Dissertation

SIMD Code Generation in Data-Parallel Programming

Nicolas Fritz

Saarbrücken, 01.06.2009

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

Tag des Kolloquiums:	21.10.2009
Dekan:	Prof. Dr. Joachim Weickert
Prüfungsausschuss:	
Vorsitzender:	Jun.-Prof. Dr. Sebastian Hack
Gutachter:	Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm Prof. Dr. Philipp Slusallek
Akademischer Mitarbeiter:	Dr. Philipp Lucas

Impressum

Herstellung und Verlag: epubli GmbH, Berlin, www.epubli.de
Printed in Germany
ISBN: 978-3-86931-240-8

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.d-nb.de` abrufbar.

Abstract

Today's desktop PCs feature a variety of parallel processing units. Developing applications that exploit this parallelism is a demanding task, and a programmer has to obtain detailed knowledge about the hardware for efficient implementation. CGiS is a data-parallel programming language providing a unified abstraction for two parallel processing units: graphics processing units (GPUs) and the vector processing units of CPUs. The CGiS compiler framework fully virtualizes the differences in capability and accessibility by mapping an abstract data-parallel programming model on those targets. The applicability of CGiS for GPUs has been shown in previous work; this work focuses on applying the abstract programming model of CGiS to CPUs with SIMD (Single Instruction Multiple Data) instruction sets. We have identified, adapted and implemented a set of program analyses to expose and access the available parallelism. The code generation phase is based on selected optimization algorithms tailored to SIMD code generation. Via code generation profiles, it is possible to adapt the code generation strategy to different target architectures. To assess the effectiveness of our approach, we have implemented backends for the two most widespread SIMD instruction sets, namely Intel's Streaming SIMD Extensions and Freescale's AltiVec. Additionally, we integrated a prototypical backend for the Cell Broadband Engine as an example for a multi-core architecture. Our experimental results show excellent average performance gains by a factor of 3 compared to standard scalar C++ implementations and underline the viability of this approach: real-world applications can be implemented easily with CGiS and result in efficient code.

Zusammenfassung

Parallelverarbeitung wird heutzutage in handelsüblichen PCs von einer Reihe verschiedener Komponenten unterstützt. Grafikprozessoren (GPUs) und Vektoreinheiten in CPUs sind zwei dieser Komponenten. Da die Entwicklung von Anwendungen, die diese Parallelität nutzen, eine anspruchsvolle Aufgabe ist, muss sich ein Programmierer detaillierte Kenntnisse der internen Hardwarestruktur aneignen. Mit CGiS stellen wir eine datenparallele Programmiersprache vor, die eine gemeinsame Abstraktion für Grafikprozessoren und Vektoreinheiten in CPUs bietet und ein einheitliches Programmiermodell für beide bereitstellt. In vorherigen Arbeiten haben wir bereits die Nutzbarkeit von CGiS für GPUs gezeigt. In der vorliegenden Arbeit bilden wir das abstrakte Programmiermodel von CGiS auf CPUs mit SIMD (Single Instruction Multiple Data) Instruktionssatz ab. Wir haben eine Reihe relevanter Programmanalysen angepasst und implementiert um Parallelität aufzudecken und zu nutzen. Die Codegenerierungsphase basiert auf ausgewählten Optimierungsalgorithmen, die speziell auf die Generierung von SIMD-Code zugeschnitten sind. Durch Profile für verschiedene Architekturen ist es uns möglich, die Codegenierung zu steuern. Um die Effektivität unseres Ansatzes unter Beweis zu stellen, haben wir Backends für die beiden am weitesten verbreiteten SIMD-Instruktionssätze implementiert: Die „Streaming SIMD Extensions“ von Intel und AltiVec von Freescale. Zusätzlich haben wir ein prototypisches Backend für den Cell Prozessor von IBM, als Beispiel für eine Multi-Core-Architektur, integriert. Die Ergebnisse unserer Experimente belegen eine ausgezeichnete durchschnittliche Beschleunigung um einen Faktor von 3 im Vergleich zu handgeschriebenen C++-Implementierungen. Diese Resultate untermauern unseren Ansatz: Mittels CGiS lässt sich leistungsstarker Code für SIMD- und Multi-Core-Applikationen generieren.

Extended Abstract

The development of desktop processor technology is moving away from a constant increase in clock rate, towards increased parallelism. Recent PCs contain a set of parallel processing units, the two most prominent of which are the Graphic Processing Units (GPUs), and the SIMD (Single Instruction Multiple Data) units in CPUs. While this parallelism is available, its exploitation is still a demanding task. Compilers for traditional sequential programming languages like C, C++ or Fortran include a multitude of vectorization techniques, i. e. methods to find parallelism inside a sequential program and possibly transform it to fit the parallel hardware while preserving the program's semantics. Yet the success of these methods remains marginal. Direct programming via assembler or intrinsics, on the other hand, requires detailed knowledge of the hardware intricacies, and afflicts maintainability and portability. The source of the problem is the use of a sequential programming language. Therefore we designed a data-parallel programming language based on a stream programming model called CGiS. It offers a unified abstraction for GPUs and the vector processing units of CPUs. The language features explicit parallelism, rendering the search for parallelism and some program transformations to access it, redundant. The task of a CGiS compiler is less finding parallelization opportunities than just mapping them to the hardware. This allows for generating efficient code.

In earlier work we have presented a GPU backend for the CGiS compiler and demonstrated that CGiS is a viable language for data-parallel programming. There are two kinds of parallelism in CGiS: parallel loops iterating over streams and small vectorial data types. The latter stem from CGiS' intended closeness to graphics hardware programming languages. The mapping of these two types of parallelism to GPU hardware is a one-to-one mapping as they contain up to 128 units all working on 4-component data registers. This thesis focuses on CPUs with SIMD units covering the most relevant architectures, namely Intel's Streaming SIMD Extensions and Freescale's AltiVec and, as an example for multi-core architectures, the IBM's Cell Broadband Engine. In contrast to GPUs, the SIMD units presented here only contain the 4-way parallelism introduced by the SIMD registers. We have developed a special transformation called *kernel flat-*

tening: by breaking down vector-type data structures and operations to scalar ones, the parallelism induced by parallel stream iterations can be mapped to SIMD hardware. Another difference between SIMD CPUs and GPUs is the presence of caches. GPUs have a fast texture memory attached, while CPUs use caches to hide memory latencies and increase the access speed of re-used data. For algorithms with neighboring operations, e. g. accesses to adjacent stream elements in two-dimensional streams, caches have to be utilized to maintain increased performance with a CGiS implementation. We integrated a form of *loop sectioning* to achieve this goal. In conjunction with kernel flattening the *control flow conversion* becomes crucial, i. e. control dependencies are replaced by data-dependencies. In the SIMD backends of the CGiS compiler, if and loop-constructs are fully converted to enable parallel execution. To show the viability of CGiS for implementing applications for CPUs featuring SIMD units, we examine six real-world examples. These examples comprise data-parallel algorithms from the fields of mathematical exploration, computer graphics, financial simulation, etc. A standard C++ implementation is then compared to parallel implementations generated with the CGiS compiler. Speedups by a factor of 3 and more can be observed.

Additionally, we examined the performance of CGiS and its compiler on multi-core architectures. As a representative, the Cell Broadband Engine was chosen. It contains a Power Processing Entity (PPE), which is based on the Power Architecture, and eight Synergistic Processing Elements (SPEs). The SPEs are connected to the PPE, each other and the main memory via the Element Interconnection Bus (EIB) and can work independently. Having so much computational power at hand, the distribution of the computations to the single entities is very important. We have investigated a static and a dynamic approach to balance the work load on the SPEs in threaded execution and compared those to standard execution on the PPE by reutilizing the examples mentioned above. For CGiS multi-core solutions tangible speedups can be observed, ranging from a factor of 1.5 to a factor of 70, depending on the characteristics of the data-parallel algorithm. To be effective very high arithmetic density is needed.

Erweiterte Zusammenfassung

In der Entwicklung von Prozessoren (CPUs) für PCs gibt es einen Trend weg von der Erhöhung der Taktrate hin zur Erweiterung der Parallelität. Es gibt verschiedene Arten paralleler Hardware in Arbeitsplatzrechnern. Die beiden bekanntesten sind Grafikprozessoren (GPUs) auf Grafikkarten und SIMD-Einheiten (Single Instruction Multiple Data) in CPUs. Diese parallele Rechenleistung ist zwar vorhanden, ihre Nutzung jedoch ein schwieriges Unterfangen. Handelsübliche Übersetzer für sequentielle Programmiersprachen wie C, C++ oder Fortran besitzen eine Vielzahl verschiedenster Vektorisierungstechniken um Parallelität innerhalb eines sequentiellen Programms aufzudecken und ihn auf die Zielarchitektur abzubilden, ohne dabei die Semantik des Programms zu verändern. Meist bieten diese Methoden allerdings nur geringen Erfolg, da bereits die Benutzung einer sequentiellen Programmiersprache eine schwerwiegende Einschränkung darstellt. Aus diesem Grund haben wir eine datenparallele Programmiersprache mit expliziter Parallelität entworfen, genannt CGiS. Basierend auf einem Datenstrom-Programmiermodell bietet CGiS explizite Parallelität und macht damit das Suchen von Parallelität und damit verbundene Programmtransformationen überflüssig. Die Aufgabe eines Übersetzers für CGiS-Programme besteht also weniger darin Parallelität aufzudecken als sie auf die Hardware abzubilden.

