the well-known `diff` tool on the Unix platform. The `diff` program compares two text files line-by-line and outputs the difference. While this usually makes little sense for two independent files, it is of great value for two different versions of the same file. In this case, the shortest edit script represents the most-like editing steps (i.e. the "history") between the two file versions: lines that have been *deleted*, that have been *inserted*, and that have been *taken over*.

The GPU^{++} development system exploits the shortest edit script to combine vector computations as follows: Consider a simple vector processor that operates on two-component vectors of type `vec2`. Furthermore, consider the following procedure for such a processor:

```
1  vec2 foobar(vec2 v2a, vec2 v2b, vec2 v2c)
2  {
3     vec2 v2temp = v2a + v2b.yx();                     // sum two vector using swizzling
4     v2temp.x() = 2.0 * v2temp.x() + 0.5;     // double first component and add offset
5     v2temp.y() = (v2temp.y() + 0.5) * 2.0;         // manipulate the second component
6     return v2temp * v2c.xx();                          // finally, return the product
7  }
```

A decoupling of vector components leads to the following instruction streams: one computes the first component of the resulting vector (`r.x`) and the other computes the second (`r.y`):

```
1  ADD t1, a.x, b.y;            // line 3
2  MUL t2, t1, 2.0;             // line 4
3  ADD t3, t2, 0.5;             // line 4
4
5  MUL r.x, t3, c.x;            // line 6
```

```
1  ADD t1, a.y, b.x;            // line 3
2
3  ADD t2, t1, 0.5;             // line 5
4  MUL t3, t2, 2.0;             // line 5
5  MUL r.y, t3, c.x;            // line 6
```

Concatenating both instruction streams results in a working solution. But the vector properties of the considered processor have not been exploited, because the first, third, and last instruction of both streams can be combined (using appropriate register aliasing).

The idea is: The two instruction streams are considered as two versions of the same stream. Furthermore, instructions of different streams are treated as "equal" whether they can be combined on the vector processor. From this follows that the SES leads to an optimal "combined instruction stream" that contains three types of instructions:

- Instructions of the first stream that cannot be combined (equivalent to "delete").
- Instructions of the second stream that cannot be combined (equivalent to "insert").
- Instructions in both streams that can be combined (equivalent to "take over").

In the next subsections, a fast algorithm is presented that computes the shortest edit script for a generic set of symbols and how it is extended to support a four-component vector architecture. In addition, a heuristic approach is presented that is used in the GPU^{++} development system to perform the vector component fusion in the back-end.

The DIFF4 Algorithm

Myers has shown that computing the shortest edit script can be reduced to finding the shortest path for an *edit graph* [87]. The edit graph for two sequences $A = a_1 a_2 \ldots a_N$ and $B = b_1 b_2 \ldots b_M$ has a vertex at each point (x, y) with $x \in [0, N], y \in [0, M]$. The vertices of the edit graph are connected with directed edges as follows: *Horizontal edges* connect each vertex to its right neighbor (i.e. $(x, y) \rightarrow (x + 1, y)$ for $x \in [0, N - 1]$ and $y \in [0, M]$), and *vertical edges* connect each vertex to the neighbor below it (i.e. $(x, y) \rightarrow (x, y + 1)$ for $x \in [0, N]$ and $y \in [0, M - 1]$). Finally, if $a_{x+1} = b_{y+1}$ then there is a *diagonal edge* connecting vertices $(x, y) \rightarrow (x+1, y+1)$. Vertex $(x+1, y+1)$ is called *match point* in such a case. Figure 6.1 illustrates the edit graph where A and B represent the aforementioned instruction streams (and where "match points" represent combinable instructions).

The problem of finding the SES is equivalent to finding a minimum-cost path from $(0, 0)$ to (N, M), where diagonal edges weight 0 and non-diagonal edges weight 1 [87]. Using the aforementioned SES terminology, horizontal edges represent "delete", vertical edges represent "insert", and diagonal edges represent "take over". Because this algorithm has been initially used to detect the differences between two text files, it is called DIFF2.

However, the algorithm can easily be extended to D instruction streams by using D-dimensional edit graphs and introducing several types of diagonal edges (and according edge weights). The GPU^{++} development system makes use of a four-dimensional version. Therefore, it deals with 15 edge types: one edge that combines all four instruction streams, four edges that combine three instruction streams, six edges that combine two instruction streams, and four edges that represent un-combinable instructions in each stream.

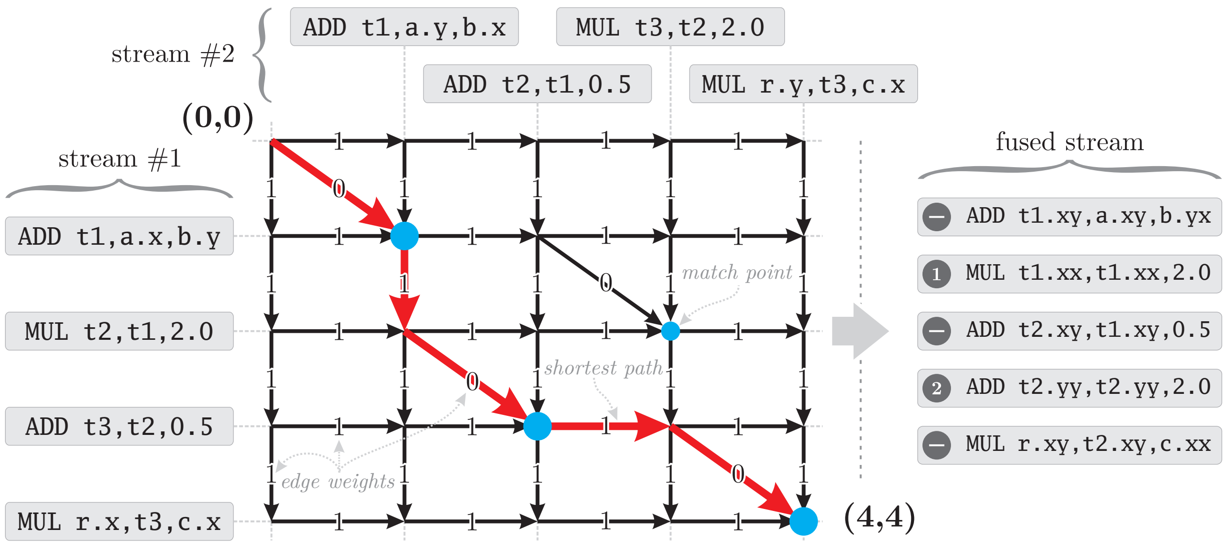


Figure 6.1: The edit graph for the two exemplar instruction streams, each responsible for computing a vector component of result `r`. The minimum-cost path from $(0, 0)$ to $(4, 4)$ represents the optimal *vector fusion* of the instruction streams, which is shown on the right.

Heuristic Extension For Altering Instruction Streams

The described approach works very efficient for separate and fixed input streams, but this might not always be the case: Instructions of a stream usually can be reordered without changing the actual computation. Such expressions are combinable in a "global context", but cannot be detected with the basic algorithm. Furthermore, prior optimizations of the DAG may result in additional "distortions" of the instruction streams. The edit graph can be extended to support the aforementioned issues, however, the complexity of finding the minimum-cost path grows exponential. By assuming that instruction streams of size N can be altered in $O(2^N)$ ways, the computational complexity to find the shortest path grows accordingly.

The following extensions have been integrated into the basic algorithm to create a heuristic DIFF4 version, that leads to efficient results at significantly lower run-time:

- The edge weights in the edit graph are preprocessed: For a small value M (e.g. $M = 4$) all edge-weights are scaled by M, except those edges that are in an edge neighborhood of size M around a *match point*. There, the weights are scaled by the match point distance. Figure 6.2 illustrates this for a DIFF2 edit graph. This re-weighting leads to noticeable earlier termination of the path finding algorithm.
- Consider point (a, b) in a DIFF2 edit graph. The end-point (N_1, N_2) can be reached in at least $|(N_1 - a) - (N_2 - b)|$ and in at most $(N_1 - a) + (N_2 - b)$ steps, due to the aforementioned edit graph structure. These boundaries are used to eliminate inefficient paths at an early stage, leading to an accelerated shortest path computation.
- The altering of instructions in the stream is limited to a range of 8–16 positions.

These extensions reduce the computational complexity of finding the minimum-cost path to $O(N^4 + D)$ in the worst-case (where the "worst-case" means "very few match points").

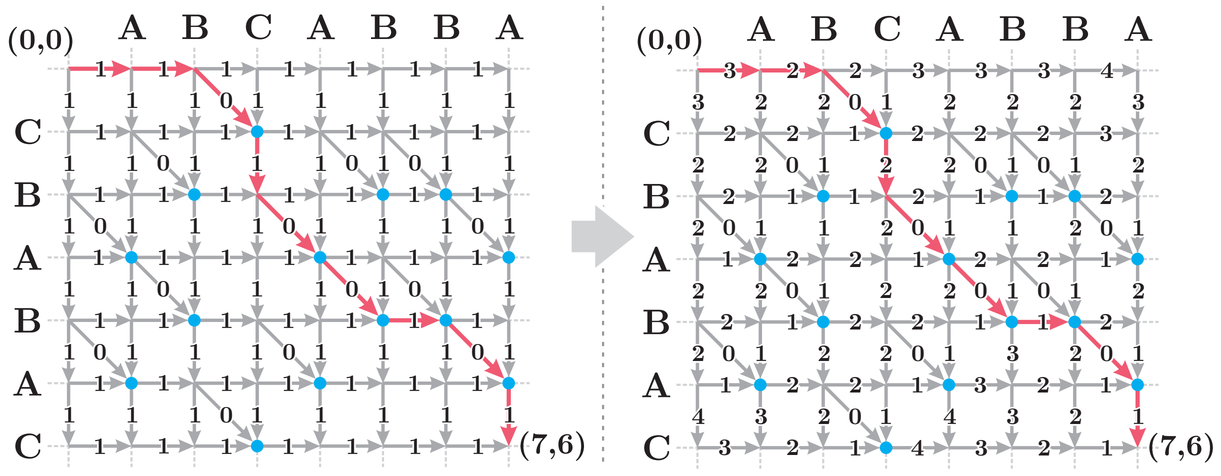


Figure 6.2: A simple DIFF2 edit graph with edge weights relative to the distance of their nearest match point. A neighborhood of $M = 4$ has been used for edge weights computing. Finding the shortest path runs more than four times faster than the non-heuristic approach.

6.2.2 Register Renaming

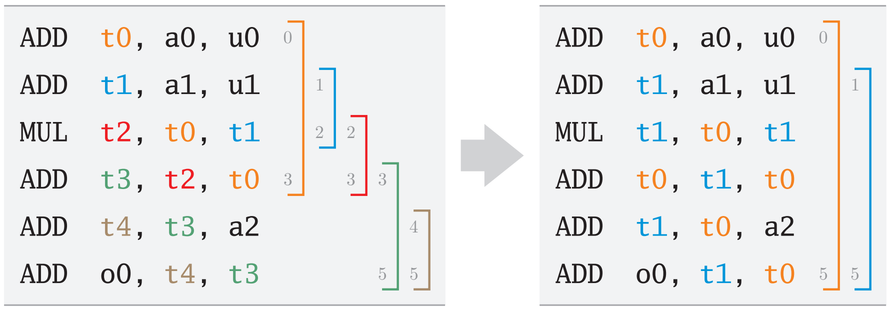
The aforementioned approach for creating the instruction stream generates new temporary registers for each computation result (i.e. each inner node of the expression DAG). While this is a simple and intuitive scheme, it may lead to resource problems, because the amount of such registers in a kernel program is limited.[4] Therefore, the OpenGL assembler language back-end uses "register renaming" to overcome this resource restriction.

Register renaming has been used in the design of parallel and super-scalar processors to avoid "register aliasing", where a specific register is frequently reused in a sequential instruction stream (and therefore restricts the parallel execution of instructions) [108]. This problem can be solved by introducing a "register alias table" (RAT) that maps between "logical" and "physical" registers. Instead of reusing the same register, a new register is introduced whenever possible. Ideally, this improves the parallel execution of instructions.

In fact, back-ends have to deal with the inverse problem, because temporary registers are used independently from each other. While this might be good for parallelism, it wastes a lot of resources. As a consequence, the back-end uses "register renaming" to combine temporary registers in the shader program, using a RAT again. Figure 6.3 shows an exemplar sequence of six OpenGL assembler instructions: Because each instruction (except the last that uses the output register `o0`) introduces a new temporary register to store the result, five registers are used in total. However, the *scope* of each register is very small, and most scopes do not conflict with each other, e.g. the scopes of `t1` and `t4` do not intersect. The register count can be drastically reduced, by using the following RAT:

$$RAT = (\texttt{t0} \mapsto \texttt{t0}, \texttt{t1} \mapsto \texttt{t1}, \texttt{t2} \mapsto \texttt{t1}, \texttt{t3} \mapsto \texttt{t0}, \texttt{t4} \mapsto \texttt{t1})$$

Obviously, the sequence is equal in functionality, but uses only two registers. Please note that this leads to instructions that write to the same registers that are used as their inputs.


```
ADD  t0, a0, u0
ADD  t1, a1, u1
MUL  t2, t0, t1
ADD  t3, t2, t0
ADD  t4, t3, a2
ADD  o0, t4, t3
```

```
ADD  t0, a0, u0
ADD  t1, a1, u1
MUL  t1, t0, t1
ADD  t0, t1, t0
ADD  t1, t0, a2
ADD  o0, t1, t0
```

Figure 6.3: A typical instruction stream created by the OpenGL back-end. Six instructions result in five temporary registers (`t0`, ..., `t4`). The thin brackets on the right illustrate the "scope" of each temporary. Obviously, some of these registers can be reused. In fact, *register renaming* reduces the amount of temporary registers to two (`t0` and `t1`).

[4]Most graphics hardware allows up to 32 temporary registers, which might leads to problems for complex computations. However, the latest generation of GPUs supports up to 4096 temporary registers.

The following algorithm performs *register renaming* for a given sequence of instructions in three steps: First, the scope of each temporary register r_t is computed (lines 2 to 10). Then, the actual mapping is created by reusing registers where the scope does not intersect (lines 11 to 22). Finally, the renaming is applied to all temporary registers (lines 23 to 25):

Algorithm 6.1: Perform *register renaming* to instruction stream I

```
 1 rat ← $\emptyset$;

 2 foreach instruction $i_n$ at position $n$ of the instruction stream $I$ do
 3     foreach temporary register $r_t$ that is used in instruction $i_n$ do
 4         if rat[$r_t$] does not exist then
 5             rat[$r_t$].map  ← $r_t$;
 6             rat[$r_t$].from ← $n$;
 7         end
 8         rat[$r_t$].to ← $n$;
 9     end
10 end

11 $R_d \leftarrow \emptyset$;
12 foreach register $r$ stored in rat do
13     foreach register $r_d \in R_d$ do
14         if rat[$r$].from $\geq$ rat[$r_d$].to or rat[$r$].to $\leq$ rat[$r_d$].from then
15             rat[$r$].map ← $r_d$;
16             rat[$r_d$].from ← min(rat[$r$].from, rat[$r_d$].from);
17             rat[$r_d$].to   ← max(rat[$r$].to, rat[$r_d$].to);
18             break
19         end
20     end
21     $R_d$ ← rat[$r$].map;
22 end

23 foreach instruction $i_n$ at position $n$ of the instruction stream $I$ do
24     remap all temporary registers in $i_n$ according to the mapping in rat;
25 end
```

Please note that *register renaming* does not optimize the speed-performance of the kernel program. Furthermore, the algorithm is very basic, because the instructions are not reordered to achieve better results, e.g. a register that is created in the first instruction and which is not further used for the next 20 instructions can be optimized by restructuring the command sequence. However, more advanced strategies are not yet implemented, because the computational costs increase exponentially for the length of the instruction stream. In practice, the aforementioned approach leads to efficient results.

6.3 Connecting to the GPU

So far, the GPU^{++} back-end architecture created and optimized a stream of instructions (based on the final expression DAG). Only two issues are left: First, the final generation of the back-end specific output format, and second, the upload and activation of all required data, like the shader code, arrays, regions, and uniforms. While the first task is the final step of the GPU-based compilation process (i.e. a conversion of the custom instructions), the second task is the first step of the GPU-based execution process. The major difference is that the compilation process belongs to the kernel program, while the execution process affects all GPU-based objects. As a consequence, a kernel program instance is explicitly bound to a specific back-end, while instances of arrays, regions, and uniforms can be assigned to multiple back-ends each time they are executed on a different kernel program.

This approach is called *late binding* and allows the same array, region, and uniform to be seamlessly used in different kernel programs (that use different back-ends). As a matter of fact, "late binding" leads directly to other features, such as delayed allocation, shared memory, and lazy up- and downloading of arrays and regions (see section 4.2.2).

6.3.1 Back-End Stubs

Instead of using a large monolithic interface that encapsulates all back-end-dependent tasks, the binding to the graphics hardware is realized using *back-end stubs*: class objects with light interfaces that are responsible for small aspects of the kernel program parameters:

Kernel Stubs

The *kernel stub* is the main stub of the back-end and bears all kernel-related tasks, including the aforementioned creation and uploading of instruction streams (by calling `compile()`), as well as its activation and deactivation, together with the evaluation of uniform variables and the transfer of regions to the GPU (via `execute()`). Furthermore, there are tool methods that can be used to manage the internal indices for uniform and attribute variables.

As aforementioned, the kernel program is different from arrays, regions, and uniforms, because of its explicit back-end binding and because it is used in both – compilation and execution. This significantly affects the design of the kernel program stub.

Array Stubs

In brief, *array stubs* are responsible for all `array`-related tasks. While this sounds complex, it actually reduces to the implementation of the following four methods: 1) `upload()` and 2) `download()` are responsible for the data transfer between the main and graphics memory. Furthermore, 3) `attach()` and 4) `detach()` are responsible for activating and deactivating the array as one of the input arrays or as the exclusive output array of the kernel program.

The array stub of the OpenGL back-end is a thin wrapper for `EXT_framebuffer_object` [11]. Note that the stub is not responsible for advanced features, like delayed up- and downloading or caching. This is done automatically by the Gpu^{++} development system.

Region Stubs

Similar to array stubs, *region stubs* handle the GPU-based communication for `region` objects. The interface is identical to the aforementioned array stubs, except that there is no way to download a region. Please note that the region stub is responsible also for the attribute variable it is assigned to. However, from an implementational point-of-view, attributes are just placeholders for regions.

While arrays are represented by textures and frame-buffer objects, regions are represented by “vertex-buffer objects”. Therefore, this stub is a very thin wrapper for the OpenGL extension `ARB_vertex_buffer_object` [10].

6.3.2 Low-Level API Interface

The Gpu^{++} development system has been designed for GPU-based programming without being an expert of the underlying graphics API. However, there might be situations, where the graphics API and its direct access are of interest, e.g. if the results of a kernel program should be visualized. While it might be intuitive to use the `array` object’s `getArray()` method as the input for a texture upload, it obviously is not efficient, because this causes two data transfers. First, from GPU to CPU, and directly after, from CPU to GPU. A much better way is to directly access the internal texture that represents the array.

This is done via the static `opengl::arrayToTexture()` function that requires a pointer to the array as its input and returns the OpenGL texture id. This texture id depends on the internal Gpu^{++} rendering context, which is accessible via `opengl::getRenderingContext()`. If the rendering context is “shared” with the application’s rendering context, all texture ids (that are returned by Gpu^{++}) can be used directly for further OpenGL processing. This approach allows the embedded creation of procedural textures with Gpu^{++} that can be attached to polygons and objects in a three-dimensional computer graphics scene.

Furthermore, there are similar static functions for accessing vertex buffer objects (i.e. `regionToVertexBuffer()`), vertex and fragment shaders (i.e. `kernelToShader()`), uniform variables (i.e. `uniformToId()`), and attribute variables (i.e. `attributeToId()`).

Chapter 7

Evaluation

While the previous chapters have presented detailed conceptional and technical descriptions of the GPU^{++} development system, this chapter evaluates the efficiency of GPU^{++} using real-life examples: A representative set of general-purpose algorithms from various domains is implemented on both, CPU and GPU, to compare their performance. The computations range from simple image processing techniques and data management to more complex Fourier analysis and tomographic reconstruction. These algorithms are evaluated concerning the following two criteria:

Speed – The major goal of every approach that uses the graphics accelerator is a significant performance gain compared to the CPU-based implementation. This is evaluated by processing two- and three-dimensional data sets of various sizes and measuring the processing run-time. Please note that these timings *exclude* any data transfer between the main and the graphics processor (see section 8.1 for a discussion).

Complexity – In contrast to speed, a quantitative measurement of the software complexity is hard to achieve: Various concurrent software metrics exist, like cyclomatic complexity, function point analysis and code coverage, but most of these code metrics have been designed for large software systems and are not feasible for measuring the complexity of source code with less than thousand lines [78, 115, 21]. Therefore, due to its simplicity, *logical source lines of code* (logical SLOC) are used to evaluate the complexity of all GPU- and CPU-based implementations in this chapter [94]

Please note that all algorithms have been implemented on the GPU before [85, 131, 39, 114, 64, 44, 43, 47]. While GPU^{++}-based solutions do not intentionally compete with such approaches, their speed-performance is similar and their complexity is significantly lower.

The following testbed has been used for all benchmarks: A personal computer, equipped with two AMD Opteron Dual Core 275 64-Bit processors running at 2.2 GHz and provided with 16 GiBytes of memory. Furthermore, the graphics hardware has been NVIDIA's GEFORCEFX 8800 GTX with 575 MHz core speed and provided with 768 MiBytes of 900 MHz clocked DDR3-memory. The operating system has been Microsoft Windows XP x64 edition, but all GPU^{++}-based executables have been compiled for a 32-Bit platform.

7.1 Medical Image Processing

Using graphics hardware to accelerate image processing in the medical field has become very popular [71, 107, 95, 130, 114]. To evaluate the GPU^{++} development system in context of this domain, a very simple but common medical processing pipeline will be implemented: First, a medical image will be partitioned using *threshold segmentation* in section 7.1.1. Then, in section 7.1.2, noise artifacts are removed using an *island removal* approach. Finally, the image is processed via *gaussian smoothing* that creates softer borders, in section 7.1.3. Performance and complexity results are further discussed in section 7.1.4.

7.1.1 Threshold Segmentation

Intensity-based thresholding is the simplest, but yet often most effective, segmentation technique that identifies anatomical structures in the image by comparing their intensity values to one or more thresholds – this is illustrated in figure 7.1 [41]. Such an approach can be used for bone segmentation in CT images, e.g. segmentation of teeth in dental planning. Please note that more advanced techniques were implemented on the GPU, such as *level sets* and *nonlinear diffusion*, but high-end segmentation is not the focus of this evaluation [71, 107]. Hadwiger et al. present an extensive overview of state-of-the-art segmentation approaches and possible implementations on graphics hardware [48].

Threshold segmentation perfectly fits into the streaming concept, because the same operation (i.e. `threshold()`) is performed on all array elements. However, the main processor also benefits from the sequential structure of this approach. Nevertheless, table 7.2 (at the end of this section) illustrates that the GPU is increasingly faster than the main processor, beginning with an image size of 512×512 and images of size 2048×2048 are processed faster by more than an order of magnitude. While this only leads to a speed-improvement of less than ten milliseconds for two-dimensional images, the performance gap increases to half a minute for volumes of size 2048^3. Please note that the GPU-based implementation (8 lSLOC) is comparable in source code complexity to the CPU-based solution (9 lSLOC).

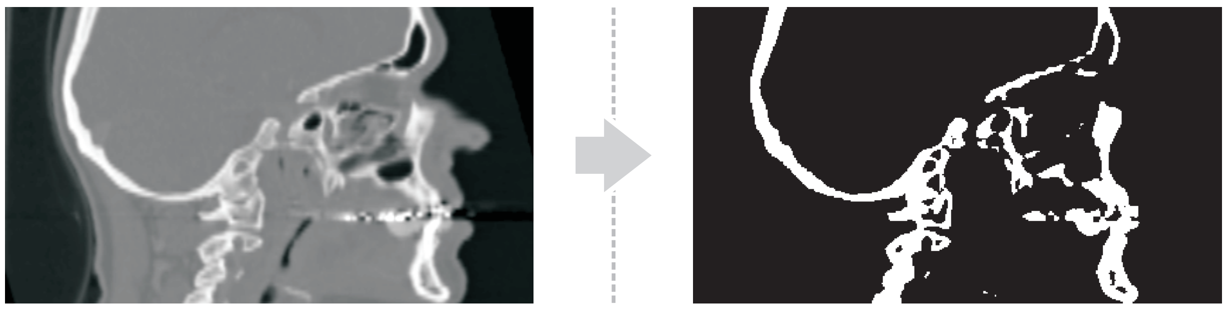


Figure 7.1: A slice through a human skull, acquired by a CT scanner (scalars are unified to the range $[0 \ldots 1]$. Bone structures have been segmented using a threshold of $t = 0.58$. Please note the artifacts, near the nasal bone, that are caused by the noise in the data set.

7.1.2 Pixel-Island Removal

By analyzing the segmentation result of section 7.1.1, artifacts caused by *image noise* are noticeable, leading to so-called *pixel islands* (see figure 7.2a). Such artifacts may accumulate within the medical processing pipeline, leading to sub-optimal results in later stages, e.g. registration. A morphological filter is used to remove these artifacts [41].

A common approach to remove pixel islands uses n “erosion operations” followed by n “dilation operations” for small N. The *erosion* causes objects to shrink, while the *dilation* causes objects to grow in size. The amount and way of growing and shrinking depends on the choice of the *structuring element*, which consists of a pattern that specifies the coordinates of some discrete points relative to its origin. Typical structuring elements are N_4 (taking five pixels into account) and N_8 (using nine pixels). The difference between both operations is, that the *erosion* operator requires *all* pixels of the structuring element to be set (`AND`), while the *dilation* operator requires at least *one* pixel to be set (`OR`).

Because the Gpu^{++} system does not support boolean operators, they are emulated by defining the boolean value “`false`” as 0.0, “`true`” as 1.0, and using `threshold()` again:

$$\begin{array}{lll} \texttt{AND:} & a_1 \wedge a_2 \wedge \cdots \wedge a_n & \Rightarrow \quad a_1 + a_2 + \cdots + a_n \geq n \\ \texttt{OR:} & a_1 \vee a_2 \vee \cdots \vee a_n & \Rightarrow \quad a_1 + a_2 + \cdots + a_n \geq 1 \end{array}$$

The limiting factor of this approach is the excessive access of the input array. For instance, N_4 requires *five* reads and N_8 accesses the memory *nine* times. Furthermore, the common three-dimensional structuring elements are N_6 (with *seven* array access) and N_{26} (that reads memory *twenty-seven* times). This results in a significant run-time slowdown.

The memory access performance can be drastically improved, using a novel approach that exploits the *linear interpolation* feature of the GPU (see section 3.5). Generally speaking, linear interpolation reads multiple values with a single array access instruction, but the intensity values are returned as a weighted sum. While this does not reduce the amount of memory that is actually accessed, hardwired linear interpolation is much faster than sequential array access in the kernel program due to its more efficient cache handling. Consequently, the amount of array access operations can be reduced by finding the “minimal coverage” of the structuring element and carefully choosing the pixel distances.

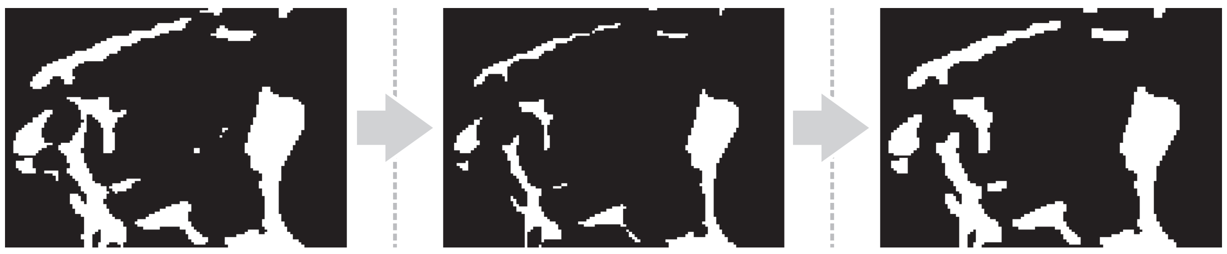


Figure 7.2: Left: Close-Up of the segmented *maxillary bone*. Center: After erosion. Right: After dilation. Islands are removed, but, some thin anatomical structure are also eliminated.

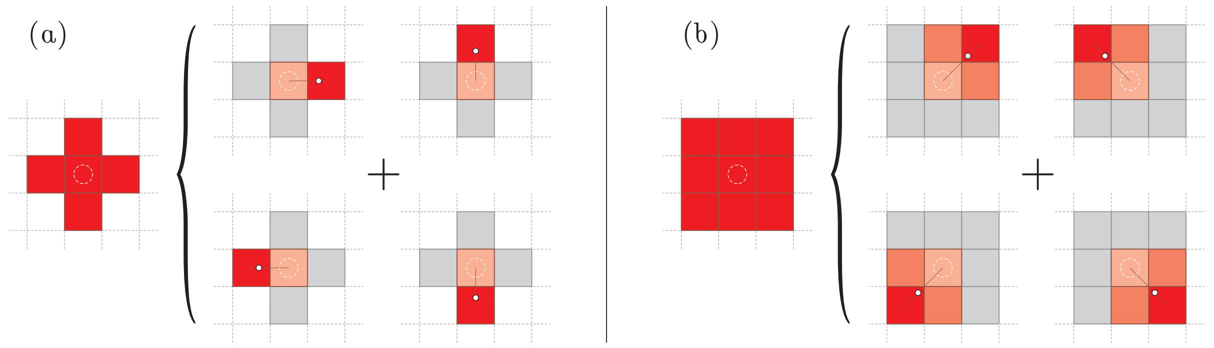


Figure 7.3: Linear interpolation returns the weighted sum of multiple array elements, which can be exploited to reduce the amount of array reads: (a) uses distance $d = 4/5$ along the principal axis for N_4, and (b) uses distance $d = 2/3$ along the diagonals for N_8.

Figure 7.3a illustrates the optimization strategy for structuring element N_4, where each array access returns the weighted sum of two intensity values. However, this also means that the center pixel is read four times as often as the pixels next to the center, i.e. the minimal coverage contains four interpolations, all including the center pixel. The optimal distance d and the constant threshold t are therefore defined by the following set of equations:

$$\begin{aligned} 4 \times (1 - d) &= t \qquad \text{(the center pixel is read four times)} \\ d &= t \qquad \text{(the pixels next to the center are read once)} \end{aligned}$$

The solution is $d = t = 0.8$, i.e. if all distances along the principal axis are $d = 0.8$ relative to the center point, the sum of the four interpolations is $t = 0.8$ for a single set point, and $5 \times t = 4$ for all points of the structuring element N_4 set. In fact, these are the `threshold()` values for erosion and dilation that have to be used in the kernel program.

Figure 7.3b illustrates the same optimization for structuring element N_8. This time, each interpolation is the sum of four pixels, due to diagonal orientation, which leads to $d = 2/3$ and $t = 4/9$ as the solution. For the three-dimensional version of the morphological operations, $d = t = 6/7$ is the solution for N_6, and $d = 2/3$ ($t = 8/27$) works for N_{26}.

The speed results in table 7.2 are benchmarks of a complete island removal iteration, i.e. the morphological erosion operator followed by the dilation operator, both using the structural element N_8. While the CPU- and GPU-based implementations are comparable in speed for a 256×256 array, the graphics processing unit performs much better for larger inputs – the GPU-based island removal processes a 1024×1024 array more than 12 times as fast as the main processor. The performance gap for three-dimensional arrays is even larger – a 1024^3 volume is processed in less than two seconds on the GPU, which is nearly 20 faster than the CPU-based implementation (using N_{28}). However, the source code of the GPU-based solution (16 lSLOC) is noticeably less complex than for the CPU-based approach (27 lSLOC). A comparable GLslang-based implementation takes 84 lSLOC.

7.1.3 Gaussian Smoothing

At this point of the medical pipeline, some artifacts are still left in the data set, e.g. noisy input data usually leads to rough borders of the segmented anatomical structures. Especially the lighting computations in visualization suffer from a coarse surface [127, 74]. There are various approaches to *smooth* two- or three-dimensional data, that all use the same idea: Each input pixel is scattered over a specific neighborhood in the output image. Or the other way around: Each output pixel is a weighted sum of gathered input data. However, while this reduces noise it also "blurs out" fine details of the original data [75].

A popular approach is *Gaussian smoothing* (sometimes also called "Gaussian blur") where the quantifiers of the weighted sum are based on the normal distribution [41]:

$$G_1(r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-r^2/(2\sigma^2)} \tag{7.1}$$

Here, r is the distance to the output pixel and σ is the *standard deviation*. In theory, the normal distribution for every distance r will be non-zero, i.e. the entire input image is included in the computations for a single output pixel. However, when computing a discrete approximation of the normal distribution, pixels outside of approximately 3σ are small enough to be considered effectively zero. For instance, using $\sigma = 0.6$ for smoothing a two-dimensional array effectively results in 13 relevant input pixels within a 5×5 neighborhood.

The memory access can be reduced even further by exploiting the symmetry property of the normal distribution: the n-dimensional weighting filter can be separated into n one-dimensional filters, each using the nth dimension as its primary direction. This means that a two-dimensional Gaussian smoothing is done by performing a *horizontal*, followed by a *vertical* one-dimensional Gaussian smooth, i.e. $G_2(u, v) = G_1(u)G_1(v)$. By using this relationship, the amount of memory reads for $\sigma = 0.6$ is reduced to $5 \times 2 = 10$.

Implementation Details and Results

The implementations for the main and graphics processor, that have been used for runtime measurements, are based on multiple one-dimensional normal distributions that use $\sigma = 0.95$ (which equals to seven neighboring pixels) that are combined to perform two- and three-dimensional smoothing of arbitrary input data. Each "smoothing pass" uses an own kernel instance that differs in the principal processing axis.

As in section 7.1.2, the novel linear interpolation approach is used to reduce the amount of explicit memory reads. This works as follows: As part of the smoothing process, two neighboring input pixels P_1 and P_2 are weighting (with w_1 and w_2) and summed together, i.e. $S = w_1P_1 + w_2P_2$. The same computation can be "simulated" via linear interpolation by using an appropriate distance d between both pixels and an overall factor f, i.e. $f((1 - d)P_1 + dP_2) \equiv S$. This set of equations can be solved to:

$$d = \frac{w_2}{w_1 + w_2} \qquad \text{and} \qquad f = w_1 + w_2$$

	$r=-3$	$r=-2$	$r=-1$	$r=\pm0$	$r=+1$	$r=+2$	$r=+3$
$G_1(r)$	0.002868	0.045788	0.241312	0.419939	0.241312	0.045788	0.002868
d	0.941038		0.465273		0.534727	0.058962	
f	0.048658		0.451282		0.451282	0.048658	
$d+r$	-2.058962		-0.534727		+0.534727	+2.058962	

Table 7.1: The seven weightings of a one-dimensional Gaussian smooth G_1 with $\sigma = 0.95$, together with the distance d and factor f for linear interpolation. Note, that the last row ("$d + r$") shows the distance in respect to the center point of the Gaussian's filter kernel.

Table 7.1 shows the original and the "optimized" weightings of the one-dimensional Gaussian smoothing with $\sigma = 0.95$. The seven memory reads are reduced to four accesses using this technique (in general, n reads are always reduced to $(n + 1)/2$ reads). Please note that the center point is used in two interpolated reads, i.e. the weighting is halved for each access. The second and third row of table 7.1 show the resulting values for d and f, while the fourth row shows the relative distances to the center. These values are used in the GPU-based implementation. Obviously, this optimization technique does not work for the main processor, due to the missing natively supported linear interpolation.

The benchmark results are presented in table 7.2. It can be seen that the GPU-based implementation exceeds the CPU-based solution even for small data sets of 256×256 pixels, where the optimized version runs more than four times faster. Arrays of size 2048×2048 are smoothed nearly two orders of magnitude faster on the GPU. The gap increases even faster for three-dimensional data sets. However, this speedup does not come for the price of a higher source code complexity: The solution for the graphics processing unit (15 lSLOC) is only half as long as the CPU-based implementation (29 lSLOC).

7.1.4 Results and Discussion

The benchmark results of the previously presented implementations are shown in table 7.2. Each implementation (i.e. CPU-based, basic GPU-based and optimized GPU-based) of the different pipeline stages, as well as the full pipeline, is speed-measured for various two- and three-dimensional data sets. It can be seen that – with one exception – the GPU-based implementation is (significantly) faster than the equivalent solution on the main processor.

Furthermore, it can be observed that the graphics hardware seems to scale much slower than the main processor, i.e. four times as much data does not automatically result in a fourth of the speed-performance. This leads to the assumption, that the GPU is limited by a high setup-time for small data sets, that neglects more and more for larger input data. For instance, the GPU-based two-dimensional threshold segmentation seems to have a setup-time of approx. 38 milliseconds. Subtracting this bias from the four speed measurements leads to the actual "pure CPU processing time": 0.02 msec for 256^2, 0.04 msec for 512^2, 0.22 msec for 1024^2, and 0.96 msec for 2048^2 – as expected, these values scale quadratically.

Algorithm		**Plane** (in msec)				**Volume** (in sec)		
		256^2	512^2	1024^2	2048^2	256^3	512^3	1024^3
Segmentation	CPU	0.19	0.76	3.20	12.64	0.05	0.39	3.28
	GPU	0.40	0.42	0.60	1.34	0.10	0.22	0.61
Island Removal	CPU	0.76	3.12	12.48	49.44	0.56	4.63	37.06
	GPU	0.75	0.84	1.43	4.63	0.48	1.08	3.66
	GPU'	0.75	0.78	1.01	2.50	0.36	0.76	1.97
Smoothing	CPU	3.30	13.72	55.23	220.6	1.27	10.54	84.83
	GPU	0.76	0.82	1.20	3.67	0.29	0.63	1.84
	GPU'	0.75	0.77	0.97	2.54	0.29	0.59	1.49
Full Pipeline	CPU	4.25	17.6	70.91	282.7	1.88	15.56	125.2
	GPU'	1.91	1.97	2.58	6.38	0.75	1.74	4.07

Table 7.2: The speed-performance of the basic medical algorithms that have been presented in this section. **CPU** means "CPU-based implementation", **GPU** represents the "basic GPU-based solution" and **GPU'** means "*optimized* GPU-based implementation". Please note that the speed measurements for most implementation do not depend on the data set content, but solely on the data set size. The fastest implementation for each size is marked in green color.