In vorherigen Arbeiten haben wir ein GPU-Backend vorgestellt und erfolgreich gezeigt, dass CGiS eine sinnvolle Sprache zum Implementieren datenparalleler Algorithmen ist. Die Programmiersprache CGiS weist zwei Arten von Parallelität auf: Einerseits parallele Schleifeniterationen über Datenströme und andererseits Vektordatentypen, beschränkt auf maximal vier Komponenten. Letztere wurden wegen CGiS' Nähe zu Programmiersprachen für Grafikprozessoren eingebaut. Das Abbilden dieser Typen von Parallelität auf Grafik-Hardware ist eine Eins-zu-eins-Abbildung, da dort bis zu 128 Einheiten auf Vektorregistern mit vier Komponenten parallel rechnen.

Die vorliegende Arbeit beschäftigt sich in erster Linie mit SIMD-Einheiten und deckt die wichtigsten Architekturen ab, nämlich Intels Streaming SIMD Extensions (SSE), Freescales AltiVec und, als Beispiel für eine Multi-Core-Architektur, IBMs Cell Broadband Engine.

Die hier vorgestellten SIMD-Instruktionssätze umfassen, im Gegensatz zu GPUs, lediglich die von den SIMD-Registern stammende Vierfach-Parallelität. Mit *Kernel Flattening* haben wir eine spezielle Programmtransformation entwickelt, die alle Vektordatentypen und -operationen in skalare Versionen aufbricht. Diese eigens entwickelte Transformation ermöglicht es, die Parallelität der Schleifen über die Datenströme auf die SIMD-Hardware abzubilden.

Während Grafikkarten einen schnellen Texturenspeicher besitzen, verwenden CPUs Caches um die durch die langsame Anbindung an den Hauptspeicher entstehenden Latenzen zu überdecken und den Zugriff auf häufig benutzte Daten zu beschleunigen. Manche datenparallele Algorithmen weisen Operationen auf, die auf benachbarte Elemente im Datenstrom zugreifen, wie z. B. den Zugriff auf angrenzende Elemente in einem zweidimensionalen Feld. Damit SIMD-Implementierungen dieser Algorithmen effizienter sind als sequentielle Lösungen, muss man Caches gezielt nutzen. Wir haben eine Form von *Loop Sectioning*, eine Anpassung der Iterationsfolge, integriert, um diese Aufgabe zu erfüllen.

Eine weitere wichtige Code-Transformation, insbesondere in Verbindung mit Kernel Flattening, ist *Control Flow Conversion*, d. h. bei Schleifen und if-then-else-Konstrukten werden die Kontrollabhängigkeiten durch Datenabhängigkeiten ersetzt. Dadurch wird eine parallele Ausführung dieser Instruktionsfolgen ermöglicht. Wir demonstrieren die Effizienz des mit CGiS erzeugten SIMD-Codes, indem wir sechs Beispielanwendungen untersuchen. Diese stammen aus den verschiedensten Anwendungsgebieten, wie z. B. komplexe mathematische Berechnungen, Computergrafik, Finanzsimulation usw. Eine Standard-C++-Implementierung wird hierbei mit der CGiS-Variante verglichen und wir können eine Leistungssteigerung von Faktor 3 und mehr beobachten.

Zusätzlich haben wir auch die Nutzung von CGiS zur Implementierung von Applikationen für Multi-Core-Architekturen untersucht. Stellvertretend wurde der „Cell Broadband Engine“-Prozessor gewählt. Er umfasst eine „Power Processing Entity“ (PPE), basierend auf der Power Architecture, und acht „Synergistic Processing Elements“ (SPEs). Die SPEs sind über den „Element Interconnection Bus“ (EIB) untereinander, mit der PPE und mit dem Hauptspeicher verbunden. Bei dieser Art von Architektur ist die Verteilung der Berechnungen auf die einzelnen Einheiten besonders wichtig. Wir haben zwei verschiedene Ansätze implementiert, nämlich die statische und die dynamische Verteilung der Berechnungen. Deren Laufzeiten haben wir mit der Standard-C++-Ausführung auf der PPE anhand der oben erwähnten Beispiele verglichen. Für den Cell-Prozessor kann man Leistungssteigerungen beobachten, die von einem Faktor von 1,5 bis zu einem Faktor von 70 reichen, je nach den Eigenschaften des datenparallelen Algorithmus. Zur effizienten Nutzung dieser Multi-Core-Architektur mit CGiS muss ein Algorithmus eine sehr hohe arithmetische Dichte aufweisen.

Acknowledgements

First, I want to thank *Reinhard Wilhelm* for allowing me to pursuit this interesting topic. He provided me with vital support whenever needed but, on the other hand, left me enough freedom to find my part in the CGiS project. The same holds for *Philipp Slusallek*.

There were many people involved in the completion of this work. The biggest influence was surely the cooperation with *Philipp Lucas*, many thanks to him. We faced a few obstacles together in the development of CGiS but managed to tackle or bypass most of them. He helped me not to lose focus on my research and I will miss our discussions about the fall of the occident, raiding the coffee dispenser, and making a living as a carpet dealer.

Furthermore, I want to thank the people at the chair for compiler construction and *AbsInt GmbH*, especially *Daniel Kästner* for proof-reading and being very patient in the process. Also, I want to thank *Kate Flynn* for bringing back punctuation to this thesis and *Emilie Cherlet* for improving the German parts.

Last but not least I want to thank my parents, *Walter* and *Ingrid*. Their continuous support, encouragement, and belief in me carried me through some setbacks and helped me find my way.

Contents

List of Figures

List of Tables

List of Programs

CHAPTER 1

Introduction

1.1. SIMD Code Generation for Multi-Media Instruction Sets

Recent hardware development moves away from the traditional clock speed increase towards increased parallelism. Parallel computing was mostly applied in scientific research and powerful parallel processing hardware has been developed since the 1970s. With the popular demand for multi-media applications in the 1990s, SIMD units were integrated into common desktop PC processors; the most widespread of which are Intel's Streaming SIMD Extensions (SSE) for the IA32 and Freescale's AltiVec for the PowerPC architecture. **S**ingle **I**nstruction **M**ulitple **D**ata units operate on short vector data types and can perform the same operation on all elements of the operand vectors in parallel. Even the modern Cell Broadband Engine from IBM features a PowerPC core with AltiVec support and up to eight additional processing elements that only operate on SIMD vectors.

While desktop PCs were enhanced with SIMD units, the development of Graphics Processing Units (GPUs) lead to a highly parallel architecture with up to 128 independent units. The low purchase costs of GPUs opened up the world of parallel computing to the average Joe programmer and the interest for scientific parallel programming is on the rise after a stagnation in the late 1990s. A trend called GPGPU, general purpose programming for GPUs, emerged and bred a series of programming languages and compilers to make best use of the graphics hardware for applications outside the graphics domain.

Desktop processors, on the other hand, are bound to be general purpose. Usually, applications for those processors are developed in sequential programming languages like C, C++ or Java. Unfortunately, finding parallelism in sequential programs is a difficult task. The requirement for identifying operations that can be executed in parallel is to exclude data-dependencies within those operations. Many ways of finding and eliminating those dependencies are integrated within most popular compilers such as the Gnu C Compiler (gcc), the Intel C Compiler (icc) or the Microsoft Visual Studio

C Compiler (msvc). Still, exploiting parallelism and the automatic use of SIMD hardware remains marginal. It boils down to the initialization of arrays in loops or simple arithmetic operations like addition or multiplication of data arrays. So even if a highly parallel algorithm is implemented in a sequential language, available parallel processing hardware cannot be exploited to its full potential. The remaining alternative is to program SIMD hardware with assembly coding or compiler intrinsics[1]. Unfortunately this kind of programming is uncomfortable and error-prone, but because of the lack of suitable automatic SIMD code generation, it is often the last resort for programmers. A successful way to ensure the most efficient use of SIMD hardware would be the use of a parallel programming language with a compiler able to generate SIMD code.

The general concept of having one programming language to implement all kinds of solutions is flawed in itself because of the differences in parallel and sequential hardware. While parallel hardware offers great computation power, it performs poorly when it comes to control-intense applications such as control automata implementations. In contrast sequential hardware is perfectly suited for task switching and diverging control flow but cannot compete with parallel processors on algorithms with high arithmetic density. The parallel processing hardware of interest here are GPUs and CPUs with SIMD units. By enhancing desktop CPUs with SIMD units, one tries to maintain high flexibility without lacking parallel processing power. IBM evolves that specialization concept by designing the core of its Cell BE in exactly this manner: a PowerPC CPU is connected to eight parallel computing processors operating on SIMD registers.

Cutting edge GPU hardware offers hundreds of floating-point units operating in parallel on SIMD vectors as well as scalar data types. Thus, GPUs can even execute scalar operations in parallel, offering heterogeneous parallelism. Various generations of Intel Pentiums and PowerPCs only feature up to three 4-way SIMD vector processing units. This means that GPUs offer both SIMD parallelism in a single element and across a multitude of elements, whereas only the element-wise parallelism is exploitable by SIMD CPUs.

The key to high performance and efficient code is not only a specialization of hardware, but also a specialization of programming languages. When the programming is styled for parallel algorithms, compilers for that language can generate efficient parallel code by design. This thought was one of our core ideas in the development of the data-parallel programming language CGiS.