While the presented algorithms are very basic and represent only a small fraction of the techniques that are used in the medical domain, they impart a good feeling how graphics hardware can help to speed up medical processing. Furthermore, this section has illustrated that the GPU^{++} development system allows such speed-improvements without the necessity of an increase in source code complexity: While the GPU outperforms the main processor in all presented cases, the actual implementations have been of similar source code size.

An additional lesson that can be learned is that a concatenation of multiple GPU-based algorithms may lead to a speed-improvement that justifies the data transfer between the main and the graphics processor. While this topic is further discussed in section 8.1, it should be pointed out that the transfer of a 512^3 volume (upload to GPU memory and read-back to main memory) takes less than 700 milliseconds, which is negligible when compared to the 15 seconds processing time of the CPU-based solution.

A final note about the input data sets: Because the speed-performance of all the GPU-based implementations does not depend on the data set content, but solely on the data set size, table 7.2 does include various sizes but *not* various data sets. The same is true for CPU-based threshold segmentation and Gaussian smoothing. But in theory, the data set content may influence the speed-performance of the CPU-based island removal. In practice, however, the detected processing time variances, for different data sets of the same size, are below the measurement error (i.e. less than 10 microseconds).

7.2 Multiplication of Dense Matrices

Beside other tasks, graphics hardware has been initially designed to accelerate the transformation of three-dimensional objects, like resize, rotate, and translate them. As a consequence, the GPU natively features linear operations, e.g. matrix and vector multiplications. However, this is only supported for matrices and vectors of a size of up to 4×4. While this is sufficient for computer graphics, it is inadequate for other disciplines, such as simulation.

The "single-precision general matrix multiplication" (SGEMM) is a fundamental operation of the "Basic Linear Algebra Subprograms" (BLAS) package and is extensively used in linear algebra [31]. GPU-based SGEMM was first implemented by Larsen et al. [70]. The approach has been improved by Galoppo et al. and Govindaraju et al., who exploited the vector architecture and used a different memory model [39, 44]. A different approach has been implemented by Bolz et al. to compute the multiplication of *sparse* matrices [15]. It should be noted that some of the most efficient CPU-based algorithms for dense matrix multiplication are not yet implemented on the graphics processor, like the one from Strassen or the approach by Coppersmith and Winograd [110, 26]. An extensive discussion of the limits of GPU-based SGEMM has been presented by Fatahalian et al. [33].

The GPU^{++}-based implementation for dense matrix multiplication is based on the aforementioned algorithm by Larsen et al. [70]. For two matrices – $p \times m$ matrix A and $n \times p$ matrix B – the $n \times m$ product $(AB)_{ij} = \sum_{k=0}^{p-1} a_{ik} b_{kj}$ is computed as follows: For each $r = 0, \ldots, p-1$, the algorithm *virtually* replicates the rth column of matrix A and the rth row of matrix B over the size $n \times m$ to create the intermediate matrices A_r and B_r. Both matrices are then multiplied component-wise to create matrix M_r. Finally, the sum of all matrices M_r leads to the matrix-product of A and B: $AB = \sum_{r=0}^{p-1} M_r$. This approach is illustrated for two 4×4 matrices in figure 7.4. In fact, the approach is equal to the classic "three-nested-loops", but the most inner loop has been swapped with the most outer loop.

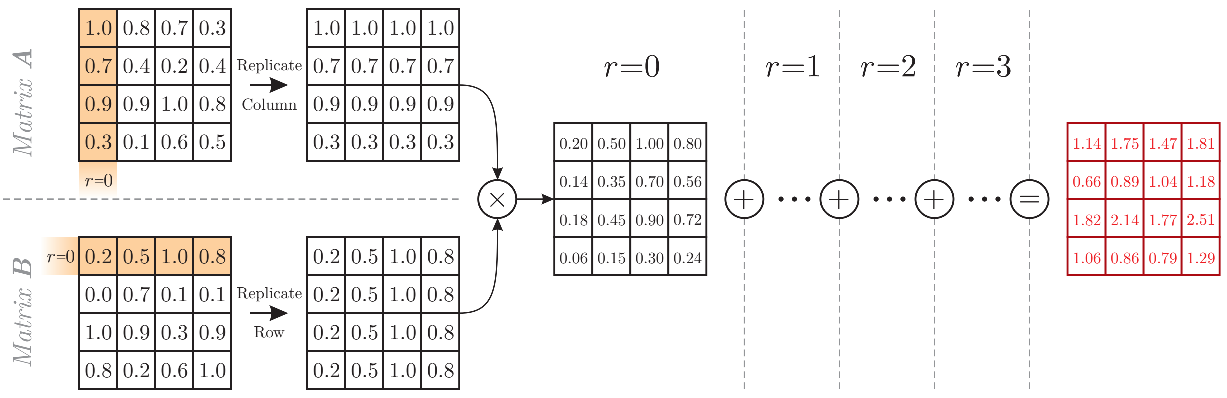


Figure 7.4: The rth column of matrix A and the rth row of matrix B are replicated across the output region, which is realized by manipulating the element position. For each output array element the component-wise product of the replications is added to the output array.

7.2.1 Implementation Details and Results

The column/row replication, component-wise multiplication, and final sum can be directly implemented on the GPU: For each output element at position `vec2(i,j)`, the inputs at `A[vec2(i,r)]` and `B[vec2(r,j)]` are multiplied together and added to the result of prior iterations. In fact, the same kernel program is called p times with a different r for each call. Unfortunately, the intermediate result cannot directly be added to the output content without saturation of the result [12]. Therefore, the *ping-pong* technique is used, where two output arrays are swapped after each iteration [101]. This leads to a major bottleneck, due to the massive memory access for writing the intermediate result in iteration r and reading it back in iteration $r + 1$. As a consequence, the GPU-based implementation can be significantly improved by processing multiple rows/columns in the same kernel pass.

Table 7.3 presents the speed-performance results for the multiplication of two dense matrices that are filled with random values in the range $[0 \ldots 1]$. For a better comparison of the source code complexity, both implementations use the aforementioned *three-nested-loops* approach. Multiple GPU-based benchmarks are presented for the different parallelization factors (i.e. the number of rows/columns processed at each kernel pass). While the computational results are the same, the run-times scale nearly linear, i.e twice the parallelization factor means twice the speed. However, a saturation is noticeable for a scalarity of 64 rows/columns. This depends on the memory bandwidth and supported kernel program size of the actual graphics processing unit.

The GPU-based implementation performs the SGEMM of small 256^2 matrices more than 50 times faster than the solution on the main-processor. This speed-gain increases up to more than two orders of magnitude for the multiplication of 2048^2 matrices. In fact, this is even more than 15 times faster than the fastest known CPU-based implementation [124]. Nevertheless, the GPU-based implementation (21 lSLOC) is just slightly more complex than the CPU solution (16 lSLOC), and it is significantly less complex than other implementations on the GPU – like the one from Wu and Liu (105 lSLOC) [129].

Size	**CPU**	**GPU**				
		$\times1$	$\times4$	$\times16$	$\times64$	Error
256×256	124.74	89.65	22.60	5.93	2.40	1.6×10^{-11}
512×512	1095.55	178.98	45.21	16.78	15.60	6.1×10^{-11}
1024×1024	35313.06	531.00	153.59	123.77	120.19	2.1×10^{-10}
2048×2048	304383.69	4228.66	1230.65	977.77	951.94	5.3×10^{-10}

Table 7.3: Speed-performance for dense matrix multiplication using a CPU-based and a GPU-based implementation (run-time is given in milliseconds). Furthermore, the GPU-based solution has been measured using different *parallelization factors*, from $\times1$ to $\times64$. Please note that the computational error for the GPU (for all parallelization factors) is less than a billionth for random input values between 0 and 1, compared to the CPU-based solution.

7.3 Sorting

Sorting a set of values is challenging on the GPU, because most popular algorithms (like Mergesort, Quicksort or Heapsort) use a *comparison sorting approach*, i.e. the algorithm's workflow depends on the content [67, 51, 126]. In other words, the result of a comparison affects the values that are used in subsequent comparisons. However, this cannot efficiently be implemented on the graphics processing unit – see section 3.6.

In contrast, a *sorting network* consists of hardwired "comparators" that sort two values (see figure 7.5a) [6]. The main difference to the aforementioned approach is that the sequence of comparisons is set *in advance*, i.e. the sorting network stays the same for different input. One of the fastest networks is based on "bitonic sequences".

7.3.1 Bitonic Sorting

A sequence of values is called *bitonic*, if it contains at most one change between ascending and descending. The *comparator module* B_n is defined as $B_n = [0 \mapsto n/2], [1 \mapsto n/2+1], \dots, [n/2-1 \mapsto n-1]$. Applying B_n to the bitonic sequence $a = a_0, \dots, a_{n-1}$ results in $B_n(a) = b_0, \dots, b_{n/2-1}, c_0, \dots, c_{n/2-1}$, where all $b_i \le c_i$ and b, c are both bitonic again [67].

Comparator module B_n is used to build the *bitonic sorting network*: First, a "merge module" is designed to sort arbitrary bitonic sequences by recursively applying B_n to the input sequence (illustrated in figure 7.5). Then, this merge module itself is recursively used to create the final sorting network that is able to sort arbitrary sequences (see figure 7.5c).

Figure 7.6 illustrates a full bitonic sorting network for $n = 8$. The complexity of this approach is $O(n \log^2 n)$ for a sequence of n values, which is not as good as Quicksort's $O(n \log n)$, but its content-independent structure fits much better to a stream processor. GPU-based bitonic sorting has been implemented before by Purcell et al. and Kipfer et al. [100, 64]. Furthermore, Govindaraju et al. have presented a cache-optimized version and Greß et al. implemented adaptive bitonic sorting with complexity $O(n \log n)$ [43, 47]. But these implementations are much more complex than the Gpu^{++}-based solution.

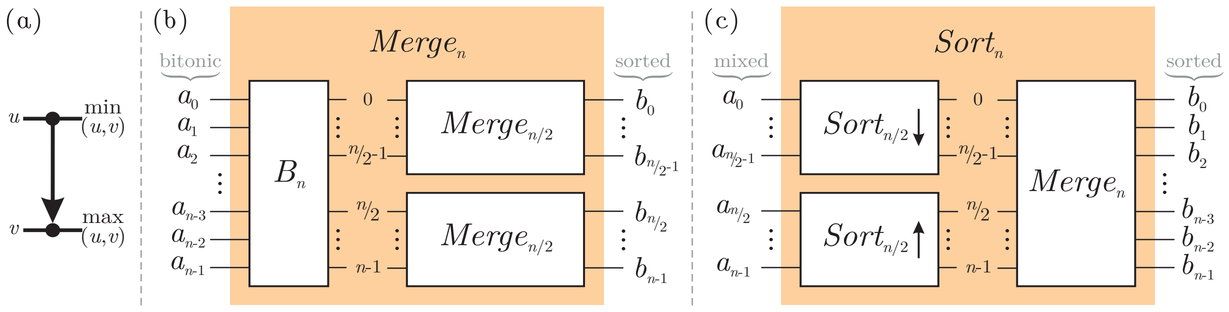


Figure 7.5: (a) The graphical representation of a comparator that sorts two input values u and v. (b) Recursive design of a *bitonic merge* module that sorts a bitonic input sequence. (c) Design of a *bitonic sort* module that recursively sorts an arbitrary input sequence.

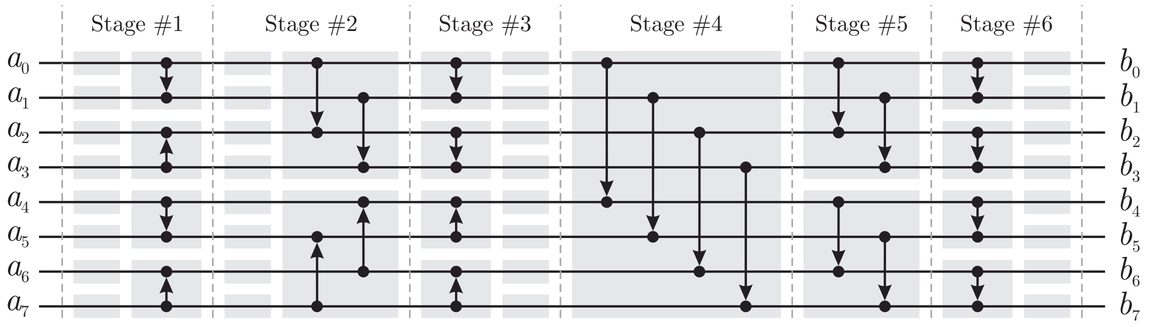


Figure 7.6: A bitonic sorting network to sort an arbitrary sequence of $n = 8$ elements. Note that comparators in the same stage are executed in parallel. There are $\log(n)(\log(n)+1)/2 = 6$ stages, with $n/2$ comparators in each stage. Gray boxes represent comparator modules B_n.

7.3.2 Implementation Details and Results

To improve the speed-performance of the sorting network, the input sequence is interpreted as a two-dimensional array and spatial coherence is exploited. However, this leads to a more complex network construction and nearly triples the source code size (110 instead of 42 lSLOC). Nevertheless, beside the performance gain of 40%, this optimization also relaxes the maximum array size that is supported by the GPU – while one-dimensional arrays allow a maximum of 8192 values, two-dimensional arrays handle more than 67 million values.

Table 7.4 shows the results for sorting sequences of varying size (where each sequence element is defined by a 16-bit *sort key* and 16-bit *attached data*). A CPU-based QUICKSORT algorithm is compared to the presented GPU^{++}-based bitonic sorting network. The CPU-based implementation works much faster for short sequences, due to the high setup-time of the graphics processor. However, beginning with a sequence size of more than 200 thousand values, the GPU outperforms the CPU significantly, and sorting four million values is faster by three orders of magnitude. Nevertheless, while the code complexity of the GPU-based implementation (110 lSLOC) is higher than for the CPU-based solution (35 lSLOC), it still is significantly lower than for other GPU-based approaches, e.g. Kipfer (597 lSLOC) [63].

Size	**CPU**		**GPU**	
	ms	mval/sec	ms	mval/sec
$256 \times 256 = 65536$	18.39	3.56	110.30	0.59
$512 \times 512 = 262144$	241.21	1.09	119.21	2.20
$1024 \times 1024 = 1048576$	3671.80	0.29	149.06	7.03
$2048 \times 2048 = 4194304$	59424.76	0.07	562.95	7.45

Table 7.4: Speed results for a CPU-based QUICKSORT and the GPU-based bitonic sorting network. Columns with “ms” show the speed, i.e. the time to sort a sequence in milliseconds, while columns with “mval/sec” show the throughput, i.e. millions of values sorted per second.

7.4 Fast Fourier Transform

In mathematical physics, the field of *Fourier analysis* is about the decomposition of a signal in the "time domain" (or "spatial domain") into the coefficients of the *sinusoidal basis functions* in the "frequency domain" [109]. For instance, the *discrete Fourier transform* (DFT) is specialized to analyze the frequencies contained in "discrete and periodic" signals, which is important in signal processing [102]. The DFT of a sequence of N complex values $x_0, \ldots, x_{N-1}$ into their frequency component $X_0, \ldots, X_{N-1}$, is given by:

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} \tag{7.2}$$

where $k = 0, \ldots, N-1$.[1] The *inverse DFT* (IDFT) is defined accordingly:

$$x_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} X_n e^{+2\pi ikn/N} \tag{7.3}$$

The DFT can easily be computed by solving linear equations or using the "correlation method", but both approaches lead to $O(N^2)$ operations [17]. In contrast, the family of *fast Fourier transforms* (FFT) computes a DFT in $O(N \log N)$ operations by factorizing N. Commonly, the "radix-2" approach, presented by Cooley and Tukey, is used: The transform is repeatedly divided into two pieces of size $N/2$, until the sub-DFT transform a single value [25]. Because the factorization can take place in the time domain or in the frequency domain, the approach is called "decimation-in-time" or "decimation-in-frequency".

The *radix-2 decimation-in-frequency FFT* separates the sequence into the "top half" $(x_0, \ldots, x_{N/2-1})$ and the "bottom half" $(x_{N/2}, \ldots, x_{N-1})$. Equation 7.2 is rearranged as:

$$\begin{aligned}
X_k &= \sum_{n=0}^{N/2-1} x_n e^{-2\pi ikn/N} + \sum_{n=N/2}^{N-1} x_n e^{-2\pi ikn/N} && (7.4)\\
&= \sum_{n=0}^{N/2-1} (x_n + x_{n+N/2} e^{-\pi ik}) e^{-2\pi ikn/N} && (7.5)\\
&= \sum_{n=0}^{N/2-1} (x_n + (-1)^k x_{n+N/2}) e^{-2\pi ikn/N} && (7.6)
\end{aligned}$$

If equation 7.6 is separately considered for even and odd values of k, it can be seen that a DFT for a sequence of N numbers can be expressed by two DFTs for $N/2$ values:

$$X_k = \begin{cases} \sum_{n=0}^{N/2-1} (x_n + x_{n+N/2}) e^{-2\pi ik'n/(N/2)} & \text{Even } k \text{, with } k' = \frac{k}{2} \\ \sum_{n=0}^{N/2-1} ((x_n - x_{n+N/2}) e^{-2\pi in/N}) e^{-2\pi ik'n/(N/2)} & \text{Odd } k \text{, with } k' = \frac{k-1}{2} \end{cases} \tag{7.7}$$

[1]The normalization factor $1/\sqrt{N}$ is merely a convention and may differ to other definitions. Its only requirement is that the product of both factors is $1/N$. Please note that the factor is discarded in further equations, because it is applied in a post-process and does not belong to the actual DFT/IDFT.

By defining the sequences x_n^e and x_n^o (the so-called *FFT butterfly*, see figure 7.7) as

$$x_n^e = x_n + x_{n+N/2} \tag{7.8}$$
$$x_n^o = (x_n - x_{n+N/2})e^{-2\pi i n/N} \tag{7.9}$$

equation 7.7 can be simplified to

$$X_k = \begin{cases} \sum\limits_{n=0}^{N/2-1} x_n^e e^{-2\pi i k' n/(N/2)} = X_{k'}^e & \text{Even } k\text{, with } k' = \dfrac{k}{2} \\ \sum\limits_{n=0}^{N/2-1} x_n^o e^{-2\pi i k' n/(N/2)} = X_{k'}^o & \text{Odd } k\text{, with } k' = \dfrac{k-1}{2} \end{cases} \tag{7.10}$$

where the sequence $X_0^e, \ldots, X_{N/2}^e$ denotes the DFT of the "sequence of even numbers" $x_0^e, \ldots, x_{N/2}^e$, and $X_{k'}^o$ denotes the DFT of the "sequence of odd numbers" $x_{k'}^o$ respectively. Note that the same recursive approach can be used to compute the inverse FFT by changing the *twiddle factors* (the factors of x_n^o in equation 7.9) from $e^{-2\pi i n/N}$ to $e^{+2\pi i n/N}$.

If N is a regular power-of-two, the method can recursively be applied down to a set of 1-point DFTs (i.e. DFT of a single value), where $X_0 = x_0$. Because of the $\log N$ recursion levels, the complexity of this approach is $O(N \log N)$. While the presented approach can be implemented directly on the CPU, such a naive implementation "wastes" a lot of time and memory to reorder the intermediate results, i.e. to interleave x^e and x^o.

However, the reordering of each recursive level can be combined to a single processing step – which is called *tangling* – that can be done in post- or preprocessing: The intermediate array I of N values contains x^e in the "top" half $(0, \ldots, N/2-1)$ and x^o in the "bottom" half $(N/2, \ldots, N-1)$. The aforementioned reordering interleaves the sub-sequences x^e and x^o, so that the source index i_s turns into $i_d = 2i_s$ (for x^e) or $i_d = 2i_s + 1 - N$ (for x^o). By treating index i_s as a binary value, the least significant bit of i_d is the most significant bit of i_s. As a consequence, reordering at each recursion level leads to a *bit-reversal* of the global index (figure 7.8 illustrates this for an 8-point DFT). In other words, an explicit "bit-reversal" of the FFT result avoids time- and memory-consuming per-level reordering.

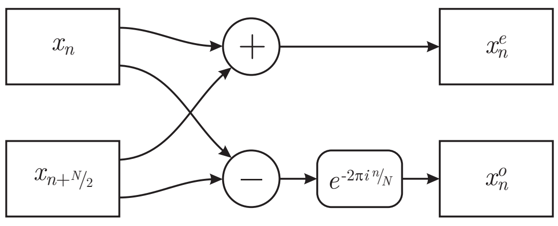


Figure 7.7: The *butterfly operation* is the core command of the FFT. It combines equations 7.8 and 7.9 to break a DFT of size N into two DFTs of size $N/2$.

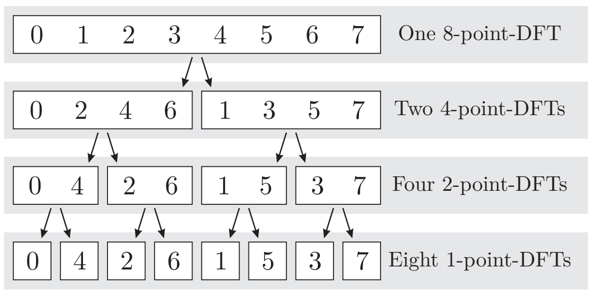


Figure 7.8: The global array index k is tangled during the $\log 8 = 3$ recursion levels of the 8-point FFT. The result at the bottom is a *bit-reversed* version of k.

7.4.1 Adaptation to the GPU – The Split-Stream-FFT

The CPU-based implementation of the presented radix-2 DIF-FFT algorithm is straightforward – some of the fastest CPU-based FFT libraries use this approach [38, 55]. However, there are some odds and ends, that prevent the algorithm to be implemented efficiently on a streaming architecture like the GPU, mainly due to its memory access pattern:

Aggregated Butterfly Operation – Currently, the FFT butterfly operation computes two values at once, which is not possible on the GPU. To be more precise, it is possible for the fragment unit of today's GPUs to compute and return multiple values, but these are stored at the same position in different output arrays. In contrast, the butterfly operation requires the result to be stored at different positions in the same output array. As a consequence, the FFT butterfly operation needs to be split, which leads to separate computations of the x^e and x^o expressions.

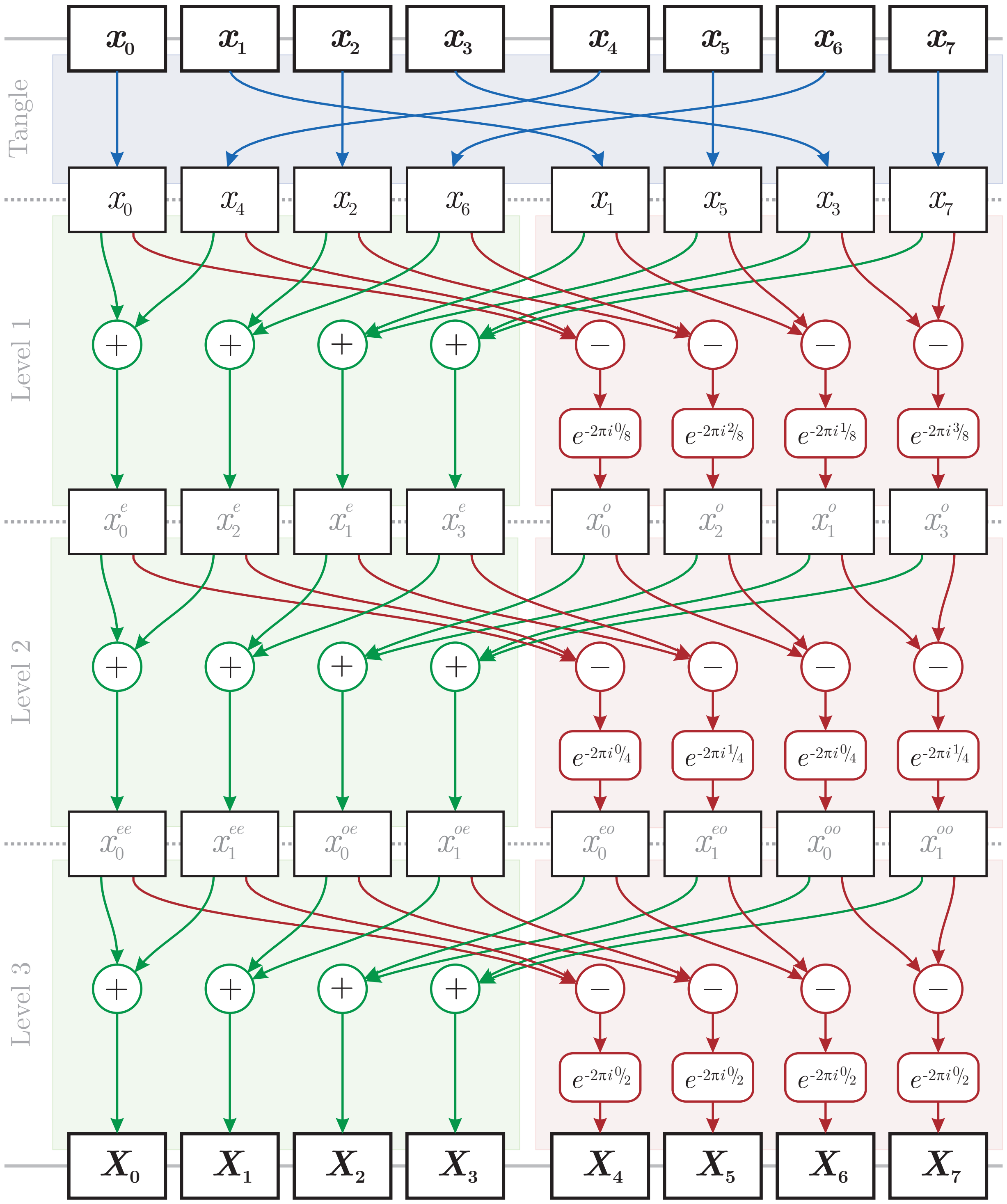
Non-Sequential Scattering – The previously described FFT approach stores the result of intermediate computations in an interleaved fashion: First, all array elements with even indices are computed, then, all array elements with odd indices. Unfortunately, it is not possible to use this memory layout on the GPU – see section 3.6. As a consequence, the ordering of the standard DIF-FFT needs to be redefined, which easily can be achieved by moving the "tangling" stage from back to front. This leads to the computation of a continuous output stream.

Explicit Tangling – Tangling has been presented as a separate preprocessing step to reorder the input stream for further processing. However, tangling can be combined with the first level of the FFT. To avoid an increase of code complexity, "scrambled regions" are used for *implicit tangling*, i.e. the first recursive level of an N-point FFT is done by performing N small butterfly operations separately, each combining x^e and x^o, where the "bit-reversal" is emulated by passing accordingly aligned input regions.

Especially the *split* of the butterfly operation and the sequential scattering of the output *stream* enable an efficient implementation of the FFT on the GPU – the so-called "Split-Stream-FFT" [56]. Figure 7.9 illustrates the data flow through a Split-Stream-FFT of size $N = 8$. The approach is very efficient, because the output stream of recursive level r can be used directly as the input stream for level $r + 1$.

7.4.2 Implementation Details and Results

The DFT is defined in complex space, hence, all computations of the Split-Stream-FFT are performed using complex values. Because the GPU^{++} system does not natively support this type, complex values are simulated using the two-component vector `vec2`, where `x()` is the *real* part and `y()` is the *imaginary* part. While this allows the direct reuse of component-wise addition and subtraction for complex values, a "complex multiplication" needs to be implemented separately. Alternatively, the STL can be used via `std::complex<vec1>` [112].



$\boldsymbol{x}_0$ $\boldsymbol{x}_1$ $\boldsymbol{x}_2$ $\boldsymbol{x}_3$ $\boldsymbol{x}_4$ $\boldsymbol{x}_5$ $\boldsymbol{x}_6$ $\boldsymbol{x}_7$
Tangle
x_0 x_4 x_2 x_6 x_1 x_5 x_3 x_7
Level 1
$e^{-2\pi i\, 0/8}$ $e^{-2\pi i\, 2/8}$ $e^{-2\pi i\, 1/8}$ $e^{-2\pi i\, 3/8}$
x_0^e x_2^e x_1^e x_3^e x_0^o x_2^o x_1^o x_3^o
Level 2
$e^{-2\pi i\, 0/4}$ $e^{-2\pi i\, 1/4}$ $e^{-2\pi i\, 0/4}$ $e^{-2\pi i\, 1/4}$
x_0^{ee} x_1^{ee} x_0^{oe} x_1^{oe} x_0^{eo} x_1^{eo} x_0^{oo} x_1^{oo}
Level 3
$e^{-2\pi i\, 0/2}$ $e^{-2\pi i\, 0/2}$ $e^{-2\pi i\, 0/2}$ $e^{-2\pi i\, 0/2}$
$\boldsymbol{X}_0$ $\boldsymbol{X}_1$ $\boldsymbol{X}_2$ $\boldsymbol{X}_3$ $\boldsymbol{X}_4$ $\boldsymbol{X}_5$ $\boldsymbol{X}_6$ $\boldsymbol{X}_7$

Figure 7.9: A graphical representation of the Split-Stream-FFT for a sequence of eight values. The approach takes $\log 8 = 3$ levels, with the same two passes in each level – one for computing the first half (x^e) and one for computing the second half (x^o). Note that the *tangling* stage at the beginning is done outside the kernel program via scrambled regions.

The GPU-based implementation of the Split-Stream-FFT approach has been compared to the fastest third-party libraries available (both using the CPU): The public FFTW software package and the commercial INTEL MATH KERNEL LIBRARY [38, 55]. The performance has been measured for a two-dimensional forward DFT of complex single-precision values. Both CPU-based competitors are highly-optimized, i.e. they make use of advanced techniques like self-modifying code, multi-threading, multi-processing, and the SSE2/SSE3 low-level instruction set. In contrast, the GPU^{++}-based graphics hardware implementation is just slightly optimized and contains large potential for further enhancements [44].

The benchmark results for various quadratic input sizes are shown in table 7.5. All timings are given in milliseconds. Furthermore, an approximated value for the "million floating-point instructions per second" (mflops) is given, using the following relation [38]:

$$\text{mflops} = \frac{N^2 \log_2 N}{100(\text{msec})} \tag{7.11}$$

Beginning with a size of 512^2 the GPU-based implementation performs much better than both CPU-based solutions and the 2048^2 array is transformed more than twice as fast on the GPU. Please note that the table does not contain "setup times", i.e. the time to create and initialize internal acceleration structures for a specific FFT size. For instance, the FFTW takes nearly three minutes to create its internal data for 1024^2 FFTs. While the IMKL does not require such setup, the Split-Stream-FFT's setup is always less than 50ms.

Comparing the source code complexity is a hard task in this case: Both CPU-based solutions provide much more functionality than the Split-Stream-FFT, e.g. multi-dimensional DFTs, non-complex values, and arbitrary input sizes. This leads to a significantly larger source code, e.g. the FFTW contains more than 120 thousand lSLOC. However, the FFT routines of the FXT library (143 lSLOC) are comparable in functionality with the presented GPU^{++}-based solution (89 lSLOC), even if it is much slower than the FFTW or IMKL [5]. It should be noted that other GPU-based approaches are much more complex regarding their source code, e.g. the FFT by Sumanaweera (1762 lSLOC) [114].

Size	**FFTW** (CPU)		**IMKL** (CPU)		**Split-Stream** (GPU)		
	ms	mflops	ms	mflops	ms	mflops	Error
256 × 256	2.31	2270	1.84	2849	9.45	555	1.6×10^{-10}
512 × 512	12.99	1816	15.45	1527	10.96	2153	7.8×10^{-10}
1024 × 1024	52.33	2004	56.32	1862	25.17	4166	4.0×10^{-9}
2048 × 2048	233.02	1980	283.97	1625	109.44	4216	2.4×10^{-8}

Table 7.5: Benchmark results for a two-dimensional DFT of single-precision complex values. The GPU-based Split-Stream-FFT is compared to the fastest CPU-based approaches available – FFTW and INTEL MATH KERNEL LIBRARY [38, 55]. The last column contains the root-mean-square-error of the GPU-based results compared to the CPU-based FFTW.

7.5 Tomographic Reconstruction

Most medical image processing algorithms (like the ones in section 7.1) expect the input data in a specific format: A three-dimensional volume (or as a two-dimensional slice through such a volume) that contains a scaled representation of the scanned object. For instance, it is assumed implicitly that a scan through a human head will show the teeth, skull, and brain in the same geometrical relationship as it is in reality. However, this is not the representation returned by most tomographic scanners, like a CT device. In fact, a tomographic device returns a set of *projections* from different scanning angles. The projections need to be *transformed* into the expected format for further processing, i.e. the object is "reconstructed" from its projections. This is called *tomographic reconstruction*.

A variety of practical algorithms have been developed to reconstruct tomographic data, such as statistical and algebraic reconstruction [42, 4]. However, by far the most popular approach is the "filtered back-projection" (FBP) and its variations [30]. While the algorithm works best for parallel projections (i.e. projections that have been generated using parallel CT rays), there are adaptations of the FBP for fan- and cone-beam scanning devices [34].

7.5.1 The Fourier Slice Theorem

A typical example for tomographic scanning is computer tomography (CT), where x-rays attenuate as they propagate through biological tissue. The total attenuation of such a beam, as it travels a straight line through the object, is represented by a *line integral* [58]: If the object is represented by a two-dimensional function $f(x, y)$, the line integral $P_\phi(t)$, where t is the distance to center of the object and ϕ is the slope of the line, is defined as:

$$P_\phi(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\delta(x\cos\phi + y\sin\phi - t)\,dx\,dy \tag{7.12}$$

Here, δ is the "delta function", i.e. $\delta(0) = 1$ and $\delta(n \neq 0) = 0$. A collection of such line integrals is called a *projection*, and the *parallel projection* is its simplest form: a set of parallel line integrals given by $P_\phi(t)$ for a constant ϕ (see the left of figure 7.10).

The *Fourier slice theorem* states that the Fourier transform of a parallel projection of an image $f(x, y)$, taken at angle ϕ, is equal to a line through the image in its two-dimensional frequency domain, $F(u, v)$, subtending an angle ϕ with the u-axis [57]. The theorem can be shown exemplarily for an angle of $\phi = 0$, which leads to a parallel projection along the v-axis. The two-dimensional Fourier transform of $f(x, y)$ is given by:

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-2\pi i(ux+vy)}\,dx\,dy \tag{7.13}$$

Consider the slice through $F(u, v)$ at $v = 0$:

$$\begin{aligned} F(u,0) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-2\pi iux}\,dx\,dy && (7.14)\\ &= \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty} f(x,y)\,dy\right]e^{-2\pi iux}\,dx && (7.15)\end{aligned}$$

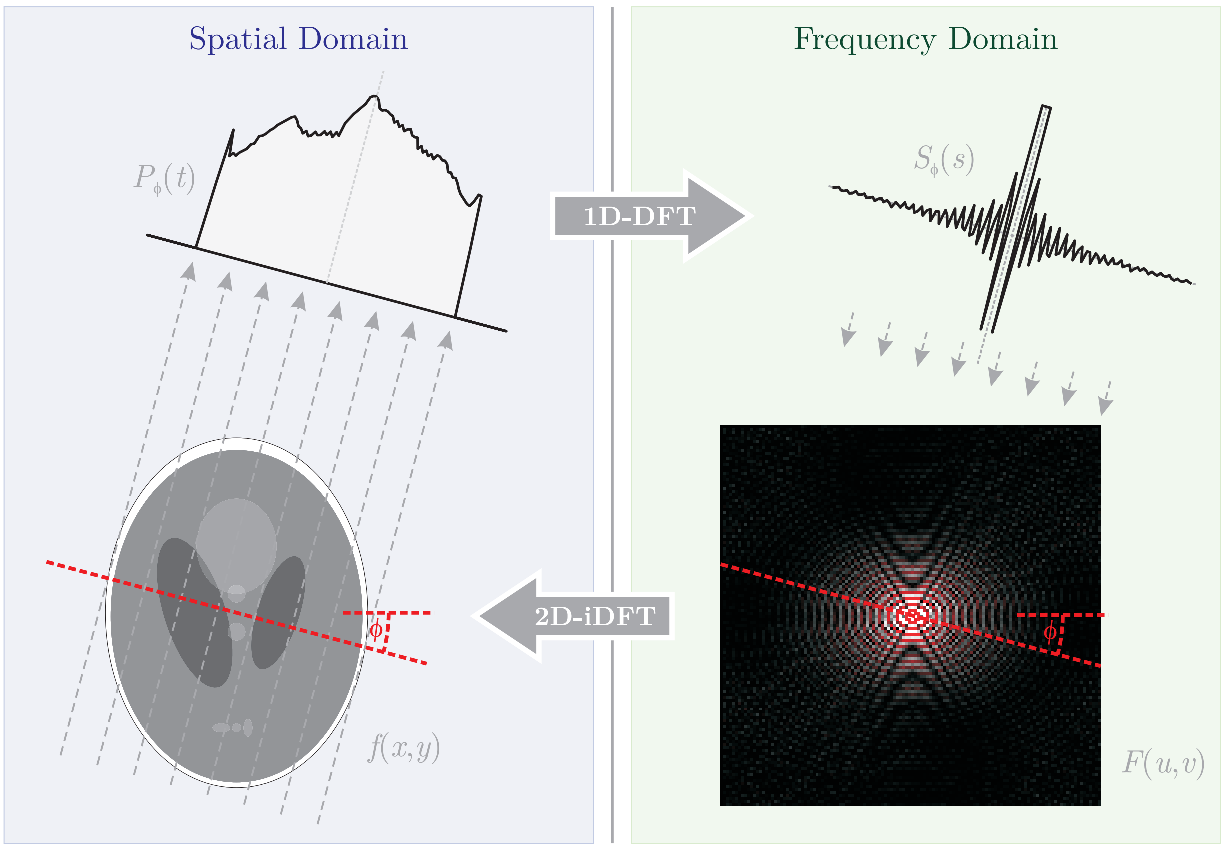


Figure 7.10: The "Fourier slice theorem" relates the Fourier transform $S_\phi(s)$ of a projection $P_\phi(t)$, which is created using parallel rays perpendicular to angle ϕ through the object $f(x, y)$, to the Fourier transform $F(u, v)$ of the same object along a single radial, also at angle ϕ.