The CGiS system strives to open up the parallel programming capabilities of commodity hardware to ordinary programmers. It consists of a data-parallel programming language, a compiler and a set of runtime libraries for the different target architectures. The abstraction level of the language is raised high enough for the developer to be kept away from all hardware intricacies. A CGiS program consists of parallel forall-loops iterating over streams of data and sequential kernels called from those loops. The CGiS framework supports CPU as well as GPU targets, exploiting their characteristics automatically. For GPUs we have shown this in [33, 32]. This dissertation focuses on the SIMD backend of the CGiS compiler. It supports code generation for the aforemen-

[1] Encapsulating functions that represent single or multiple assembler instructions.

tioned SIMD units of Intel's IA32 and Freescale's PowerPC architecture, as well as the Cell Broadband Engine.

The CGiS programming language exhibits two levels of explicit parallelism: large scale SPMD (**S**ingle **P**rogram, **M**ultiple **D**ata) parallelism by the iteration over streams, and small scale SIMD parallelism by vectorial data types. A CGiS backend needs to map these parallelisms to the ones offered by the target architecture. For GPUs this is a one-to-one mapping, as GPUs contain many parallel computation units all working on vector operands; for SIMD CPU architectures the backend has to choose which parallelism opportunity to map to the hardware peculiarities.

This work examines the possibilities of exploiting SIMD parallelism with the data-parallel programming language CGiS. Common multi-media instruction sets are abstracted as target architectures for the CGiS compiler by integrating three new backends:

- ☐ Streaming SIMD Extensions
- ☐ AltiVec
- ☐ Cell Broadband Engine

With the addition of a prototypical backend for the Cell BE processor, the potential of using CGiS as a programming language for multi-core architectures is inspected. Its instruction set is similar to AltiVec. So far CGiS has only been evaluated on graphics processing units but here approached SIMD hardware poses other problems when it comes to generating efficient code.

- ☐ The mapping of CGiS' SPMD parallelism is not trivial anymore.
- ☐ CPUs contain data caches emphasizing the need for efficient data layouts in memory intensive algorithms.
- ☐ Full control flow conversion becomes indispensable for exploiting SIMD parallelism.

In our work we tackle these obstacles with a combination of well-known and novel techniques. We have developed a new method to map SPMD parallelism to SIMD hardware, called *kernel flattening*. This transformation breaks down compound data into scalars to enable sensible packing of new SIMD vectors for parallel execution. To ensure the preservation of the program semantics, static program analyses are used to guarantee the premises. This also requires reordering of the input data, either locally to the kernel call or globally for all kernels.

For many algorithms, memory access is the bottleneck. Data cannot be fetched fast enough from the main storage to keep the units in the processor busy. To cope with this problem all desktop CPUs implement cache hierarchies with success. Caches are small memory segments on the die of the processor core enabling a fast connection and low latencies. The whole idea of caches is based on the theory that most data is accessed multiple times, thus storing it in a nearby location benefits the performance of

the processor. When it comes to stream programming, one faces the problem that not all the data usually fit into the data cache and thus eviction and re-insertion occurs, inducing severe performance penalties. Many data-parallel algorithms, such as image filters, contain neighboring operations, i. e. accesses to stream elements adjacent to the one currently operating. As the CGiS compiler has a clear view of the data access patterns it can change the iteration sequence over the data to be processed to achieve a better cache performance, a form of *loop sectioning*.

Parallel execution of different input elements gets inconvenient when it comes to diverging control flow. To guarantee the semantics of a program, the control dependencies have to be converted into data dependencies by full *control flow conversion*, especially in the context of kernel flattening. The appropriate replacement of if-then-else constructs and loop control constructs by introducing masking and select operations enable parallelization.

We have shown in [14] the efficiency of CGiS applications on single core desktop processors with the abstraction of multi-media instruction sets. The next group of targets in line are multi-cores, one of which is IBM's Cell Broadband Engine or in short the *Cell* processor. It features a PowerPC core and eight synergistic processing elements (SPEs). Those are pure short-vector SIMD processors with an instruction set very similar to AltiVec. With a unique local storage and DMAs to main memory the SPEs offer different kinds of obstacles for a compiler to overcome. Data alignment and distribution of computation jobs to the different SPEs are crucial to derive significant speedups. The explicit parallelism of CGiS, however, lets the backend handle these issues in an efficient way. The SPEs can work independently and can be accessed via thread programming. The Cell backend of the CGiS compiler now has the task of distributing the computations on the different processing elements. While there are plenty of opportunities to increase the performance of the code running on the SPEs, the backend presented here merely serves as a proof of concept underlining the aptitude of programming multi-core architectures with CGiS.

To emphasize the viability of our approach to exploit SIMD parallelism with CGiS a set of scientific, data-parallel algorithms has been examined. The testing environment comprises two sets of different machines: the Intel Core2Duo E6300 and E8400 as well as the Core i7 920 are representatives for CPUs with Streaming SIMD Extension units. The Freescale PowerPC G4 and a Sony Playstation 3 are used to evaluate the AltiVec backend and the Cell BE backend. The sample applications originate from different application domains. The CGiS version of the algorithm is compared to standard C++ implementations on different input sets regarding execution times for all three new SIMD backends. For most applications a significant speedup is observable due to successful SIMDization. For each test we will explain its SIMD performance to acquire an insight into which algorithms are suited best for SIMD compilation with the CGiS system.

1.2. Outline

The remainder of this thesis is organized as follows. Chapter 2 gives an introduction to stream programming as well as the SIMD hardware for which code is to be generated. The presentation of the target architectures is as detailed as is necessary for the reader to understand their characteristics with respect to generating efficient code. In Chapter 3 data-parallel programming is discussed in the context of SIMD hardware. From close to the hardware up to high-level languages, the possibilities are revealed. For selected languages, a short example will be shown to give a good impression of how SIMD hardware can be programmed. Our work focuses on CGiS as a data-parallel programming language, thus giving an overview of this language and stepping into detail whenever illustrative is done in Chapter 4.

Chapter 5 then presents the CGiS compiler. Its workflow is shown as well as its infrastructure together with the runtime libraries. The compiler has been augmented with an interface for program analyses to allow for easy collecting and evaluating program characteristics for later optimizations. The optimizations and program transformations fabricated for the SIMD backend of the compiler are given here. The viability of this approach is discussed in Chapter 6 by presenting a set of example applications and analyzing their performance. The benefits and the limitations of the SIMD backends are exhibited by classifying the set of profitable applications.

Finally, Chapter 7 concludes this work with a discussion of the viability of the CGiS language for SIMD hardware and gives an outlook on ongoing and future work.

CHAPTER 2

SIMD: Hardware and Programming

This chapter investigates SIMD hardware, its characteristics and restrictions. Section 2.1 gives an introduction to SIMD hardware and stream programming. The history and development of this kind of hardware is briefly described in Section 2.2. Section 2.3 takes a closer look at today's SIMD hardware found in widespread consumer CPUs and their instruction sets, the Streaming SIMD Extensions and AltiVec. The Cell Broadband Engine Architecture poses more challenges to code generation despite a SIMD instruction set and thus is examined in more detail in Section 2.4. To conclude this chapter Section 2.5 points out the restrictions with which a programmer of SIMD hardware has to cope.

2.1. Overview

Our everyday life is determined by machines that gather, receive, display or send data. This can be the cell phone letting you talk to other people by decoding incoming audio data, transforming it into sound and encoding your speech into a digital data stream and sending it through the hemisphere to your communication partner. It can be your PC encrypting your credit card information for your recent online shopping. It can also be the ultrasound probe at your doctor's gathering reflected ultrasound rays to let a computer display your insides. All of these machines got one thing in common, amongst others, they process series of data packages or data streams.

Audio or video encoding or decoding, scrambling of sensitive data, compression of data and transformation of data are stream applications with the commonality that on every data package the same computations are done. This kind of computing is known as *stream computing* or *stream processing*.

Stream processing[57] is also a computer programming paradigm. The system can be seen as a directed graph with sources, filters and sinks as nodes. The edges are data channels. Stream data enter the system via the sources, are processed in filters and leave the system in sinks. In other models, instead of filters the term *kernel* is employed.

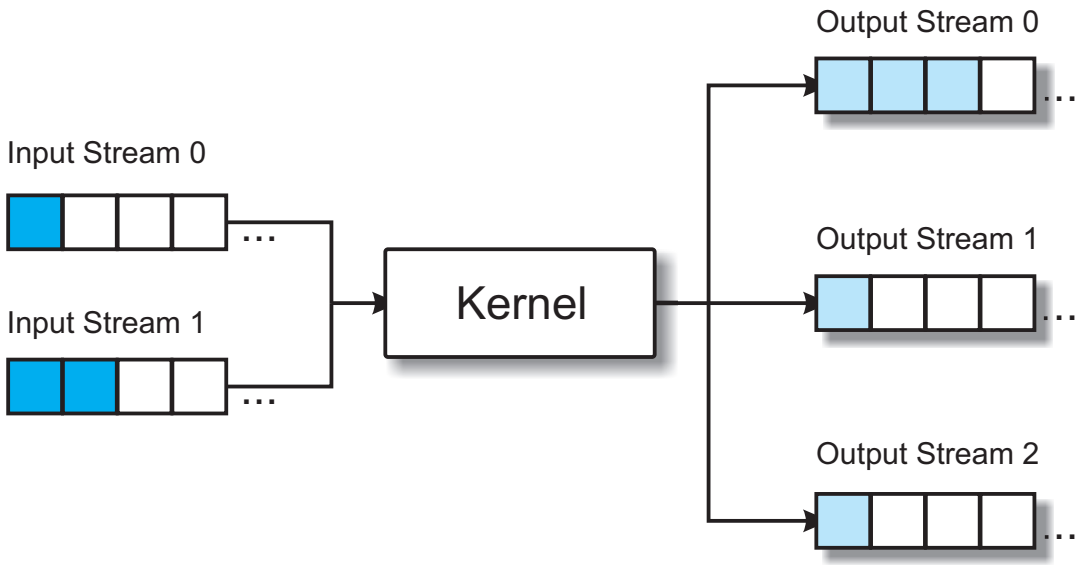


Figure 2.1.: The basic view of stream processing. Several input streams are read and several output streams are computed.

There is no difference in their meaning and they can be used interchangeably. On every element of the stream the same computations are applied which offers the possibility of parallelizing these computations. The basic streaming model consists of a *kernel* and several input and output streams. The kernel processes n_i elements of each input stream to create m_j elements of each output stream, see Figure 2.1 where $n_0 = m_1 = m_2 = 1$, $n_1 = 2$ and $m_0 = 3$.

Applications suitable for stream processing comprise three main properties:

- ☐ High arithmetic density, a lot of computations are performed on the input data, the arithmetic operations in a kernel are compute dominated.
- ☐ Data parallelism, the computation of the current stream elements are independent of the results of the previously processed elements.
- ☐ Data locality, usually data is produced once and read again once or twice and then never touched again.

Related to stream processing but far from being the same is the *SIMD*, Single Instruction – Multiple Data streams, paradigm. It is more closely connected to the underlying hardware, thus more concrete. Computers featuring SIMD instructions contain a vector register set to apply the same operation to all elements of vector register operands. Containing a vector register set SIMD hardware can also be seen as (short-) vector processing hardware. A typical SIMD operation on vectors with four elements is depicted in Figure 2.2. The elements of the two vector operands vA and vB are combined concurrently to produce the output vector vC. SIMD vector operands are of fixed length and the number of elements depends on the architecture and the supported data types.

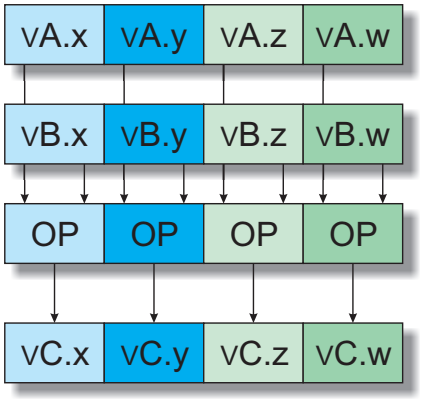


Figure 2.2.: A SIMD operation, elements of vector a and b are combined pairwise.

To get a better understanding what kind of stream computing hardware there is, we take a look at the basic classification of computer architecture done by Flynn in 1966, which distinguishes four different types:

- ☐ Single Instruction, Single Data stream (SISD). A sequential computer, neither instruction nor data parallelism is supported.
- ☐ Single Instruction, Multiple Data streams (SIMD). The same operation can be executed on a multitude of data streams in parallel.
- ☐ Multiple Instruction, Single Data streams (MISD). Multiple instructions are applied on one data stream.
- ☐ Multiple Instruction, Multiple Data streams (MIMD). Multiple autonomous processors operate on multiple data streams, such as distributed systems.

The arrangement of hardware into one of the above four categories is vague when it comes to vector processing hardware. Strictly following this taxonomy, a vector computer with only one arithmetic logic unit (ALU) would belong to SISD hardware, whereas vector processors with more ALUs are considered SIMD hardware[64]. Over the last decades, the differentiation has softened to the point where vector processors and SIMD hardware are considered the same class of hardware.

2.2. History of SIMD Hardware

The ILLIAC IV[6] from the University of Illinois in the 1960s was the first SIMD machine ever built. It was a 64-bit design, rather unusual for that time. The core of the ILLIAC IV consisted of a control unit (CU) responsible for fetching and decoding of instructions and sixty-four processing elements (PE). The CU exhibits sixty-four 64-bit general purpose registers and a PE six special purpose registers. In the original design concept there were supposed to be four CUs with sixty-four PEs each, but because of

time pressure the other three were abandoned resulting in a performance of roughly 200 MFLOPS[1].

Other vector machines which emerged in the 1970s were much more successful, such as the Cray-1[49], developed by Cray Research. The Cray-1 was equipped with a huge register set to keep values on which is repeatedly operated. This was a significant performance improvement compared to supercomputers that only work on memory operands. While the machines were powerful computation-wise, many applications did not offer the kind of parallelism to be exploited or it was too hard to find for compilers. Finding and excluding data-dependencies for transformations to enable vector code generation is still a difficult task. Throughout the 1980s and 1990s more supercomputers were developed but their tasks remained special purpose and they remained unaffordable to the common user. In the meantime, desktop PCs became the focus of the mass market. The IA32 [24] from Intel was probably the most successful processor architecture for the average consumer. The software quickly broadened from pure office and development applications to entertainment. Audio and video processing are typical stream applications and can be parallelized for better performance.

With the arising need of parallel computation power for desktop PCs in the late 1990s, widely used architectures like the x86 architecture or the PowerPC architecture are supplemented with SIMD units. Though the core is still a SISD machine these additional units allow for a notable speed-up in many multi-media applications.

The main force behind needs for parallel computing in the mass market nowadays is graphics. Video games demand a high amount of floating point computations for rendering almost photo-realistic scenes and simulation of correct physics of objects. With their astonishing price-to-computation power ratio of graphic processors, the interest in parallel computing increases every day as high performance parallel computing architectures are now available for everyone.

2.3. Multi-Media Instruction Sets

Multi-media instruction sets have been used in scientific as well as engineering applications, showing decent performance increase[19]. There is a variety of micro processor extensions allowing SIMD execution of different data-types. [56] gives an exhaustive overview of the current instruction set architectures using multi-media extensions.

The two most popular SIMD instruction sets are the Streaming SIMD Extensions found in Intel's IA32 and the AltiVec[2] instruction set for Freescale's PowerPC architecture. While having some features in common such as the width of the vectors, they also differ in others, e. g. the supported operations. A brief overview over the main differences is shown in Table 2.2 on page 14.

[1] 1 FLOPS equals one floating point operation per second.
[2] AltiVec was originally developed by Motorola.

2.3.1. Streaming SIMD Extensions (SSE)

The first SIMD instruction set in commercially successful desktop processors, the *Multimedia Extensions (MMX)* [37], was introduced by Intel in 1997. MMX extends the core instruction architecture with eight 64-bit registers and offers only integer instructions. The Intel Pentium MMX processor was the first processor with MMX support.

Only two years later, MMX was followed by the Streaming SIMD Extensions (SSE) [23] which were added to the Intel Pentium III. The first version of SSE provides a set of instructions operating on packed single-precision floating-point values. The SSE instruction set was successively extended by introducing integer support, horizontal operations and specialized operations (SSE2, SSE3 and SSSE3).

The Streaming SIMD Extensions hardware defines a set of eight named 128-bit wide vector registers, called XMM registers. In addition, there is a parallel set of 64-bit MMX registers that are used by the MMX extension to x86.[3]

There are three major classes of data on the SSE vector unit: integer, single precision floating point and double precision floating point vectors. Each of those may be serviced by separate parts of the processor. Most instructions take two source operands where one is also the destination (two-address instructions). One of the operands can either be register or memory type. This eases the register pressure, as only eight XMM registers are available.

The SSE instruction set combines a variety of different arithmetic instructions from standard addition and multiplication to computing the reciprocal or the square root. Clearly, the operations available depend on the data type of the operands. To convert data types, there are special conversion instructions.

Sometimes it is necessary to access single components of a SIMD vector or re-arrange the contents of a vector. The Streaming SIMD extensions offer three different instructions to implement data reordering as shown in Table 2.1. Each of these instructions takes two vector register operands to create a new one. The unpack instructions combine either the two low-order elements of the two input vectors or the two high-order elements. The shuffle instruction shown in Figure 2.3 is more versatile. A bit mask determines which elements of the target vector are taken from which input vector. The main restriction to this operation is that the two high-order elements of the target vector have to be arbitrarily chosen from the first operand and the low-order elements from the second operand. Depending on the kind of reordering needed, multiple shuffle instructions have to be applied.

SSE2 extensions were introduced in the Pentium 4 and the Intel Xeon processors. The SSE2 instructions operate on packed double-precision floating-point values and on packed integers. They extend IA32 SIMD operations by adding new 128-bit SIMD integer operations and by expanding existing 64-bit SIMD integer operations to 128-bit

[3]The MMX register file aliases the x87 floating point register stack. Use of MMX causes an automatic x87 state save. The x87 unit will not function properly until you issue a EMMS instruction.

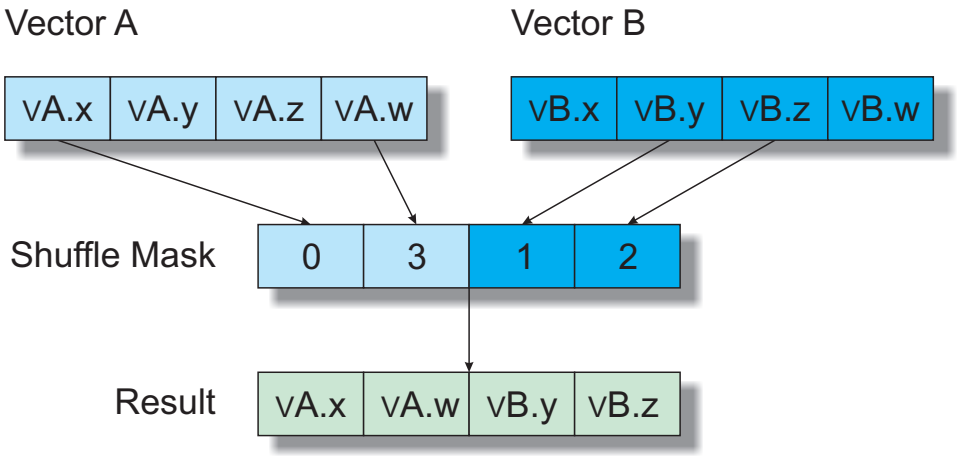


Figure 2.3.: SSE shuffle operation.