Substituting the term in brackets with equation 7.12 of a parallel projection leads to:

$$F(u, 0) = \int_{-\infty}^{\infty} P_{\phi=0}(t = x)e^{-2\pi iux}\,dx \tag{7.16}$$

This shows that a line at angle $\phi = 0$ through the two-dimensional Fourier transform of the image $f(x, y)$ is equal to the one-dimensional Fourier transform of the parallel projection at the same angle $\phi = 0$. Obviously, this result is independent of the actual orientation between the object and the coordinate system, because the coordinate system can be rotated by an arbitrary angle ϕ. Figure 7.10 illustrates the Fourier slice theorem.

The Fourier slice theorem can be used for tomographic reconstruction: By giving enough projections in frequency space at different angles, they could be assembled into a complete estimate of the two-dimensional Fourier transform of the scanned object, which then could be inverted to get the representation of the object in the spatial domain. While this provides a nice conceptual model, it is not feasible for practical implementations, due to its speed and reconstruction quality. Instead, the algorithm that is currently being used in almost all tomography applications is the *filtered back-projection* (FBP).

7.5.2 Filtered Back-Projection

Because the algorithm can be explained in a very intuitive and illustrative way, the mathematical foundation of the filtered back-projection is left for further reading [30, 58].

To develop the idea of the FBP, a single parallel projection is considered, taken at angle $\phi = 0$. Because of the Fourier slice theorem, the Fourier transform of this projection is equal to the u-axis of the two-dimensional frequency representation of the object at $v = 0$. By assuming all other projections to be zero and by performing a two-dimensional inverse DFT, a simple (but distorted) reconstruction of the object is created. This reconstruction can be seen as a "filtered version" of the original object, i.e. frequencies in the object's frequency domain are masked out. To improve the reconstruction, a second parallel projection at angle $\phi = \pi/2$ (i.e. perpendicular to the first) is considered and its DFT is *added* to the aforementioned two-dimensional frequency domain of the object. The important idea is: Because of the linearity of the Fourier transform, this summation can be done either in the frequency domain or in the spatial domain: Instead of combining all transformed projections to build the two-dimensional Fourier transform of the object followed by an inverse Fourier transform, the two-dimensional frequency domains of each projection can be created independently – and all their inverse transforms are summed together. This sounds more time-consuming as it is, because the inverse Fourier transform of the "sparse" two-dimensional frequency domain (containing the Fourier transform of a single projection) is the projection itself, rotated by angle ϕ and duplicated over the complete image space.

However, rotating the projections by ϕ, duplicating them along a line perpendicular to ϕ and summing them together obviously cannot result in an adequate reconstruction, because adding two Fourier transformed projections together in the two-dimensional frequency space doubles the DC frequency (at $u = v = 0$). In fact, for N projections the frequency is N times higher as expected. Thus, the frequencies of the Fourier transformed projections have to be *weighted* before back-projecting (where the weighting is proportional to the distance to $u = v = 0$). Weighting in the frequency domain is the same as filtering in the spatial domain, hence the name of the "*filtered* back-projection". The algorithm is illustrated in figure 7.11.

7.5.3 Implementation Details and Results

The filtered back-projection consists of two steps: The *filtering* is based on a DFT to weight the frequency components, and the *back-projection* sums "smeared" and rotated images. While the first step can be implemented using the GPU-based FFT of section 7.4, the second step is implemented by exploiting the texturing stage of the graphics hardware.

As in section 7.2, the back-projection stage can be optimized by smearing multiple filtered projections over the reconstruction image, i.e. n passes can be combined by smearing n rows at the same time, each with a different input region. This allows the graphics accelerator to make use of all texture stages (at least 16 on today's graphics hardware).

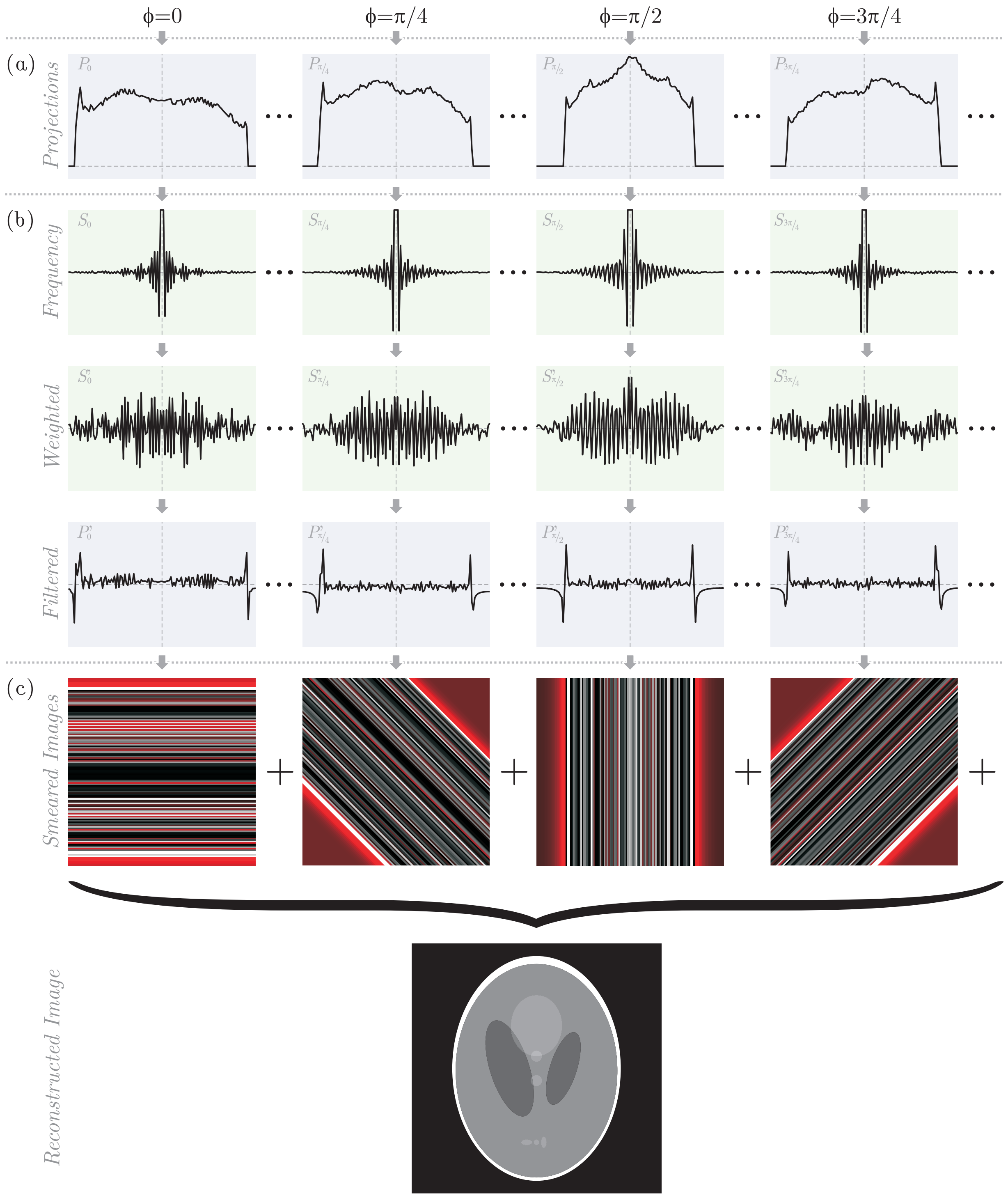


Figure 7.11: Tomographic reconstruction of the Shepp-Logan phantom using filtered back-projection [58]: (a) projections measured by a tomographic device are (b) Fourier transformed, weighted, and back-transformed to the spatial domain. The filtered projections are then (c) rotated and smeared over the image plane. The total sum leads to the reconstruction.

Size	**CPU**		**GPU**		
	Discrete	Linear	×1	×2	×4
128 × 128	36.86 ms	54.73 ms	38.20 ms	23.37 ms	15.99 ms
256 × 256	303.43 ms	434.52 ms	68.69 ms	40.16 ms	24.67 ms
512 × 512	2419.1 ms	3465.3 ms	130.55 ms	70.78 ms	41.78 ms
1024 × 1024	19591 ms	27900 ms	545.44 ms	281.96 ms	151.40 ms
128 × 128 × 128	4.75 sec	7.09 sec	4.90 sec	3.00 sec	2.06 sec
256 × 256 × 256	78.11 sec	114.1 sec	17.73 sec	10.31 sec	6.29 sec
512 × 512 × 512	1245 sec	1781 sec	66.88 sec	36.41 sec	21.41 sec
1024 × 1024 × 1024	20065 sec	28556 sec	560.5 sec	378.7 sec	155.9 sec

Table 7.6: Benchmark results that have been measured for the filtered back-projection of two-dimensional (in milliseconds) and three-dimensional (in seconds) input data sets. While the CPU-based timings are separated in *discrete* (i.e. nearest-neighbor) and *linear* interpolation, all timings on the GPU were measured using linear interpolation for the best quality.

Table 7.6 contains the benchmark results for tomographic reconstruction via filtered back-projection of two- and three-dimensional data sets with various sizes. While no performance change is noticeable for different interpolation schemes (i.e. nearest-neighbor and linear interpolation) on the graphics hardware, it makes a huge difference on the main processor: The speed-performance decreases by more than 40% for linear interpolation. However, nearest-neighbor interpolation is inadequate in most cases. The GPU-based implementation outperforms the CPU-based solution for all relevant sizes: 1024 two-dimensional projections of size 1024 × 1024 are reconstructed to a volume in 2:35 minutes on the GPU, while the main processor requires more than eight hours. However, today's practical reconstruction size is 512^3, where the tomographic reconstruction is computed in 21 seconds instead of half an hour, i.e. the graphics accelerator is 83 times faster than the CPU.

The FFT's source code complexity has been discussed extensively in section 7.4.2, therefore it is *excluded* from the complexity comparison of the filtered back-projection. This leads to 46 lSLOC for the GPU-based implementation, in contrast to the 81 lSLOC of the CPU-based solution. Furthermore, the Gpu^{++}-based approach is significantly less complex than other GPU-based implementations of the FBP, like the one by Sumanaweera (253 lSLOC) [114].

Chapter 8

Discussion

The utility of the GPU^{++} development system has been evaluated in chapter 7 by implementing various general-purpose algorithms on the main and the graphics processor. Two factors have been of importance: the *speed-increase* of the GPU-based implementation compared to the CPU-based solution, and the *source code complexity* of both versions. Figure 8.1 shows that all implementations on the GPU are significantly faster than their CPU equivalents. Furthermore, the figure shows that the source is of comparable complexity for the different processor types. While the first finding ("performance") has been presented by others, the second achievement ("complexity") is a novel result of this work.

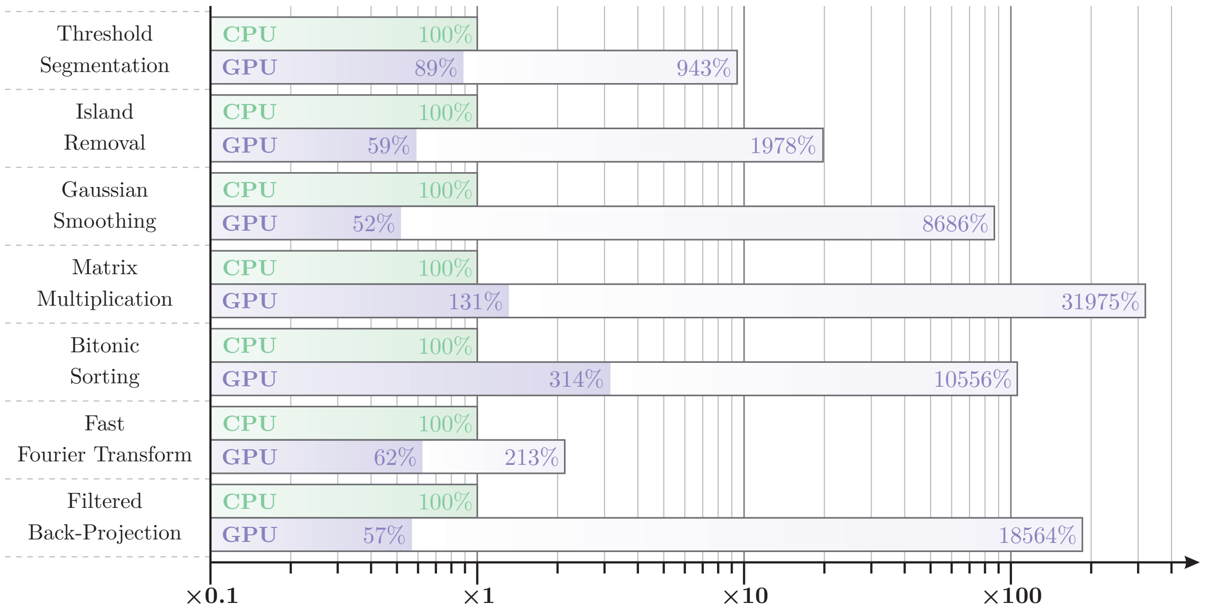


Figure 8.1: Speed and complexity comparison of the CPU- and GPU-based implementations for an input size of 2048^2 (except FBP, where the size is 1024^2). The logarithmic measurements are normalized to the main processor. The light blue bars illustrate the relative speed, while the dark blue bars stand for the relative software complexity of the graphics hardware.

8.1 Data Transfer

The benchmark results that have been presented in chapter 7 do not include any data transfer between the main and the graphics processor. This is comprehensible, because the figure shows the relationship of the *processing performance* instead of the data throughput from the CPU's point-of-view. As a matter of fact, it is ambiguous if the data transfer is added to the CPU-based implementation or if it should be associated with the GPU-based solution – it depends on the context. While in the beginning of GPGPU the transfer time was included to the graphics hardware measurements, this has changed in the last years. More often, the GPU is responsible for the complete processing pipeline and the main processor is only used for pre- and post-processing the data sets.

However, transfer timings cannot be ignored when a decision ("CPU vs. GPU") needs to be made. How often is the data transferred? How much data is up- or downloaded? Can more steps be done on the GPU – even if an individual step is slower – to reduce data transfer? But these questions need to be answered in context of the actual project.

Since 2005, most graphics accelerators make use of the "PCI-Express" transfer bus, which allows a theoretical maximum transfer rate of 8 GiBytes/sec in both directions (which makes a combined transfer rate of 16 GiBytes/sec) [7]. However, the practical transfer rate is far below this theoretical maximum, e.g. the benchmarking system of chapter 7 achieves an average transfer rate of 900 MiBytes/sec for up- and 650 MiBytes/sec for downloading.

Nevertheless, the more complex the processing task, the more negligible the data transfer. Table 8.1 shows the speed-performances of the algorithms of chapter 7, but this time the transfer has been included in the GPU measures. While "threshold segmentation" is the only GPU-based approach that no longer is competitive with the CPU-based implementation, the full medical pipeline still beats the CPU by a factor of 13.

Algorithm	**In-/Out Types**	**CPU** 1024^2	**CPU** 1024^3	**GPU** 1024^2	**GPU** 1024^3
Threshold Segmentation	$1 \mapsto 1$ `short`	3.20 ms	3.28 sec	5.90 ms	6.04 sec
Island Removal	$1 \mapsto 1$ `short`	12.48 ms	37.06 sec	6.31 ms	7.40 sec
Gaussian Smoothing	$1 \mapsto 1$ `short`	55.23 ms	84.83 sec	6.27 ms	6.92 sec
Full Medical Pipeline	$1 \mapsto 1$ `short`	70.91 ms	125.2 sec	7.88 ms	9.50 sec
Matrix Multiplication	$2 \mapsto 1$ `float`	35313 ms	-	135.2 ms	-
Bitonic Sorting Network	$2 \mapsto 2$ `short`	3672 ms	-	159.7 ms	-
Fast Fourier Transform	$2 \mapsto 2$ `float`	52.33 ms	-	46.37 ms	-
Filtered Back-Projection	$1 \mapsto 1$ `short`	27900 ms	28556 sec	156.7 ms	161.3 sec

Table 8.1: Benchmarks that include the data transfer between CPU and GPU. While simple GPU implementations (i.e. threshold) may no longer compete with the main processor, most other GPU-based approaches still outperform the CPU by multiple orders of magnitude.

8.2 Software Complexity

Measuring the complexity of software is a nontrivial task, mainly because the definition of complexity varies between projects, programming languages, and developers. In fact, the same source code is judged differently by individual developers and a simple implementation in one language may be hard to understand in another. Furthermore, even relative statements are highly dependent on the software engineer's experience. As a consequence, developers with in-depth knowledge of computer graphics programming and HLSLs may find shader-based GPGPU development very basic and are not willing to learn the concepts and terminology of the GPU^{++} development system. But other software engineers find HLSL-driven development too complex, compared to the GPU^{++}-based development.

There are many (sometimes even concurrent) complexity metrics that use the aforementioned factors to compute the complexity and costs of software components [78, 115, 21]. Such metrics result in a clear quantitative statement about the benefit of GPU^{++} in contrast to traditional GPGPU development in a specific software project. However, the mentioned criteria are highly speculative and as long as no further details about the software project are known, an average software developer – without further computer graphics skills – is considered and the overall project also is assumed to be of average complexity. As a consequence, the limited information only allows a qualitative tendency statement about GPU^{++}-based development.

According to the widely-accepted *constructive cost model* (COCOMO), the amount of "delivered source instructions" (i.e. those lines-of-code that make it into the final software) exponentially influences the development time: The development time for a medium size software project is computed by $T = 3{*}\mathrm{lSLOC}/1000^{1.12}$, which leads to an average efficiency of 7–8 lSLOC per developer and day for a project of 100.000 lSLOC [13, 14].[1] Because of this simple relationship between code lines and development efficiency, the lSLOC metric has been used for all algorithms presented in chapter 7.

For instance, basic COCOMO predicts a development time of approx. 5 days to implement the FFT using the GPU^{++} development system, while it would take nearly 9 days for the CPU-based approach and more than 5.6 months for a GPU-based solution using a string-based shader language. Again, this only gives a tendency and depends on the specific parameter of the actual software project and the software engineer's experience.

[1]There are various versions of the "constructive cost model": *Basic* and *intermediate* COCOMO (sometimes also referred as "COCOMO 81") have been designed primarily for software development of large projects on mainframe computers [13], while "COCOMO II" (also called "COCOMO 2.0") has been extended to desktop development, code reusability, and the use of off-the-shelf software components [14]. However, all versions lead to similar lSLOC throughput when applied to "average software projects".

Chapter 9

Conclusion

The acceleration of general-purpose computations by using programmable graphics hardware has become very popular over the last decade, mainly due to the massively parallelized processor design and the enormous availability of GPUs as an inexpensive standard component of today's personal computers. However, most existing GPU-related development systems are graphics-oriented and developing general-purpose applications in such environments is a challenging task for programmers who are not familiar to computer graphics.