Instruction	Intrinsic	Meaning
SHUFPS	_mm_shuffle_ps	Shuffles values in packed single-precision floating-point operands
UNPCKHPS	_mm_unpackhi_ps	Unpacks and interleaves the two high-order values from two single-precision floating-point operands
UNPCKLPS	_mm_unpacklo_ps	Unpacks and interleaves the two low-order values from two single-precision floating-point operands

Table 2.1.: SSE shuffle and unpack instructions

XMM capability. SSE2 instructions also provide new cache control and memory access operations, such as reads and writes of XMM register bypassing the cache. SSE3 extensions were also introduced with the Pentium 4 processor supporting Hyper-Threading Technology. It offers thirteen instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities such as horizontal vector operations. The Intel Xeon processor 5100 series and Intel Core 2 processor families introduced the Supplemental Streaming SIMD Extensions 3 (SSSE3). These include more horizontal addition and subtraction operations as well as multiply-add operations. SSE4 promises a broader connection to the SIMD processing unit as well as more specialized instructions to speed up common algorithms. It is available in Intel processor generations built from 45nm process technology. SSE4 is split into two extension sets. SSE4.1 aims at improving performance of multimedia applications by adding special operations such as a dot product and more data insertion and extraction operations. SSE4.2, on the other hand, is targeted at improving applications in the areas of string processing, application-targeted accelerator (ATA) instructions and 128-bit integer computations.

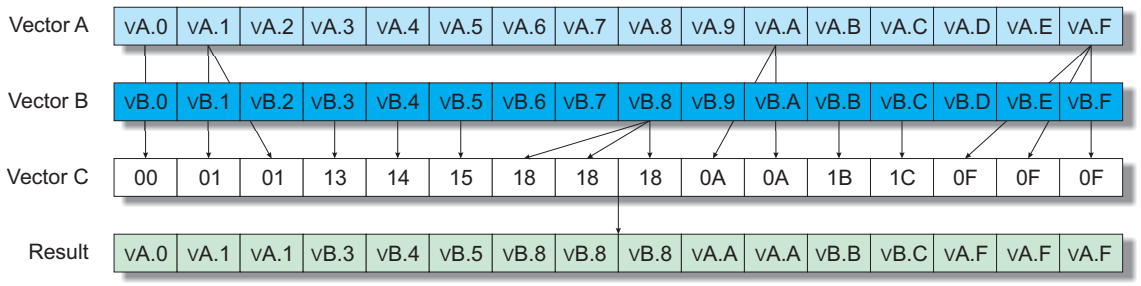


Figure 2.4.: The AltiVec instruction vperm.

2.3.2. AltiVec

The PowerPC architecture was augmented with the Velocity Engine or *AltiVec* [13] in 1999 with the PPC7400. The original chip ran at clock rates from 350MHz to 500MHz. It contains 10.5 million transistors, manufactured using Motorola's 0.20μm process. Most of the design was done by Motorola with the assistance of Apple and IBM. At this point IBM chose not to manufacture the chip as they saw no need for the vector processing unit.

AltiVec technology as an extension to the PowerPC architecture adds a 128-bit vector execution unit. It operates concurrently with the existing integer and floating-point units. Instructions can be issued to different units at the same time. Such a vector execution unit can perform all arithmetic instructions as well as data permutes. Usually two vector units are assembled together, one handling permutation operations and one handling arithmetic ones. All SIMD operations, regardless of the data type, are done in one cycle. AltiVec provides thirty-two 128-bit registers to hold vectors, and supports integers of various widths as well as single precision floating-point data. The data type width for integers supported are 8, 16 and 32 bits, signed and unsigned. In contrast to SSE, AltiVec supports very powerful data reordering or permutation instructions, allowing arbitrary interchange of input vectors.

A typical AltiVec instruction gets up to three input operands and produces one output. Target as well as source operands must be present in registers unlike the SSE operands. All important kinds of instructions are covered for their appropriate data type. AltiVec does not support horizontal operations but compensates for this shortcoming with its vector permutations. The `vperm` instruction depicted in Figure 2.4 lets the programmer arbitrarily reorder bytes inside a vector register.

An AltiVec permute operation gets three input vectors. The first two are the source vectors from which the target vector should be composed, and the third vector contains a series of bytes denoting which byte in the target vector is taken from which source. The fifth bit in a selection byte decides which input vector is chosen and the lower four bits decide which byte from that input vector. For example, the third value in the selection vector C selects the third byte in the target vector `vD` from vector A. So the value `0x01` selects the first byte of vector A for the third byte of the result.

The permute instruction is very useful for implementing horizontal operations and

	AltiVec	**SSE, SSE2, SSE3**
registers	32	8 XMM
maximum throughput	8 Flops/cycle	4 Flops/cycle
32-bit arithmetic	full	no saturation
integer compares	full	no unsigned
instructions per cycle	1	2 for most instructions
IEEE-754 compliance	Java subset	full
number of operands	3-4	2
unaligned loads	-	MOVUPS, MOVDQU

Table 2.2.: Differences between AltiVec and SSE

can also be used to meet the memory alignment requirements of AltiVec hardware. Misaligned data can be loaded aligned, reordered and processed.

The memory accesses of the AltiVec hardware need to be 16-byte aligned. The lower 4 bits of any address to be loaded with a `lvx` or `lvxl` instructions are zeroed out.

2.4. Cell Broadband Engine

The Cell Broadband Engine Architecture (CBEA) is a recent multi-core architecture that extends the 64-bit PowerPC Architecture. The development of the CBEA began in 2001 with three heavy-weight partners in microprocessor and computer entertainment industry: Sony Entertainment Inc. (SCEI), IBM and Toshiba. The Cell Broadband Engine (CBE) processor is the first implementation of a multiprocessor family conforming to the CBEA.

The computer entertainment market is growing every day and the demands for multimedia hardware are increasing. Apart from decoding or encoding audio and video data, video games challenge today's systems requesting high performance for graphics, physics calculation and communication.

The CBE is designed to support a great variety of applications. The key design decision hereby is the single-chip multiprocessor. It features nine processor elements operating on a coherent memory. One PowerPC Processor Element (PPE) works together with eight Synergistic Processor Elements (SPE). The PPE is a general purpose 64-bit PowerPC architecture thought to run the operating system and handle task switching. The SPEs are specialized to compute SIMD-only operations. They are designed to be programmed in C or C++, so SIMD operations are preferred but not mandatory. Each SPE is able to run its own application individually. Both PPE and SPE are capable of computing and task switching though it is obvious which processor element is designated to which duty.

Memory access varies for the different processor elements. The PPE has common load/store instructions to exchange data between its register file and the main memory. All accesses are cached as the PPE contains a discrete L1 instruction and data cache and a common L2 cache. Contrary to this the SPEs are equipped with a private uncached

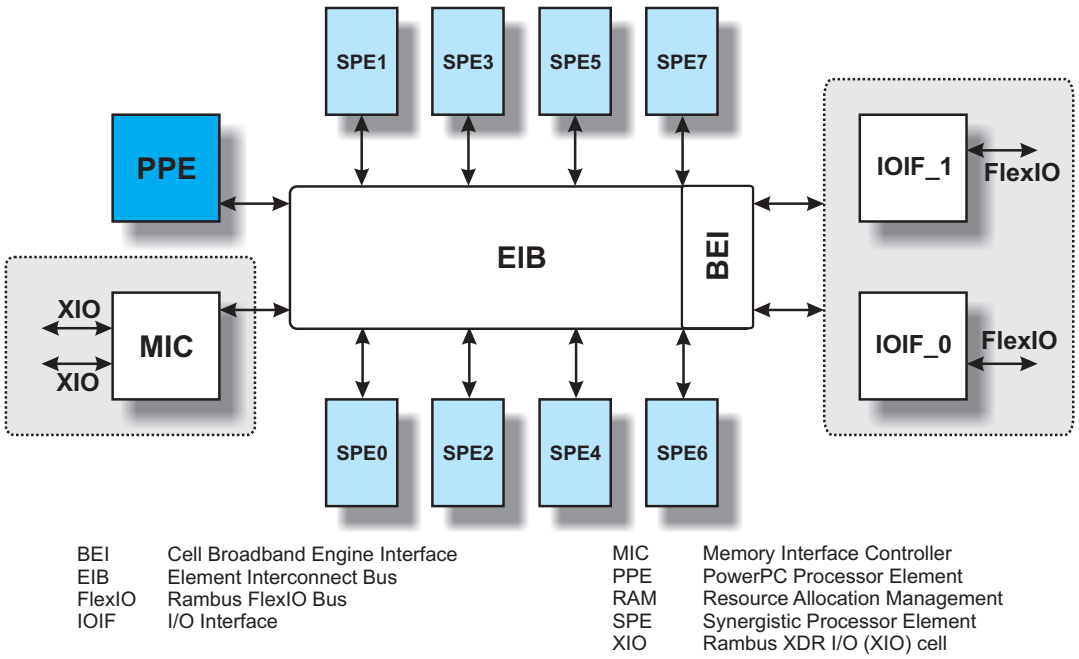


Figure 2.5.: Core of the Cell BE.

Local Storage (LS). Accesses to main memory must be scheduled via asynchronous Direct Memory Access (DMA) transfers. This enables hiding of high memory latencies which pose problems in other processors. The DMAs only take a few cycles to set up and the system supports up to twelve pending DMAs.

The CBE gains its advantage over other processors by specialization of its processor elements. The PPE is very well suited to control-intensive tasks and obtains its efficiency by creating and managing threads and not single-thread performance. On the other hand, the SPEs excel at compute-intensive tasks. They feature a large register file and support many in-flight instructions. Their behavior is predictable as all local storage accesses are uncached. There is no speculation, out-of-order execution or register renaming necessary.

Figure 2.5 shows a general overview of the CBE. The PPE, the SPEs, the main memory and the I/O devices are connected via an Element Interconnection Bus (EIB). It consists of four 16-byte wide data rings pushing data stepwise toward its destination.

2.4.1. PowerPC Processor Element

The PowerPC processor element is a general-purpose RISC processor. It is dual-threaded and supports a 64-bit address space. The core comprises a PowerPC architecture, version 2.02, including AltiVec units. The PPE consists of two parts: the PowerPC Processing Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS).

The PPU features two distinct L1 caches for data and instructions and it includes five

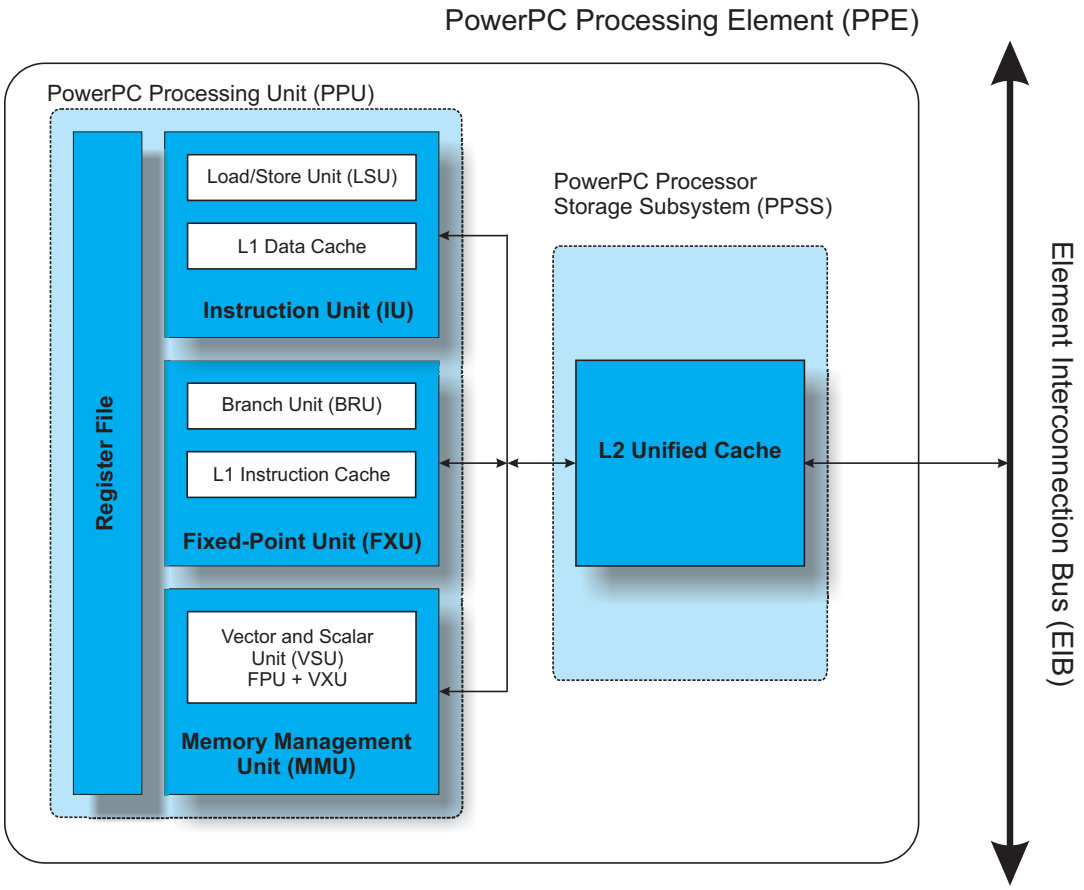


Figure 2.6.: The PowerPC Processor Element.

execution units:

- An instruction unit (IU) that handles fetching, decoding, dispatching, issuing, branching and completion of instructions. Part of it is the L1 instruction cache.
- A load/store unit (LSU) which performs data accesses and execution of loads and stores. It includes the L1 data cache.
- A vector scalar unit (VSU) that holds a floating-point unit (FPU) and a vector/SIMD multimedia extension unit (VXU).
- A fixed-point unit (FXU) for integer operations.
- A memory management unit (MMU) manages address translation.

A coarse grain overview of the PPE system can be seen in Figure 2.6. It shows the data-paths of the different functional units within the PPU and the L2 cache.

The register file of the PPE includes 32 general purpose registers, 32 floating-point registers and 32 vector registers. The vector registers are 128-bit wide; the others are

64-bit wide. The L1 caches are sized 32KB each, 4-way-set-associative, support write-through and have a line-size of 128 bytes. The replacement policy for the L1 instruction cache is least recently used (LRU), for the L1 data cache and for the L2 cache it is pseudo-LRU.

The PPSS takes care of all memory and I/O accesses issued by the PPU and is connected to the element interconnection bus. It has a unified L2 cache which is guaranteed to hold the content of the L1 data cache but not the L1 instruction cache. The L2 cache is a unified, 8-way set-associative cache of 512KB. The cache-line size is the same as for the L1 caches: 128 bytes.

The instruction set of the PPE is a superset of the PowerPC instruction set including AltiVec. It is described in the previous section.

2.4.2. Synergistic Processor Element

The synergistic processor elements are 128-bit RISC processors. They are specialized for data-rich and compute-intensive SIMD applications. A SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC).

Synergistic Processor Unit

The SPU is subdivided into three parts: the local storage (LS), the Synergistic Execution Unit (SXU) and the SPU Register File (SRF). Since the SPU has no direct access to the main memory it has its own local storage. Data and instructions used by the SPU are held here. Instruction prefetches are 128 bytes per cycle and the data access bandwidth is 16 bytes per cycle. The local storage holds up to 256KB of data and is not cached, which results in a locally deterministic execution behavior. An access of the local storage from its SPU is said to complete in one cycle.

An SXU contains six different execution units in two different execution pipelines, the *even* and the *odd* pipeline. The even pipeline is connected to the SPU Even Fixed-Point Unit (SFS) and the SPU Floating-Point Unit (SFP). The odd pipeline is connected to the SPU Odd Fixed-Point Unit (SFS), the SPU Load and Store Unit (SLS), the SPU Control Unit (SCN) and the SPU Channel and DMA Unit (SSC). Figure 2.7 shows a synergistic processor element with LS, MFC and SXU and its connection to the element interconnection bus.

The SFS executes shifts, rotates and shuffles, while the SFX can execute arithmetic and logical instructions, shifts, rotates, floating-point compares, floating-point reciprocal and reciprocal square-root estimates. All other floating-point instructions are handled in the SFP. Loads and stores as well as DMA requests to the LS are handled by the SLS. The SCN is used to fetch and issue instructions to the even and the odd pipelines. It also executes branch instructions, arbitrates access to the local storage and performs some other control functions. The SSC is responsible for communication, data transfer and control into and out of the SPU.

Having two independent execution paths, it is possible to execute two instruction per cycle (dual-issue). The names of the two pipelines, even and odd, result from the