This thesis has taken a novel direction in the design and implementation of the GPU^{++} development system, which sees GPU programming no longer as a separate task, which is done by experts of the graphics domain. Instead, it is integrated in the existing application development workflow. This integration has been realized in two ways: First, the graphics-oriented programming paradigms have been replaced by generic concepts, using novel techniques like the *unified kernel definition* and the *vector fusion* approach. Second, the definition of GPU-based code has been embedded into the C^{++} programming language using *ad-hoc polymorphism* and *operator overloading*. In addition, the development complexity is further reduced by providing automatic optimizations of the GPU-based code.

The GPU^{++} development system has been extensively evaluated in chapter 7 by implementing well-known general-purpose algorithms on the CPU and the GPU. Six out of the seven exemplar algorithms run at least one order of magnitude faster on the graphics processor, and four algorithms are even *two orders of magnitude faster than the CPU*. While such a performance increase has been shown for other GPU-based programming systems, it usually comes for the price of significantly higher software complexity. This is not the case for the GPU^{++} system: The GPU implementations of two of the seven exemplar algorithms have a slightly higher complexity compared to the main processor solutions, while the other five implementations have even *less complexity than their CPU-based counterparts*.

These results impressively illustrate that the novel design of GPU^{++} enables software developers of general-purpose algorithms to benefit from the computational power of today's graphics hardware, without an increase of source code complexity and development time. This makes the GPU^{++} system a valuable tool that may significantly help to increase the acceptance of general-purpose GPU-based programming in the industrial domain.

9.1 Contributions

This thesis has made several contributions to computer graphics, graphics hardware, software engineering, and general-purpose computations on the graphics processing unit:

Generic GPU-Programming Concepts – The GPU^{++} development system that has been presented in this thesis uses a compact set of *generic concepts* and a *corresponding terminology* for GPU-based programming. Such an abstraction from the prior graphics-oriented programming paradigms enables the software developer to implement general-purpose algorithms on the GPU *without being an expert in computer graphics or graphics hardware*. Thus, the complexity of GPU-based implementations is significantly reduced, which leads to a noticeably shorter development time.

Embedded Kernel Definition – One of the most important features of GPU^{++} is the seamless integration of its kernel program definition into the C^{++} programming language. This is realized by using *ad-hoc polymorphism* via *operator overloading*. The GPU^{++} system enables the integration of GPU-based programming into the existing software development workflow, which includes the seamless interchange and communication between code for the different platforms, as well as the detection of most *compile-time errors* of the kernel program at compile-time of the host application.

Graphics Hardware Abstractions – A major contribution of the GPU^{++} development system is the efficient abstraction of graphics hardware features, such as the *vector processor* architecture and the *different computation units* (vertex and fragment processing). This includes a novel technique that *automatically combines individual scalar instructions* to generate a single equivalent vector instruction. Furthermore, kernel programs are *automatically partitioned* to efficiently exploit the different GPU computation units. In other GPU-based programming approaches, such advanced optimizations have to be done manually by the developer. Hence, both techniques lead to a significant *reduction of the GPU's source code complexity*.

Optimizations – The GPU^{++} development system applies *optimizations* to the GPU-based code automatically, which leads to a significantly increased run-time performance. The optimizations include dead-code elimination, algebraic simplifications, and the elimination of common sub-expressions. Nevertheless, the compile-time performance of kernel program optimizations is noticeably reduced by using the novel *visitor traits* technique and an efficient custom RTTI scheme.

Performance And Complexity Analysis – A *variety of algorithms* have been implemented on the main and graphics processor to *evaluate the GPU-based development system*, including computations from digital signal processing and the medical domain. The evaluation impressively demonstrates that the GPU^{++} development system enables software developers to implement a large range of algorithms (from matrix-multiplication to filtered back-projection) on the GPU that *run significantly faster* than their CPU-based counterparts, but have *similar source code complexity*.

9.2 Future Work

Graphics processing units evolve at enormous rates and new features are introduced in every new hardware generation. At the time of writing this thesis NVIDIA released the NV80 chip design, which is the first GPU that supports the specifications of MICROSOFT's DIRECTX10 application programming interface. DIRECTX10 introduces a new computational unit, the *geometry shader*, that is not yet supported by the GPU^{++} system. This is just one area of further investigation that is addressed in the following subsections.

Run-Time Flow-Control / Conditional Statements

The GPU^{++} development system does not support all flow-control approaches for the kernel program (i.e. the program flow depends on a GPU-based run-time variable) that have been discussed in section 3.6. While the program flow of GPU-based code cannot be explicitly affected by the computations of the code itself, there are implicit ways via the `min()`, `max()`, `select()`, and `threshold()` commands. However, a much more intuitive way would be the use of C^{++}'s built-in conditional commands, like `if`, `else`, and `while`. Therefore, native conditionals will be supported in future releases of the GPU^{++} development system.

Geometry Shader

The latest generation of GPUs introduces a new computation unit that sits in-between the existing vertex and fragment units. The *geometry shader* allows the creation of new, or the elimination of existing stream elements [90]. The new computation unit vastly enlarges the family of algorithms that can be accelerated by the GPU, because the size of the output stream is more flexible. While extending the GPU^{++} development system tends to be non-critical (because all computational units of the GPU are virtualized internally and abstracted through the interface), the static "`array`" class type may become obsolete.

Additional Back-Ends

The GPU^{++} development system is based on the full virtualization of the graphics processing unit combined with a flexible and efficient back-end infrastructure. It is planned to add additional back-ends to the software, including the latest version of the DIRECTX low-level shading language, NVIDIA's CUDA, ATI's CTM, and a CPU-based reference back-end for advanced debugging issues. In fact, while the design focus of the GPU^{++} system has mainly been the graphics processing unit, there is no architectural limitation to address other hardware, like FPGA boards, DSP chips, and the CELL processor broadband engine.

Hybrid and Distributed Computing

The GPU^{++} development system exploits the computational power of the graphics processing unit to perform algorithms faster than the CPU. However, while the GPU does all the work, the main processor is idle most of the time. Using both processors in parallel would be much more efficient, i.e. the data processing is split into two parts, where one is computed by the GPU and the other is computed by the CPU. While such a *hybrid computing* approach sounds trivial, the "split of the data processing" requires a general and adaptive cost/performance model of the different back-ends (CPU vs. GPU) and the data transfer bandwidth. As a start, it is planned to extend the GPU^{++} development system to support "parallel back-end execution", but leave the actual splitting of the data to the developer.

Furthermore, such systems allow *distributed computing*, where each node of a large computing cluster applies the kernel program to a small part of the data. Again, while the partitioning of the data has to be done by the software developer, the transfer and actual computation will be integrated in future versions of the GPU^{++} development system.

Appendix A

Reference of the Language Extension

This appendix will give a brief reference to the C^{++} language extension introduced by the GPU^{++} system. It includes an overview of the new vector and matrix class types, the mathematical operations that are specified for these types, type qualifiers to pass optimization-hints to the compiler, as well as, built-in functions and variables. Further information can be found in chapter 4, where the embedded approach of GPU^{++} is described.

In addition to this reference, appendix B presents basic programming examples together with a line-by-line description. In addition, more complex applications are introduced in chapter 7. However, it is highly recommended to read both appendices in advance.

A.1 Vectors

The GPU^{++} development system presents data types for vectors with up to four floating-point components – `vec1`, `vec2`, `vec3`, and `vec4`. Vectors can be used to store computer graphics related data (e.g. colors, normals, and positions) as well as general-purpose data (e.g. complex values, array positions, and polar coordinates). Please note that variables of type `float` can be used wherever type `vec1` is expected, but this does not work vice versa, i.e. once a `float` value is stored in a vector component, it cannot be casted back to `float`.

Constructors

Constructors are used to create vectors from a set of scalars or other vectors. A single scalar parameter (i.e. `vec1` or `float` type) causes the new vector to initialize all components with that scalar. Non-scalar parameters will be assigned in order, from left to right, to the new vector components – there have to be enough components provided in the constructor.

```
vec2 v2a(3.5);                                                  // v2a = (3.5, 3.5)
vec3 v3a(1.0, v2a);                                        // v3a = (1.0, 3.5, 3.5)
vec4 v4a(5.0, v2a);    // compilation-error: not enough scalars for all components!
```

Component Access

Individual vector components can be accessed using the *array subscript* operator `[]`, i.e. the first component of the three-component vector `v3fb` is accessed via `v3fb[0]`, while the third component is accessed by `v3fb[2]`. This returns a new `vec1` object that can be used for reading and writing (where writing will change the addressed vector component). Please note that it is illegal to access nonexisting components, however, it does not result in a compile-time error, instead, the run-time exception `exceptionRange` is thrown.

```
vec3 v3col(1.0, 0.6, 0.4);       // initialize three-component vector with "brown"

vec1 v1red   = v3col[0];                         // returns the first component = 1.0
vec1 v1alpha = v3col[3];               // run-time exception: there is no 4th component

v3col[1]     = 1.0;                                        // v3col = (1.0, 1.0, 0.4)
v3col[0]     = v3col[2];                                   // v3col = (0.4, 1.0, 0.4)
```

Swizzling

An additional approach to access vector components is *swizzling*, where each component is associated with a letter: `x()` returns the first, `y()` the second, `z()` the third, and `w()` the fourth component of a vector. Furthermore, the letters can be concatenated to access multiple vector components, e.g. `v3p.yx()` returns a two-component vector with *rearranged* components, and `v3p.zzzz()` returns a four-component vector with *replicated* components (while the first can be used for reading and writing component, the second cannot be used for writing components, due to ambiguities).

```
vec3 v3a(0.0, 1.0, 3.0);                                   // v3a = (0.0, 1.0, 3.0)

vec1 v1a = v3a.y();                                         // returns v1a = (1.0))
vec2 v2a = v1a.xy();          // compilation-error: there is no 'y' component in v1a

v3a.y() = 5.0;                                             // v3a = (0.0, 5.0, 3.0)
v3a.xz() += 0.5;                                           // v3a = (0.5, 5.0, 3.5)
v1a.xx() = vec2(6.6, 2.0);   // compilation-error: writing to replicated components
```

Accessing components via *swizzling* is type-safe, i.e. illegal statements (like accessing the eighth component of a two-component vector) are detected at CPU-based compile-time. Hence, it is recommended to use swizzling in favor to *array-driven component access*.

Please note that shading languages like HLSL or GLSLANG allow swizzling with additional letter-sets – `rgba` is used for colors and `stpq` for texture coordinates. However, such sets are relevant only to computer graphics and provide no additional functionality.

			vec1				vec2				vec3				vec4		
	+	−	+	−	*	/	+	−	*	/	+	−	*	/	+	−	*	/
vec1	●	●	●	●	●	●	🟢	🟢	●	●	🟢	🟢	●	●	🟢	🟢	●	●
vec2	●	●	🟢	🟢	●	🟢	●	●	🔴	🟢	○	○	○	○	○	○	○	○
vec3	●	●	🟢	🟢	●	🟢	○	○	○	○	●	●	🔴	🟢	○	○	○	○
vec4	●	●	🟢	🟢	●	🟢	○	○	○	○	○	○	○	○	●	●	🔴	🟢

Table A.1: All supported operators between vectors of different sizes. Mathematically incorrect operations are shown in red, while undefined operations are shown in green.

Basic Operators

Table A.1 shows all basic operators that are defined for the aforementioned vector types. Please note that the operators work component-wise – this might result in definitions that are mathematically incorrect (e.g. the multiplication of two vectors via "*" does *not* result into a vector dot product) or mathematically undefined (e.g. division of vectors using "/").

```
vec2 v2a(0.0, 1.0);                                         // v2a = (0.0, 1.0)
vec2 v2b(1.0, -1.0);                                       // v2b = (1.0, -1.5)

vec2 v2c = v2a + v2b;                                      // v2c = (1.0, -0.5)
vec2 v2d = v2a * v2c;                                      // v2d = (0.0, -0.5)
vec2 v2e = v2a / (-v2c.y());                                // v2e = (0.0, 2.0)
```

A.2 Matrices

The GPU^{++} development system supports 2×2, 3×3, and 4×4 matrices of floating-point values: `mat2`, `mat3`, and `mat4`. Note that matrices are accessed in column major order.

Constructors

Matrices are initialized via a set of scalars or vectors. However, it is not possible to construct a matrix from a set of smaller matrices. A single scalar will initialize the diagonal of a matrix, with all other elements set to zero. Furthermore, a set of vectors will initialize the matrix columns, while a set of scalars will initialize each matrix element.

```
mat3 m3a(1.0);         // m3a = ((1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0))
mat2 m2a(vec2(0.0, 1.0), vec2(1.0, 2.0));         // m2a = ((0.0, 1.0), (1.0, 2.0))
mat3 m3b(0.0, 1.0, 2.0,  3.0, 4.0, 5.0,  6.0, 7.0, 8.0);             // m3b = (...)
mat2 m2b(0.0, 1.0, vec2(2.0, 3.0)); // compilation-error: mixed scalars and vectors
```

			vec1				vec2				vec3				vec4				mat2				mat3				mat4			
	+	-	+	-	*	/	+	-	*	/	+	-	*	/	+	-	*	/	+	-	*	/	+	-	*	/	+	-	*	/
mat2	●	●	○	○	●	●	○	○	●	○	○	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○
mat3	●	●	○	○	●	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○
mat4	●	●	○	○	●	●	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	●	●	●	○

Table A.2: All supported operators between vectors and matrices of different sizes. In contrast to the overloaded vector operators, matrix operators are mathematically correct.

Column Vector Access

Specific columns are accessed via the `[]` operator. In combination with the *component access* of vectors, this leads to the well-known C^{++} syntax for matrix component access.

```
mat3 m3(4.0);            // m3 = ((4.0, 0.0, 0.0), (0.0, 4.0, 0.0), (0.0, 0.0, 4.0))

vec3 v3r = m3[1];                                                // v3r = (0.0, 4.0, 0.0)
m3[2] = vec3(1.0);       // m3 = ((4.0, 0.0, 0.0), (0.0, 4.0, 0.0), (1.0, 1.0, 1.0))

vec1 v1r = m3[1][1];                                                       // v1r = (4.0)
m3[0][2] = -1.0;        // m3 = ((4.0, 0.0, -1.0), (0.0, 4.0, 0.0), (1.0, 1.0, 1.0))
m3[0].y() = -1.0;      // m3 = ((4.0, -1.0, -1.0), (0.0, 4.0, 0.0), (1.0, 1.0, 1.0))
```

Operations

An overview of the basic operators defined for the matrix types is presented in table A.2. In contrast to vector operators, the matrix operators do *not* always work component-wise.

A.3 Type Qualifiers

The *type qualifier* mechanism is used to declare variables as constant for specific computation frequencies – this may lead to optimizations. The concept of computation frequencies is explained in section 3.4 and type qualifiers are described in further detail in section 4.2.3.

Uniforms

The `uniform` type qualifier marks a variable as "mutable between multiple executions", i.e. the variable stays constant within a single execution. In other words, uniforms can be changed without the need of a kernel program re-compilation. The `uniform` keyword can be used for all vector and matrix classes, i.e. `vec1`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, and `mat4`.

The following code fills two arrays (`array1` and `array2`) with the user-specified four-component vector `m_v4uniform` (includes the kernel and its binding in the main procedure):

```
int main(int, char * *)
{
   struct foobar : public kernel< opengl > {
      uniform< vec4 > m_v4uniform;    // specify a four-component vector as uniform
      virtual vec4 element(attribute< vec4 > const & v4Pos) const
      {
         return m_v4uniform;              // copy the uniform to the output element
      }
   }

   foobar fb;                                    // instantiate the kernel program

   array2< float, 3 > a2out1(512, 512);    // create a two-dimensional output array
   fb.m_v4uniform = 0.5;                    // set uniform to (0.5, 0.5, 0.5, 0.5)
   a2out1 << fb;                                // (compile and) execute on a2out1

   array2< float, 3 > a2out2(64, 64);      // create a two-dimensional output array
   fb.m_v4uniform = vec4(0.1, 0.2, 0.3, 0.4);        // set to (0.1, 0.2, 0.3, 0.4)
   a2out2 << fb;                                              // execute on a2out2
}
```

Attributes

The `attribute` type qualifier is used to mark an input variable as "mutable between the vertices of a region", i.e. the variable contains vectors for all vertices of the kernel program output region – such vectors are linearly related[1] to the array elements. Please note that such "vertex-attached data" is given implicitly (i.e. "in advance") as a custom region.

The kernel program's output region (which can be passed via `array2::execute()`) specifies the array elements the kernel program is applied to – it may be more complex than the default region that covers the complete array (i.e. it may contain more than four vertices). In such a case, all other attribute regions that are used in the kernel program must have the same amount of vertices – otherwise, the behavior of the kernel program is undefined.

The `attribute` keyword can be used for all GPU vector classes, i.e. `vec1`, `vec2`, `vec3`, and `vec4`. A basic example for the attribute keyword can be found in section 4.2.4.

[1]A graphical explanation of "linearly related" is as follows: each element is surrounded by two edges of the kernel region (where one edge is on the left and the other edge is on the right side of the element), and each of the two edges is itself defined by two vertices. The value of the attribute at the given array element is computed by its weighted distance to the values attached to the four vertices – see section 4.2.3.

A.4 Built-In Functions and Variables

The GPU^{++} development system provides a variety of built-in functions for vector and matrix types that can be used in GPU-based programming. Developers are encouraged to use these functions rather than to do the equivalent computations on their own, because the built-in functions are assumed to be optimal. For instance, the `mix()` operation is mapped directly to a single instruction on the graphics processor, while a "custom" linear interpolation is made of multiple instructions and therefore is executed much slower.

Many of the functions are known from the C math libraries `<math.h>`, e.g. the trigonometric routines `sin`, `cos`, `tan`. However, they are extended to also support vector and matrix arguments. In the following function declarations, the type `vecType` is used where the argument is a vector – similar for `matType`, which can be any matrix type.

Common Functions

`vecType abs(vecType` a`)` – Absolute Value

Performs a component-wise absolute value operation on the single vector operand:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return a_n if $a_n \geq 0$, otherwise return $-a_n$

```
vec3 v3r = abs(vec3(-1.0, 0.0, 1.0));                 // v3r = (1.0, 0.0, 1.0)
```

`vecType inverse(vecType` a`)` – Reciprocal

The reciprocal of each component of the single vector operand is computed:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $1/a_n$

```
vec3 v3r = inverse(vec3(-1.0, -2.0, -3.0));    // v3r = (-1.0, -0.5, -0.3333..)
```

`vecType floor(vecType` a`)` – Floor

For each component a_n, the largest integer less than or equal to a_n is computed:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\lfloor a_n \rfloor$

```
vec3 v3r = floor(vec3(-1.2, 0.0, 0.9));              // v3r = (-2.0, 0.0, 0.0)
```

`vecType ceil(vecType` a`)` – Ceil

For each component a_n, the smallest integer greater than or equal to a_n is computed:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\lceil a_n \rceil$

```
vec3 v3r = ceil(vec3(-1.2, 0.0, 0.9));               // v3r = (-1.0, 0.0, 1.0)
```

vecType fract(vecType *a*) – Fraction

This function extracts the fractional part of each component of the vector operand:
⇒ for each $n = 0, \ldots, 3$: return $a_n - \lfloor a_n \rfloor$

```
vec3 v3r = fract(vec3(-1.2, 0.0, 0.9));                // v3r = (0.8, 0.0, 0.9)
```

vecType mod(vecType *a*, vecType *b*) – Modulus

Performs a component-wise computation of the division remainder of a by b:
⇒ for each $n = 0, \ldots, 3$: return $a_n - b_n\lfloor a_n/b_n \rfloor)$

```
vec2 v2r = mod(vec2(-2.0, 4.0), vec2(0.9, 0.7));            // v2r = (0.7, 0.5)
```

vecType min(vecType *a*, vecType *b*) – Minimum

This function computes the component-wise minimum of the two operands:
⇒ for each $n = 0, \ldots, 3$: return a_n if $a_n \leq b_n$, otherwise return b_n

```
vec2 v2r = min(vec2(0.2, 4.0), vec2(0.9, 0.7));             // v2r = (0.2, 0.7)
```

vecType max(vecType *a*, vecType *b*) – Maximum

This function computes the component-wise maximum of the two operands:
⇒ for each $n = 0, \ldots, 3$: return a_n if $a_n \geq b_n$, otherwise return b_n

```
vec2 v2r = max(vec2(0.2, 4.0), vec2(0.9, 0.7));             // v2r = (0.9, 4.0)
```

vecType mix(vecType *a*, vecType *b*, vecType *c*) – Linear Interpolation

A component-wise linear interpolation between the first two operands a and b is computed, with the third operand c as the blending factor:
⇒ for each $n = 0, \ldots, 3$: return $a_n + c_n(b_n - a_n)$

```
vec2 v2r = mix(vec2(1.0), vec2(0.0), vec2(0.3, 0.7));       // v2r = (0.7, 0.3)
```

vecType mad(vecType *a*, vecType *b*, vecType *c*) – Multiply And Add

The first two vectors are multiplied component-wise and summed with the third vector:
⇒ for each $n = 0, \ldots, 3$: return $a_n b_n + c_n$

```
vec2 v2r = mad(vec2(0.0, 1.0), vec2(1.0), vec2(0.5));       // v2r = (0.5, 1.5)
```

vecType select(vecType a, vecType b, vecType c) – Select Vector Components

A component-wise comparison of the first operand against zero is performed, where the second or third operand is returned based on the result of the comparison:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return b_n if $a_n < 0$, otherwise return c_n

```
vec2 v2r = select(vec2(-0.1, 0.1), vec2(1.0), vec2(0.5));   // v2r = (1.0, 0.5)
```

vecType threshold(vecType a, vecType b) – Set On Greater Or Equal

Performs component-wise comparison of two operands, if the component of the first operand is greater than or equal to that of the second, `1.0` is returned, otherwise `0.0`:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return 1.0 if $a_n \geq b_n$, otherwise return 0.0

```
vec2 v2r = threshold(vec2(0.5, 1.5), vec2(1.0));             // v2r = (0.0, 1.0)
```

Trigonometric Functions

Function arguments specified as *angle* are assumed to be in units of radians. In no case will any of these functions result in a division-by-zero error. If the divisor of a ratio is 0, the result will be undefined.

vecType radians(vecType a) – Convert Degrees To Radians

Component-wise conversion from degrees ($d \in [0, \ldots, 360]$) to radians ($r \in [0, \ldots, 2\pi]$):
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\pi a_n / 180$

```
vec3 v3r = radians(vec3(0.0, 90.0, 180.0));  // v3r = (0.0, 1.5708.., 3.1416..)
```

vecType degrees(vecType a) – Convert Radians To Degrees

Component-wise conversion from radians ($r \in [0, \ldots, 2\pi]$) to degrees ($d \in [0, \ldots, 360]$):
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $180 a_n / \pi$

```
vec3 v3r = degrees(vec3(0.0, 1.5708.., 3.1416..));  // v3r = (0.0, 90.0, 180.0)
```

vecType sin(vecType a) – Sine

A component-wise approximation of the standard trigonometric sine function of the vector operator. The angle is given in radians, and has to be in the range $[-\pi, \pi]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\sin(a_n)$

```
vec3 v3r = sin(vec3(0.0, 1.5708.., 3.1416..));         // v3r = (0.0, 1.0, 0.0)
```

vecType cos(vecType *a*) – **Cosine**

A component-wise approximation of the standard trigonometric cosine function of the vector operator. The angle is given in radians, and has to be in the range $[-\pi, \pi]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\cos(a_n)$

```
vec3 v3r = cos(vec3(0.0, 1.5708.., 3.1416..));        // v3r = (1.0, 0.0, -1.0)
```

vecType tan(vecType *a*) – **Tangent**

A component-wise approximation of the standard trigonometric tangent function of the vector operator. The angle is given in radians, and has to be in the range $[-\pi, \pi]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\tan(a_n)$

```
vec3 v3r = tan(vec3(0.0, -0.5, 1.0));       // v3r = (0.0, -0.5463.., 1,5574..)
```

vecType asin(vecType *a*) – **Arc Sine**

A component-wise approximation of the standard trigonometric arc sine function of the vector operator. The angle is returned in radians, and is in the range $[-\pi/2, \pi/2]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\arcsin(a_n)$

```
vec3 v3r = asin(vec3(0.0, 1.0, 0.0));        //      v3r = (0.0, 1.5708.., 0.0)
```

vecType acos(vecType *a*) – **Arc Cosine**

A component-wise approximation of the standard trigonometric arc cosine function of the vector operator. The angle is returned in radians, and is in the range $[0, \pi]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\arccos(a_n)$

```
vec3 v3r = acos(vec3(1.0, 0.0, -1.0));       // v3r = (0.0, 1.5708.., 3.1416..)
```

vecType atan(vecType *a*) – **Arc Tangent**

A component-wise approximation of the standard trigonometric arc tangent function of the vector operator. The angle is returned in radians, and is in the range $[-\pi/2, \pi/2]$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\arctan(a_n)$

```
vec3 v3r = atan(vec3(0.0, -0.5463.., 1,5574..));      // v3r = (0.0, -0.5, 1.0)
```

Exponential Functions

vecType pow(vecType a**, vecType** b**)** – Exponentiate

A component-wise computation of the first operand raised to the power of the second operand. Note, that this might be approximated by $a^b = \exp_2(a)\log_2(b)$, thus, it may not be possible to exponentiate correctly with a negative base:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $a_n^{b_n}$

```
vec3 v3r = pow(vec3(4.0, 9.0, 16.0), vec3(0.5));      // v3r = (2.0, 3.0, 4.0)
```

vecType exp(vecType a**)** – Exponential Naturalis

A component-wise computation of the natural exponentiation of the first operand. This might be approximated using `pow`, thus, note the restrictions of the `pow` function:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return e^{a_n}

```
vec2 v2r = exp(vec2(2.0, 3.0));                  // v2r = (7.3891.., 20.08554..)
```

vecType log(vecType a**)** – Logarithm Naturalis

A component-wise computation of the natural logarithm of the first operand:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\ln(a_n)$

```
vec2 v2r = log(vec2(7.3891.., 20.08554..));                  // v2r = (2.0, 3.0)
```

vecType exp2(vecType a**)** – Exponential Base-2

A component-wise computation of the base-2 exponentiation of the first operand. This might be approximated using `pow`, thus, note the restrictions of the `pow` function:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return 2^{a_n}

```
vec2 v2r = exp2(vec2(2.0, 3.0));                             // v2r = (4.0, 8.0)
```

vecType log2(vecType a**)** – Logarithm Base-2

A component-wise computation of the base-2 logarithm of the first operand:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\log_2(a_n)$

```
vec2 v2r = log2(vec2(4.0, 8.0));                             // v2r = (2.0, 3.0)
```

vecType sqrt(vecType a**)** – Positive Square Root

This function performs a component-wise computation of the positive square root of the first operand. Results are undefined if $x < 0$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $\sqrt{a_n}$

```
vec2 v2r = sqrt(vec2(4.0, 9.0));                    // v2r = (2.0, 3.0)
```

vecType inversesqrt(vecType a**)** – Reciprocal Of The Positive Square Root

This function performs a component-wise computation of the reciprocal of the positive square root of the first operand. Results are undefined if $x \leq 0$:
$\Rightarrow$ for each $n = 0, \ldots, 3$: return $1.0/\sqrt{a_n}$

```
vec2 v2r = inversesqrt(vec2(4.0, 9.0));         // v2r = (0.5, 0.3333..)
```

Vector Functions

vec1 length(vecType a**)** – Length of a Vector

Computes the length of the specified vector, hence, a one-component vector is returned:
$\Rightarrow$ return $\sqrt{a_0^2 + \ldots + a_3^2}$

```
vec1 v1r = length(vec3(1.0, 2.0, 3.0));             // v1r = (3.7417..)
```

vec1 distance(vecType a**, vecType** b**)** – Distance Between Vectors

Computes the distance between the first and the second operand yielding to a scalar:
$\Rightarrow$ return $\sqrt{(a_0 - b_0)^2 + \ldots + (a_3 - b_3)^2}$

```
vec1 v1r = distance(vec3(1.0, 2.0, 3.0), vec3(0.5));        // v1r = (2.9580..)
```

vec1 dot(vecType a**, vecType** b**)** – Dot Product

Generates a scalar by computing the dot product of the first and the second operand:
$\Rightarrow$ return $a_0b_0 + \ldots + a_3b_3$

```
vec1 v1r = dot(vec3(1.0, 2.0, 3.0), vec3(0.5));                // v1r = (3.0)
```

`vec3 cross(vec3` a`, vec3` b`)` – Cross Product

Generates a three-component vector by computing the cross product of the first and the second operand. The cross product is only defined for three-component vectors:
$\Rightarrow$ return $(a_1b_2 - b_1a_2, a_2b_0 - b_2a_0, a_0b_1 - b_0a_1)$

```
vec3 v3r = cross(vec3(1.0,0.0,0.0), vec3(0.0,1.0,0.0));  // v3r = (0.0,0.0,1.0)
```

`vecType normalize(vecType` a`)` – Normalize Vector

Returns a vector with unit length, pointing at the same direction as the first operand:
$\Rightarrow$ return a/`length`(a)

```
vec3 v3r = normalize(vec3(1.0, 2.0, 3.0)); // v3r = (0.267.., 0.534.., 0.802..)
```

Matrix Functions

`matType matrixCompMul(matType` a`, matType` b`)` – Component-Wise Matrix Product

Returns the component-wise multiplication of all elements of a with all elements of b:
$\Rightarrow$ for each $i, j = 0, \ldots, 3$: return $a_{ij}b_{ij}$

```
mat2 m2r = matrixCompMul(mat2(vec2(1.0, 2.0), vec2(-1.0, 1.0)),
                         mat2(vec2(0.5, 1.0), vec2(0.2, -0.9)));
                                      // m2r = ((0.5, 2.0), (-0.2, -0.9))
```

`matType matrixCompDiv(matType` a`, matType` b`)` – Component-Wise Matrix Division

Returns the component-wise division of all elements of a with all elements of b:
$\Rightarrow$ for each $i, j = 0, \ldots, 3$: return a_{ij}/b_{ij}

```
mat2 m2r = matrixCompDiv(mat2(vec2(1.0, 2.0), vec2(-1.0, 1.0)),
                         mat2(vec2(0.5, 1.0), vec2(0.2, -0.9)));
                                 // m2r = ((2.0, 2.0), (-5.0, -1.1111..))
```

Variables

There are some constant and uniform variables that can be used from within any GPU program. Such variables are initialized automatically. However, there will be no overhead if they are not used within the program. The following variables can be used:

`vec1 const m_v1pi` – The Pi Value

A one-component vector holding $\pi = 3.14159265358979323846264338327 95$.

`vec1 const m_v1euler` – The Euler Value

A one-component vector holding $e = 2.71828182845904523536028747135 27$.

`uniform< vec2 > m_v2dim` – Dimensions of the Output Array

A two-component vector, where the first component (`m_v2dim.x()`) holds the width, while the second component (`m_v2dim.y()`) contains the height of the output array.

`uniform< vec2 > m_v2step` – Step-Size in the Output Array

A two-component vector, where the first component (`m_v2step.x()`) holds the reciprocal of the width, while the second component (`m_v2step.y()`) contains the reciprocal of the height of the output array. This is useful to find elements near a given position.

Appendix B

Examples

This appendix contains basic programs that have been implemented using the GPU^{++} development system. In contrast to the more complex algorithms of chapter 7 the following sections contain the complete source code together with a line-by-line description.

B.1 Hello World

One of the most basic examples is presented in listing B.1 – the position of each element in the output array is stored to that element. After the inclusion of the GPU^{++} header files

```
#include <kernel.hpp>
#include <array2.hpp>
#include <opengl.hpp>

using namespace gpp;

int main(int, char * *)
{
   struct hello_world : public kernel< opengl > {

      virtual vec4 element(attribute< vec4 > const & v4pos) const
      {
         return v4pos;                                     // return the output element position
      }
   };

   array2< float, 3 > a2out(256, 256);            // create a three-component array of size 256 x 256
   a2out << hello_world();                          // execute an instance of the kernel on the array
   return 0;
}
```

Listing B.1: Store the position for each element of the output array.

(line 1–3), the namespace `gpp` is made available. The kernel program `hello_world` (lines 9–15) implements the `element()` method that returns the input position without further processing (line 11–14). The `main()` function instantiates an array `a2out` of 256×256 three-component vectors of type `float` (line 17) and executes a kernel program instance on that array (line 18). The resulting content of `a2out` is shown in figure B.1 on page 117.

B.2 Array Access

Listing B.2 shows an extended version of the aforementioned example, where an input array is accessed in the `element()` method of a kernel program. This is done using sampler `s2input` (line 12) and the *array subscript* operator `[]` (line 16). Please note that the input position `v4pos` is swizzled to perform a mirroring of the input array along its diagonal axis.

In `main()`, an input array `a2in` is created from an image file (line 22) and is attached to the aforementioned sampler (line 23). The resulting `a2out` content is shown in figure B.2.

```
 1 #include <kernel.hpp>
 2 #include <array2.hpp>
 3 #include <sampler2.hpp>
 4 #include <opengl.hpp>
 5
 6 using namespace gpp;
 7
 8 int main(int, char * *)
 9 {
10    struct array_access : public kernel< opengl > {
11
12       sampler2< float, 3 > s2input;                      // sampler to access an attached input array
13
14       virtual vec4 element(attribute< vec4 > const & v4pos) const
15       {
16          return s2input[v4pos.yx()];        // sample input (diagonally mirrored) and return result
17       }
18    };
19
20    array_access aa;                                               // instantiate the kernel program
21
22    array2< float, 3 > a2in("images/lenna.bmp");       // create input array from the given BMP image
23    aa.s2input.setArray(a2in);                       // attach the input array to the kernel's sampler
24
25    array2< float, 3 > a2out(256, 256);           // create a three-component array of size 256 x 256
26    a2out << aa;                                     // and execute the kernel instance on that array
27    return 0;
28 }
```

Listing B.2: The input image `a2in` is mirrored along its diagonal axis.

B.3 Array Rotation

Very similar to the previous example, the program in listing B.3 accesses an input array `a2in` via sampler object `s2input` to create the output array. But this time the input array is rotated by the specific angle `v1a` (line 13) – which is declared as `uniform` and therefore can be changed without update and re-compilation of the kernel program (line 29).

To perform the rotation around the array's center, the input position `v4pos` needs to be translated (line 17) and then transformed using the known rotation equations (line 18–19). Finally, the back-translated rotated position is used to access the input array (line 20).

The `main()` function instantiates the kernel program and initializes its sampler (line 24–26) – linear interpolation is used to achieve better results. Finally, the rotation angle is set (line 29) and the kernel is executed (line 30). Figure B.3 shows the resulting `a2out`.

```
 1  #include <kernel.hpp>
 2  #include <array2.hpp>
 3  #include <sampler2.hpp>
 4  #include <opengl.hpp>
 5
 6  using namespace gpp;
 7
 8  int main(int, char * *)
 9  {
10     struct rotate : public kernel< opengl > {
11
12        sampler2< float, 3 > s2input;                        // sampler to access an attached input array
13        uniform< vec1 > v1a;                                           // the rotation angle (in radiants)
14
15        virtual vec4 element(attribute< vec4 > const & v4pos) const
16        {
17           vec2 v2off = v4pos.xy() - 0.5;                     // compute position relative to the center
18           vec1 v1x = cos(v1a) * v2off.x() - sin(v1a) * v2off.y();            // the rotated x position
19           vec1 v1y = sin(v1a) * v2off.x() + cos(v1a) * v2off.y();            // the rotated y position
20           return s2input[vec2(v1x, v1y) + 0.5];       // shift the position back to center and sample
21        }
22     };
23
24     rotate rot;                                                    // instantiate the kernel program
25     rot.s2input.setArray(array2< float, 3 >("images/mandrill.bmp"));          // attach an input image
26     rot.s2input.setSampling(SAMPLE_LINEAR);                             // linear interpolation is used
27
28     array2< float, 3 > a2out(256, 256);            // create a three-component array of size 256 x 256
29     rot.v1a = 0.52359877559829887307710723054658;       // set rotation angle to 30 degree (*PI/180.0)
30     a2out << rot;                                     // and execute the kernel instance on that array
31     return 0;
32  }
```

Listing B.3: The input array is rotated by the uniform angle `v1a`.

B.4 Blending of Two Arrays

The final example – in listing B.4 – combines two arrays using a circular blending function. The kernel program uses two samplers to access both arrays (line 12–13). The blending function computes the quadratic distance to the array's center (line 17–18). Then, the quadratic distance `v1dis` is scaled (line 20) and clamped (line 21), which equals to a ramp from 0 to 1 for $0.2 \leq$ `v1dis` ≤ 0.22. Please note, that all scaling parameters can be specified as `uniform` to increase the flexibility. Finally, `mix()` is used to blend the arrays (line 23).

In `main()`, the kernel program is instantiated and its samplers are initialized with two imported images (line 27–29). The resulting content of `a2out` is illustrated in figure B.4.

```
 1 #include <array2.hpp>
 2 #include <sampler2.hpp>
 3 #include <kernel.hpp>
 4 #include <opengl.hpp>
 5
 6 using namespace gpp;
 7
 8 int main(int, char * *)
 9 {
10    struct blend_array : public kernel< opengl > {
11
12       sampler2< float, 3 > s2input1;                        // sampler to access the first input array
13       sampler2< float, 3 > s2input2;                       // sampler to access the second input array
14
15       virtual vec4 element(attribute< vec4 > const & v4pos) const
16       {
17          vec2 v2off   = v4pos.xy() - 0.5;                   // compute position relative to the center
18          vec1 v1dis   = v2off.x() * v2off.x() + v2off.y() * v2off.y();   // find (center distance)^2
19
20          vec1 v1scale = (v1dis - vec1(0.20)) / vec1(0.02);            // rescale the center distance
21          vec1 v1blend = min(max(v1scale, vec1(0.0)), vec1(1.0));    // and clamp against 0.0 and 1.0
22
23          return mix(v1blend, s2input1[v4pos.xy()], s2input2[v4pos.xy()]);       // blend both arrays
24       }
25    };
26
27    blend_array ba;                                                // instantiate the kernel program
28    ba.s2input1.setArray(array2< float, 3 >("images/peppers.bmp"));      // load and attach 1st image
29    ba.s2input2.setArray(array2< float, 3 >("images/lenna.bmp"));        // load and attach 2nd image
30
31    array2< float, 3 > a2out(256, 256);           // create a three-component array of size 256 x 256
32    a2out << ba;                                     // and execute the kernel instance on that array
33    return 0;
34 }
```

Listing B.4: Two input arrays are combined using a circular blending function.