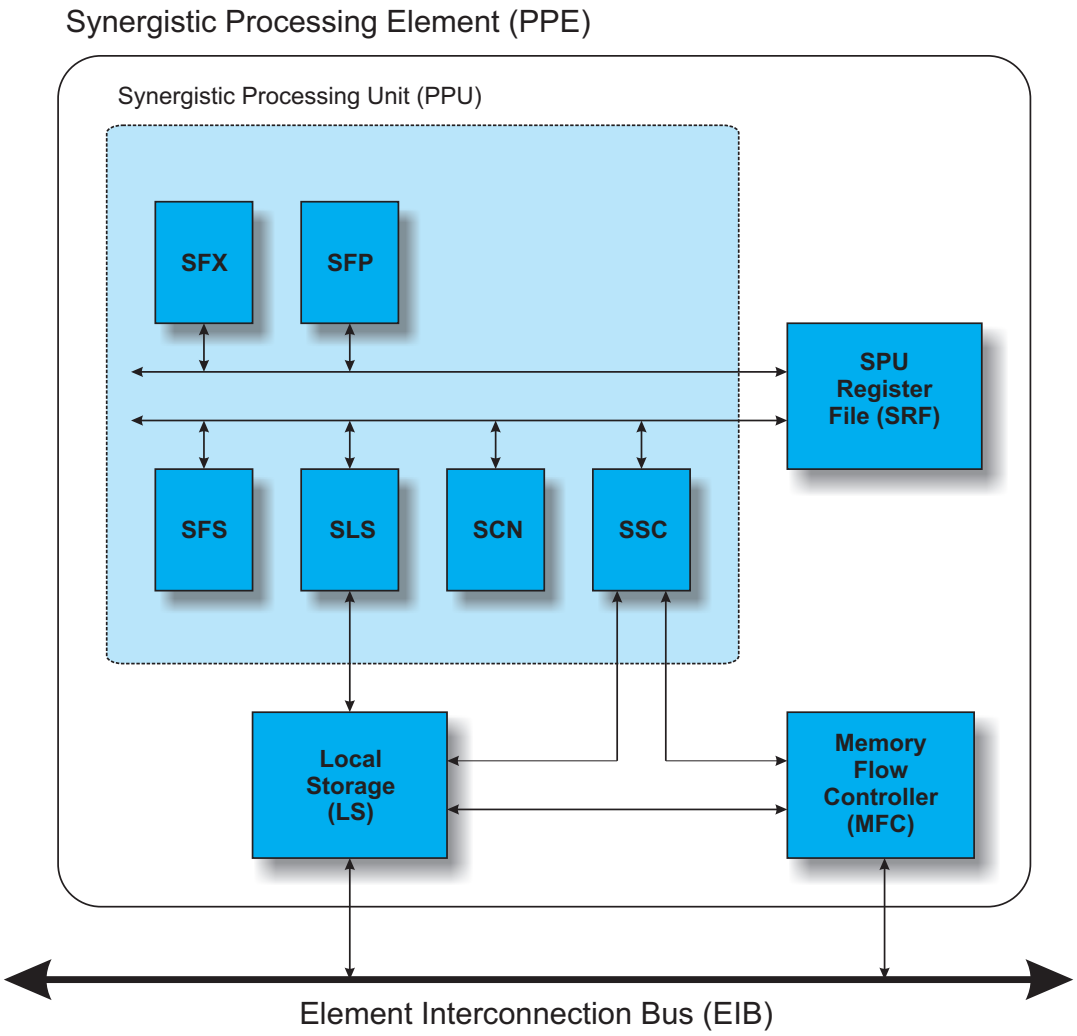


Figure 2.7.: The Synergistic Processor Element.

addresses the pipelines expect their instruction to be in, in the local storage. So the even pipeline expects for example its instruction at an even word address. A dual-issue can only take place if two consecutive instructions match the address and execution unit requirements. Should an instruction be placed at the wrong address regarding its required execution unit, a small penalty for switching re-assigning occurs. The latency of the instructions depends on their type and the unit occupied.

The register file of the SPU consists of 128 general purpose registers of which each is 128-bits wide, suitable for SIMD and scalar operations. The data types supported comply with the ones from the AltiVec instruction set.

Memory Flow Controller

The MFC contains a Synergistic Memory Management (SMM) and a Direct Memory Access Controller (DMAC). Its primary function is to interface its LS-domain with the

main memory domain moving data and instructions back and forth. Communication between other SPEs or the PPE works via DMA or mailboxes.

The SPE's channel interface supports two mailboxes for sending messages from the SPE to the PPE; one mailbox is provided for sending messages from the PPE to the SPE. The channel interface also features two signal-notification channels for inbound messages to the SPE.

Applications running on the SPE, the PPE or on devices connected to the EIB, can initiate DMA transfers by using MFC commands. There are two separate command queues maintained by the MFC, one for the attached SPU and one for commands received from the PPE. Out-of-order execution of DMA commands is possible. By grouping DMA commands and using tags, one can wait for certain DMAs to complete in a group. Also group-tagged commands can be synchronized with a fence or a barrier option, enforcing the completion of pending DMAs before subsequent DMAs can be issued. The MFC autonomously executes DMA commands allowing parallel execution of SPU instructions.

DMA transfers between the LS and main storage are coherent in the system. A pointer to a data structure created on the PPE can be passed by the PPE, through the main-storage space, to an SPU, and the SPU can use this pointer to issue a DMA command to bring the data structure into its LS.

2.4.3. Element Interconnection Bus

The communication bus internal to the Cell processor which connects the various on-chip system elements is called *Element Interconnection Bus* (EIB). It provides means for its participants to exchange data. The twelve participants communicating over the EIB are the PPE, the memory controller (MIC), the eight SPEs, and two off-chip I/O interfaces. The EIB supports full memory-coherent and symmetric multiprocessor (SMP) operations, meaning the Cell BE is designed to be grouped with other Cell BE processors to produce a cluster. The implementation of the EIB consists of four 16-byte-wide data rings or channels. Three transactions can be carried out simultaneously per channel. Data is propagated stepwise across the rings; two rings propagate data clockwise the others counterclockwise. This stepwise transferring of the data results in different latencies for different participants depending on their communication partner's position in the ring. Data is always sent the shortest way so that the longest distance it possibly has to travel is six hops. While the performance impact should not be significant as the transmission steps are very fast, longer communication distances can reduce available concurrency.

One ring is able to transfer 16 bytes per two processor clock cycles. Sixteen bytes exactly comprise one cache line of the PPE. Each PPE and SPE features one on-ramp and one off-ramp. Thus, data can be received and sent at the same time. With four rings and three active transactions per ring this leads to a theoretical maximum throughput of 96 bytes per cycle.

With a clock frequency of 3.2GHz one channel is able to transmit 25.6GB/s. Running twelve simultaneous data transfers, this throughput leads to a peak performance of 307.2GB/s. This is however only a theoretical number. Typically most transfers origi-

nate from or are destined to the memory. The memory interface controller offers only a combined read and write throughput of 25.6GB/s, limiting the calculational maximum by a factor of 12.

2.4.4. Memory Controller and I/O

The Cell processor has a Rambus XIO macro to interface with the Rambus XDR DRAM (eXtreme Data Rate Dynamic Random Access Memory). The memory controller itself is separated from the macro. The link between this XIO and the XDR features two 32-bit channels which provide transfer rates of up to 25.6GB/s.

Rambus also designed the interface to I/O devices called FlexIO. This system consists of twelve different uni-directional, 8-bit wide lanes. Five of those are inbound to the Cell system and the remaining seven are outbound. This results in a maximum bandwidth to external peripherals of 36.4GB/s outgoing and 26GB/s incoming.

2.5. Characteristics of SIMD Programming

While the SIMD hardware presented here offers a decent amount of parallel computation power, it also burdens the programmer with applicability restrictions. Data to be loaded into SIMD registers usually need to match a certain memory alignment which either forces a preprocessing step to re-arrange data or some sort of on-the-fly reordering or shuffling. Also, SIMD instruction sets can be limited regarding the variety of the available operations, obliging the programmer to store intermediate results in memory and simulate the absent operation with scalar operations. The instruction sets contain a few special operations on floating-point values which do not comply with the IEEE standard causing possible precision problems.

2.5.1. Data Alignment

The biggest issue when it comes to *SIMDization* – the transformation of a program into a SIMD program – is memory alignment. A memory address A is *n-aligned* if $A \bmod n = 0$ with n being the width of the data in bytes. If $A \bmod n$ is not equal to zero the value determines the offset from the alignment.

All SIMD instruction sets introduced here require the data to be 16-byte aligned. 128 bits, or 16 bytes, is the common size for a SIMD register. Only data matching this requirement can be loaded into a vector register. SSE2 introduced an instruction to do misaligned loads. It eases the access of misaligned data but also has a severe impact on the performance of the application discouraging its use. The impact of misalignment on applications using multi-media extensions is extensively discussed in [53].

One can distinguish between two different problems when it comes to the alignment of stream data. First, is the whole stream aligned, meaning “Does the first element fit the alignment?” and secondly “Is each element aligned?”.

The first problem is only a special case of the second but fulfilling this constraint suffices for some problems to be solved. There are two easy ways to achieve the alignment of the first element by either allocating the memory of the stream data accordingly, e. g. using an `aligned_malloc`, or executing a certain amount of loop iterations until the alignment requirement is matched. This is called *loop peeling*.

The second problem, alignment of every stream element, is not as easily acquired. There are three techniques to avoid misalignment.

- **Replication:** data that needs to be loaded into vector registers is held in memory multiple times, each time with a different alignment offset. This has been shown to be profitable for small constant tables that are often reused in parallelized algorithms. Depending on the addressing supported by the target architecture and the size of the data-types used, up to $n - 1$ copies have to be made.
- **Padding:** stream data is copied in-memory before the computation starts and each stream element i gets m_i dummy bytes added so that $(A_i + m) \bmod n = 0$. This technique is time and space consuming, and thus only profitable if the stream is processed several times in the same manner.
- **Shuffling:** here the data is loaded from the closest aligned address, lower than the wanted one. Then the filled vector registers are reordered to reconstruct the stream element as needed. This induces possibly an additional aligned load if the stream element is split across the boundaries of the SIMD load.

As an example for alignment issues consider the following simple image processing algorithm called *masked alpha blending*. A semi-lucent image L is blended over an opaque background B according to a mask of alpha values. Each point of the semi-lucent picture can have a different blending value. For image data represented in red, green and blue values, the masked blending operation looks as follows:

$$\begin{aligned} \forall j \in 0..N-1 : I_j &= (1 - \alpha_j) \cdot B_j + \alpha_j \cdot L_j \\ \text{with } \alpha_j \in [0..1] &\text{ and } I_j, B_j, L_j \in [0..1]^3 \end{aligned}$$

for a picture with N pixels.

Program 2.1 holds an implementation of the alpha blending function in C. The image data is given as a stream of pixels, each of which is a struct containing three floating-point values. The alpha mask is a stream of single floating-point values.