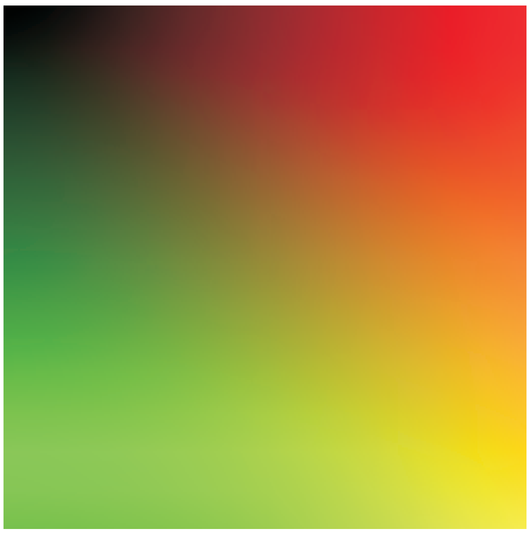


Figure B.1: The `a2out` content of listing B.1 interpreted as an RGB image – the vertical position of each output array element is mapped to the *red* component, while the horizontal position is mapped to *green*.

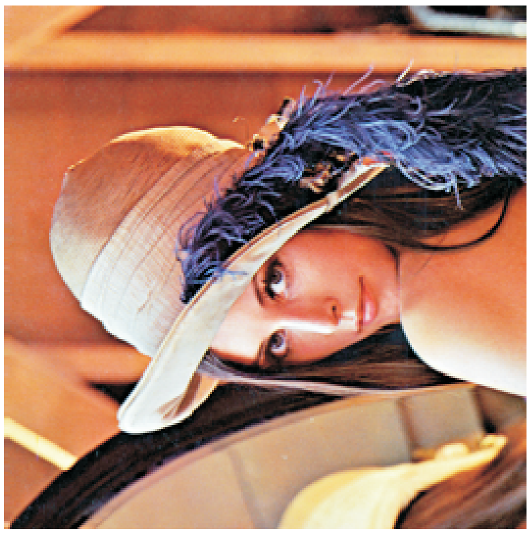


Figure B.2: Result of listing B.2, where the famous *Lenna* input image is stored in output array `a2out`. The mirroring along the diagonal axis is achieved by swapping the x and y components while sampling.

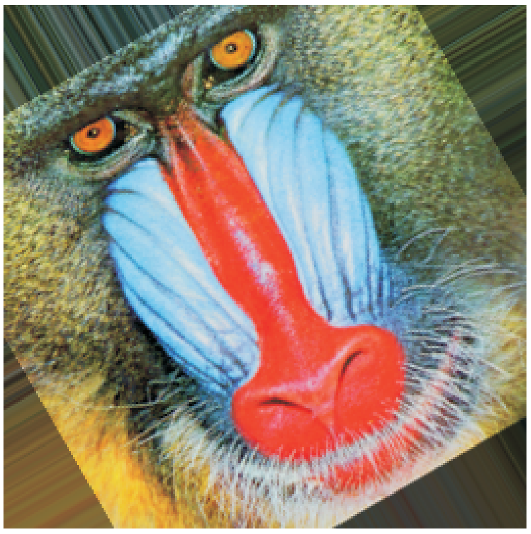


Figure B.3: Listing B.3 generates a rotated version of the input array, where the rotation depends on the uniform variable `v1a` ($= 30°$). Please note the "border artifacts" due to a *clamped* array sampling.

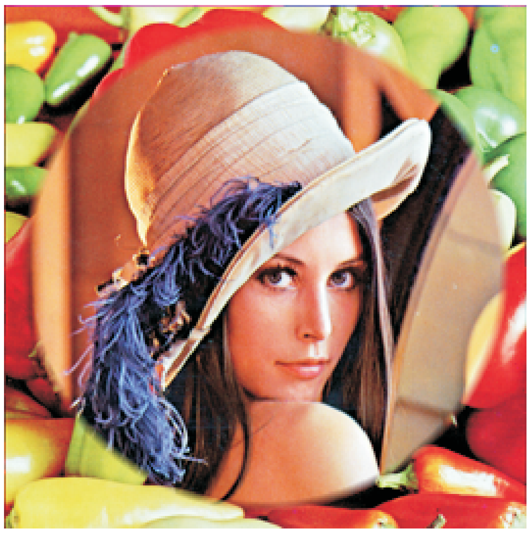


Figure B.4: The program in listing B.4 performs a smooth blending of two arrays to create the final `a2out`. The blending depends on the Euclidean distance of each element to the output array's center.

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C^{++} Template Metaprogramming – Concepts, Tools and Techniques from Boost and Beyond (C^{++} in Depth Series).* Addison-Wesley Professional, 2004.

[2] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* U. S. Department of Commerce, 1964.

[3] A. Alexandrescu. *Modern C^{++} Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, 2001.

[4] A. H. Andersen and A. C. Kak. Simultaneous algebraic reconstruction technique (SART): A superior implementation of the ART algorithm. *Ultrasonic Imaging*, 6:81–94, 1984.

[5] J. Arndt. The FXT library: Fast transforms and low level algorithms. As source-code, 2006. Source code is online available at `http://www.jjj.de/fxt/fxtpage.html`.

[6] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computing Conference (AFIPS'68 Proceedings)*, pages 307–314, 1968.

[7] A. V. Bhatt and Intel Corporation. Creating a PCI Express interconnect. White Paper, 2002. Online available at `http://www.pcisig.com/specifications/pciexpress/technical_library/pciexpress_whitepaper.pdf`.

[8] OpenGL Architecture Review Board. `ARB_fragment_program`. OpenGL Extension, 2002. Online available at `http://www.opengl.org/registry/specs/ARB/fragment_program.txt`.

[9] OpenGL Architecture Review Board. `ARB_vertex_program`. OpenGL Extension, 2002. Online available at `http://www.opengl.org/registry/specs/ARB/vertex_program.txt`.

[10] OpenGL Architecture Review Board. `ARB_vertex_buffer_object`. OpenGL Extension, 2003. Online available at `http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt`.

[11] OpenGL Architecture Review Board. `EXT_framebuffer_object`. OpenGL Extension, 2005. Online available at http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt.

[12] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional, second edition, 2005.

[13] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[14] B. W. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. Winsor Brown, S. Chulani, and C. Abts. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[15] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *Computer Graphics (SIGGRAPH'03 Proceedings)*, pages 917–924, July 2003.

[16] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Conference on Programming Language Design and Implementation (SIGPLAN'94 Proceedings)*, pages 159–170, 1994.

[17] E. O. Brigham. *The Fast Fourier Transform and its applications*. Prentice Hall, 1988.

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[19] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Symposium on Volume Visualization (VVS'94 Proceedings)*, pages 91–98, October 1994.

[20] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Department of Computer Science, University of Utah, December 1974.

[21] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[22] J. Cocke. Global common subexpression elimination. In *Symposium on Compiler Optimization ('70 Proceedings)*, pages 20–24, July 1970.

[23] G. Colvin. Smart pointers for C^{++} garbage collection. *C/C^{++} Users Journal*, 13(12), December 1995.

[24] R. L. Cook. Shade trees. In *Computer Graphics (SIGGRAPH'84 Proceedings)*, volume 18, pages 223–231, July 1984.

[25] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, April 1965.

[26] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *ACM Symposium on Theory of Computing ('87 Proceedings)*, pages 1–6, 1987.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[28] Nullstone Corporation. Nullstone optimization categories, 2006. Online available at http://www.nullstone.com/htmls/category.htm.

[29] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, Ja. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Super Computing (SC'03 Proceedings)*, November 2003.

[30] S. R. Deans. *The Radon Transform and Some of Its Applications*. Wiley, 1983.

[31] J. J. Dongarra, J. du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[32] S. Du Toit. Sh. Software Library, 2006. Online available at http://libsh.org.

[33] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH'04 Proceedings)*, pages 133–138, 2004.

[34] L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America*, 1984.

[35] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[36] O. Fluck, S. Aharon, D. Cremers, and M. Rousson. GPU histogram computation. In *ACM SIGGRAPH posters and demos*, 2006.

[37] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[38] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[39] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *ACM/IEEE Conference on Supercomputing ('05 Proceedings)*, November 2005.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[41] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Prentice Hall, second edition, 2002.

[42] R. Gordon, R. Bender, and G. T. Herman. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. *Journal of Theoretical Biology*, 29:471–481, 1970.

[43] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTerasort: high performance graphics co-processor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data ('06 Proceedings)*, pages 325–336, June 2006.

[44] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. Technical Report MSR-TR-2006-108, Microsoft Research, August 2006. Online available at ftp://ftp.research.microsoft.com/pub/tr/TR-2006-108.pdf.

[45] K. Gray. *The Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, 2003.

[46] S. Green. The OpenGL framebuffer object extension. Talk on the Game Developers Conference (GDC'05), 2005. Online available at http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf.

[47] A. Greß and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'06 Proceedings)*, April 2006.

[48] M. Hadwiger, C. Langer, H. Scharsach, and K. Bühler. State-of-the-art report 2004 on GPU-based segmentation. Technical Report TR-VRVIS-2004-17, VRVis Research Center, 2004. Online available at http://medvis.vrvis.at/fileadmin/publications/TR_VRVIS_2004_17.pdf.

[49] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH'90 Proceedings)*, volume 24, pages 289–298, August 1990.

[50] M. Harris. GPGPU. Website, 2006. Online available at http://www.gpgpu.org.

[51] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.

[52] H.-P. Hofstee. Power efficient processor architecture and the Cell processor. In *International Symposium on High-Performance Computer Architecture (HPCA'05 Proceedings)*, volume 11, pages 258–262, February 2005.

[53] D. Horn. Stream reduction operations for GPGPU applications. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 36. Addison-Wesley Professional, 2005.

[54] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Computer Graphics (SIGGRAPH'99 Proceedings)*, pages 277–286, July 1999.

[55] Intel. Intel Math Kernel Library 8.1. Software Library, 2006. Online available at http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm.

[56] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier Volume Rendering on the GPU using a Split-Stream-FFT. In *Vision, Modeling, and Visualization (VMV'04 Proceedings)*, November 2004.

[57] A. C. Kak. Tomographic imaging with diffracting and non-diffracting sources. In *Array Signal Processing*. Prentice-Hall, 1985.

[58] A. C. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. IEEE Press, 1988.

[59] D. Kalev. *ANSI/ISO C^{++} Professional Programmer's Handbook*. Que, 1999.

[60] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *USENIX Security Symposium (SECURITY'99 Proceedings)*, pages 93–98, August 1999.

[61] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[62] J. Kessenich, D. Baldwin, and R. J. Rost. The OpenGL shading language, 2004. Online available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf.

[63] P. Kipfer, M. Segal, and R. Westermann. UberFlow: A GPU-based particle engine. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (EGGH'04 Proceedings)*, pages 115–122, 2004.

[64] P. Kipfer and R. Westermann. Improved GPU sorting. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[65] R. C. Kirby. A new look at expression templates for matrix computation. *Computing in Science and Engineering*, 5(3):66–70, May 2003.

[66] D. Kirk. The future: programmable GPUs & cinematic computing. Presentation at WinHEC'03, 2003. Online available at http://developer.nvidia.com/object/cg_tutorial_teaching.html.

[67] D. E. Knuth. *The Art of Computer Programming, Volume 3 — Sorting and Searching.* Addison-Wesley, second edition, 1998.

[68] P. Kohlmann, S. Bruckner, A. Kanitsar, and E. Gröller. Evaluation of a bricked volume layout for a medical workstation based on Java. Technical Report TR-186-2-06-04, Institute of Computer Graphics and Algorithms, Vienna University of Technology, October 2006. Online available at `http://www.cg.tuwien.ac.at/research/publications/2006/TR-186-2-06-04/`.

[69] J. L. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

[70] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *High Performance Networking and Computing (SC'01 Proceedings)*, November 2001.

[71] A. Lefohn, J. Cates, and R. Whitaker. Interactive, GPU-based level sets for 3D brain tumor segmentation. In *Medical Image Computing and Computer Assisted Intervention (MICCAI'03 Proceedings)*, April 2003.

[72] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *Computer Graphics (SIGGRAPH'90 Proceedings)*, volume 24, pages 327–335, August 1990.

[73] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *Computer Graphics (SIGGRAPH'01 Proceedings)*, volume 35, pages 149–158, August 2001.

[74] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH'87 Proceedings)*, volume 21, pages 163–169, July 1987.

[75] R. G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall PTR, second edition, 2004.

[76] W. R. Mark. Future visualization platform. Panel Presentation at IEEE Visualization (VIS'04), 2004. Online available at `http://www-csl.csres.utexas.edu/users/billmark/talks`.

[77] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Computer Graphics (SIGGRAPH'03 Proceedings)*, volume 37, pages 896–907, July 2003.

[78] T. J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[79] M. D. McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical Report CS-2000-14, University of Waterloo, 2000. Online available at http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/smash.pdf.

[80] M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multicore Applications Conference ('06 Proceedings)*, 2006.

[81] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *Computer Graphics (SIGGRAPH'04 Proceedings)*, volume 38, pages 787–795, August 2004.

[82] M. D. McCool and W. Heidrich. Texture shaders. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (EGGH'99 Proceedings)*, pages 117–126, August 1999.

[83] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (EGGH'02 Proceedings)*, pages 57–68, September 2002.

[84] C. McNairy. Hyper-threading on dual-core Intel Itanium 2 processors. In *Itanium Conference and Expo (ICE'06 Proceedings)*, 2006.

[85] Á. Moravánszky. Dense matrix algebra on the GPU. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware Publishing, Inc., 2003.

[86] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.

[87] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[88] NVIDIA. Image histogram using occlusion queries. Source Code Sample, 2005. Online available at http://download.developer.nvidia.com/developer/SDK/Individual_Samples/gpgpu_samples.html.

[89] NVIDIA. NVIDIA CUDA homepage. Website, 2006. Online available at http://developer.nvidia.com/object/cuda.html.

[90] NVIDIA. NVIDIA GeForce 8800 GPU architecture overview. Technical Brief, 2006. Online available at http://www.nvidia.com/object/IO_37100.html.

[91] M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Computer Graphics (SIGGRAPH'98 Proceedings)*, volume 32, pages 159–168, July 1998.

[92] J. D. Owens. *Computer Graphics On A Stream Architecture.* PhD thesis, Stanford University, November 2002.

[93] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 2007.

[94] R. E. Park. Software size measurement: A framework for counting source statements. Technical report, Software Engineering Institute, Carnegie Mellon University, October 1992. Online available at `http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr20.92.pdf`.

[95] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Symposium on Visualization (VisSym'04 Proceedings)*, February 2004.

[96] PeakStream, Inc. The PeakStream Platform: High productivity software development for multi-core processors. Technote, 2006. Online available at `http://www.peakstreaminc.com/reference/peakstream_platform_technote.pdf`.

[97] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Computer Graphics (SIGGRAPH'00 Proceedings)*, volume 34, pages 425–432, July 2000.

[98] K. Perlin. An image synthesizer. In *Computer Graphics (SIGGRAPH'85 Proceedings)*, volume 19, pages 287–296, July 1985.

[99] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (SIGGRAPH'01 Proceedings)*, volume 35, pages 159–170, August 2001.

[100] T. J. Purcell, C. Donner, M. Cammarano, H. Wann Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (EGGH'03 Proceedings)*, pages 41–50, 2003.

[101] Z. Qin. An embedded shading language. Master's thesis, University of Waterloo, 2004. Online available at `http://etd.uwaterloo.ca/etd/zqin2004.pdf`.

[102] K. R. Rao and P. Yip. *Discrete Cosine Transform - Algorithms, Advantages and Applications.* Academic Press, 1990.

[103] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *ACM/IEEE International Symposium on Microarchitecture (MICRO'98 Proceedings)*, pages 3–13, November 1998.

[104] R. J. Rost. *OpenGL Shading Language.* Addison-Wesley Professional, second edition, 2006.

[105] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley Professional, second edition, 2004.

[106] M. Segal and M. Peercy. A performance-oriented data parallel virtual machine for GPUs. In *Computer Graphics (SIGGRAPH '06 Proceedings)*, 2006.

[107] A. Sherbondy, M. Houston, and S. Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visualization (VIS'03 Proceedings)*, October 2003.

[108] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[109] E. M. Stein and G. Weiss. *Introduction to Fourier Analysis on Euclidean Spaces.* Princeton University Press, 1975.

[110] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[111] B. Stroustrup. Operator overloading in C^{++}. In *Conference on System Implementation Languages: Experience & Assessment ('84 Proceedings)*, September 1984.

[112] B. Stroustrup. *The C^{++} Programming Language.* Addison-Wesley Professional, special third edition, 2000.

[113] R. Strzodka. *Hardware Efficient PDE Solvers in Quantized Image Processing.* PhD thesis, Universität Duisburg-Essen, Campus Duisburg, February 2005.

[114] T. Sumanaweera and D. Liu. Medical image reconstruction with the FFT. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 48. Addison-Wesley Professional, 2005.

[115] C. R. Symons. Function point analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, 14(1):2–11, January 1988.

[116] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Architectural support for programming languages and operating systems (ASPLOS'06 Proceedings)*, 2006.

[117] A. Thall. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH posters and demos*, 2006.

[118] A. Trew and G. Wilson. *Past, Present, Parallel; A Survey of Available Parallel Computing Systems.* Springer-Verlag, 1991.

[119] S. Upstill. *The RenderMan Companion.* Addison Wesley, 1990.

[120] T. L. Veldhuizen. Expression templates. *C^{++} Report*, 7(5):26–31, June 1995. Online available at http://osl.iu.edu/~tveldhui/papers.

[121] T. L. Veldhuizen. Using C^{++} template metaprograms. *C^{++} Report*, 7(4):36–43, May 1995. Online available at http://osl.iu.edu/~tveldhui/papers.

[122] S. Venkatasubramanian. The graphics card as a streaming computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams ('03 Proceedings)*, October 2003.

[123] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21:168–178, 1974.

[124] R. C. Whaley, Aa Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Journal of Parallel Computing*, 27(1–2):3–35, 2001.

[125] T. Whitted and D. M. Weimer. A software testbed for the development of 3D raster graphics systems. *ACM Transactions on Graphics*, 1:43–57, January 1982.

[126] J. W. J. Williams. Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[127] O. Wilson, A. Van Gelder, and J. Wilhelms. Direct volume rendering via 3D textures. Technical Report UCSC-CRL-94-19, University of Califonia, Santa Cruz, 1994.

[128] S. Wolfram. *Mathematica: A System for doing Mathematics by Computer.* Addison-Wesley, second edition, 1991.

[129] E. Wu and Y. Liu. General purpose computation on GPU. *Journal of Computer-Aided Design and Computer Graphics*, 16(5), 2005. Source code is online available at http://lcs.ios.ac.cn/~lyq/demos/opensource/software.html.

[130] F. Xu and K. Mueller. Ultra-fast 3D filtered backprojection on commodity graphics hardware. In *IEEE International Symposium on Biomedical Imaging (ISBI'04 Proceedings)*, pages 571–574, April 2004.

[131] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, 2005.