The computations on the different color values could very well be executed in parallel. Therefore however, each element of the streams must be loaded into a SIMD vector. Assuming the target is one of the SIMD units presented above, the loads would have to be 16-byte aligned and a floating-point value would require 4 bytes of memory. Figure 2.8 depicts one of the RGB streams and the three possibilities to use SIMD loads to transfer the data into a SIMD register. The first method called replication would require the data

```
struct pixel { float r; float g; float b; };

void masked_blend (pixel *result, pixel *bg,
                   pixel *luc,    float alpha)
{
  float inv = 1 - alpha;
  result->r = inv * bg->r + alpha * luc->r;
  result->g = inv * bg->g + alpha * luc->g;
  result->b = inv * bg->b + alpha * luc->b;
}
```

Program 2.1: Alpha blending in C++.

stream to be copied three times, each with another 4 byte offset. The second method is to do an in-memory copy of the data aligning the first element to match the requirement and then inserting a dummy floating-point value after each pixel data. A padded stream element now consists of four floats which equal 16 bytes and the alignment conditions are matched. The third possibility is to do up to 2 SIMD loads and then use a shuffle, or permutation operation, to reconstruct the wanted RGB value in a SIMD register. For the storing of misaligned data, the same problems arise. Especially when using shuffling one has to consider that the storing also overwrites surrounding data falling into the 16-byte store operation.

Padding as well as replication consume time and space which can make the resulting performance of the SIMDized algorithm not worthwhile. Many articles on algorithms implemented with SIMD hardware do not take this time and space loss for memory copies into account. For example, the FIR filter in [54] requires the data to be present in memory four times, each time with another 4-byte shift, so that every element in the stream can be loaded with an aligned SIMD load.

2.5.2. Available Operations

Usually, SIMD instruction sets feature the common arithmetic instructions used to implement scientific data-parallel algorithms. Problems arise when special operations are required such as the computation of a sinus function or bitwise rotation.

Many cryptographic algorithms do bit operations on message blocks when encrypting them. A typical bit operation used in such algorithms is a bit-rotation, meaning the bits in the block are shifted by a certain amount and the ones pushed out of the block are filled in at the beginning. Let us assume a message block is 32-bits wide and we want to process four of those blocks in parallel with SIMD arithmetic. In C, the scalar implementation of a bitwise rotation would look like this:

```
#define ROTL(x,y) (((x)<<(y&(32-1))) | ((x)>>(32-(y&(32-1)))))
#define ROTR(x,y) (((x)>>(y&(32-1))) | ((x)<<(32-(y&(32-1)))))
```

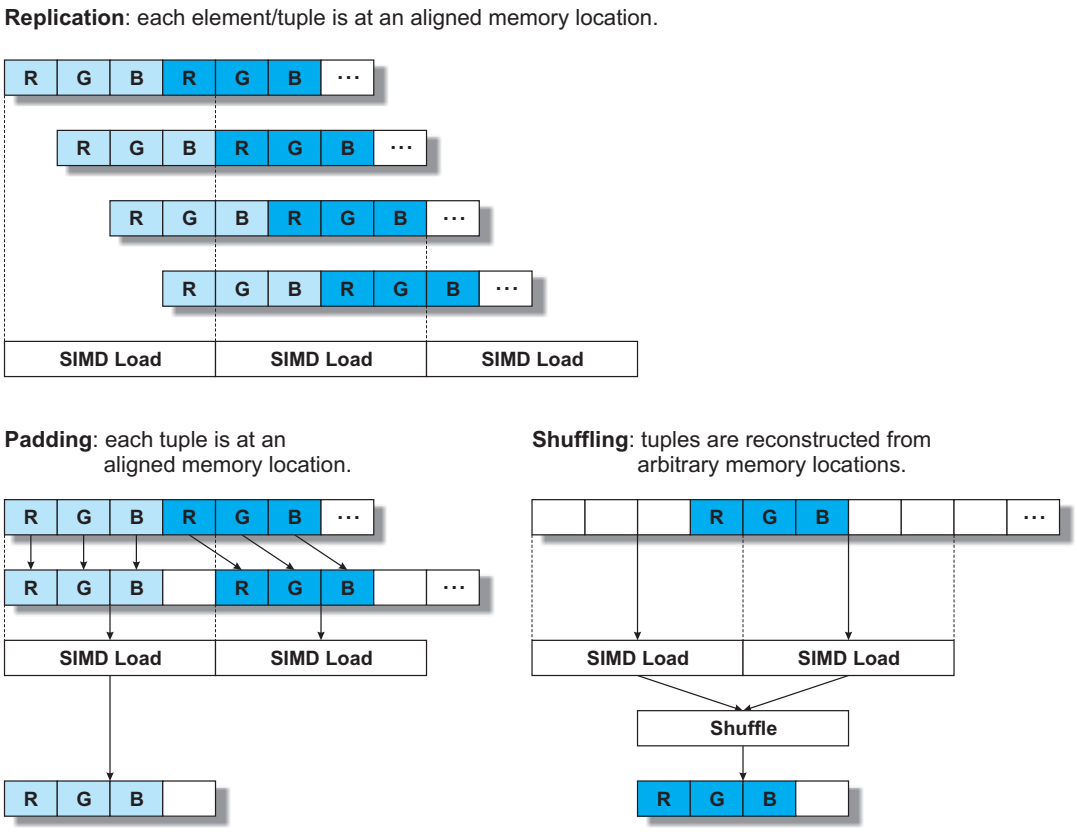


Figure 2.8.: Tackling of SIMD alignment issues.

Though almost impossible to recognize by a PowerPC compiler, a programmer could use the built-in rotation operation; for IA32 contrarily the rotation would be realized by using shifts, binary ands and binary ors. The same fact maps to the SIMD versions. The AltiVec instruction set contains a vector rotation operation called `vrlb` or its intrinsic `vec_rl`. The Streaming SIMD extensions, on the other hand, do not have a rotation operation and even the shifts are only present in a uniform variant. The SSE2 instruction `PSLLD` allows logical left shifts of integers; widths of 16, 32, 64 and 128 bits are supported. Elements cannot be shifted individually inside a vector but are all shifted by the same amount. Thus, a rotation cannot be implemented with SSE instructions alone requiring scalar rotation.

For most mathematical functions there is a possibility to compute an approximation. For example the exponential function e^x can be approximated with the Taylor series. If this approximation can also be implemented with the desired SIMD instruction set there will be no performance loss. Should the instruction set not offer the operations needed to compute the approximation, the vector has to be copied back to memory and has to be computed element-wise on each scalar. The results then need to be packed into an SIMD vector again for the algorithm to continue.

The described setting where data needs to be copied back for scalar execution mostly negates all speed-up gains from SIMDization and might even add up to a slower parallel

execution than standard scalar.

For the Streaming SIMD Extension there is also a lack of functionality regarding integer multiplication. The operations included in SSE2 only operate on 16bit integer values and are not as easy to use as their AltiVec counterparts. For full 32-bit multiplication of two integer vectors, conversion to floating-point vectors is needed. The SSE floating-point instruction `MULPS` can then be used to compute the values that need to be converted back again.

2.5.3. Precision

The Streaming SIMD extensions as well as AltiVec are IEEE floating-point compliant. Some included operations do not support full single precision.

The integrated reciprocal square root approximation and reciprocal approximation are mainly targeted at 3D algorithms. Both operations are needed to compute a normalized vector, for instance. AltiVec features the instructions `vrsqrtefp` and `vrefp` which both return a 12 bit mantissa. Full IEEE instructions are required to hold 24 bits of mantissa. The same holds for the SSE instructions `rcp` and `rsqrt`.

Full precision is provided for the computation of the square root and division but only for SSE. AltiVec, on the other hand, offers approximations for log_2 and exp_2.

CHAPTER 3

Programming Languages for SIMD Hardware

This chapter handles the programming languages that can be used to program the SIMD hardware presented in the previous chapter. Not all of those languages presented come with a compiler able to generate SIMD code. The given selection results from the popularity, applicability and closeness to the hardware.

SIMD hardware can be programmed in multiple ways. First, let us determine a suitable programming model. A *programming model* establishes the hardware abstraction, i. e. how a programmer sees the hardware. As the hardware operates on short vectors in parallel, a data-parallel programming model must be the basis. The stream programming model is applicable if the host processor maps the kernels or filters to the SIMD unit(s). Distributing the data is still driven by the host, only computations are done on the SIMD hardware. In specialized programming languages the programming model is reflected in the syntax and semantics of the language.

In practice there are a few ways of exploiting SIMD hardware:

- ☐ relying on a compiler optimization to identify opportunities for parallelization in standard C/C++[1] code,
- ☐ direct programming via assembly instructions or intrinsics[2] embedded in C or C++ code or
- ☐ using a data-parallel programming language tailored for the task.

Exploiting SIMD parallelism from standard C code is a complicated task. Common C compilers like gcc or icc are facing a multitude of problems both in analyzing the input code and in mapping it efficiently to the restricted SIMD hardware [62]. Specialized operations such as saturated operators or bit rotation operation are common to multimedia applications. These operations have to be reconstructed from C code by idiom recognition [46, 47].

[1] Other sequential programming languages like Fortran are to be ignored here as they have a smaller user base.

[2] Compiler integrated functions directly mapping to instructions or instruction sequences.

To utilize SIMD potential on scalar code, superword level parallelism is able to recognize isomorphic operations on sequential, scalar code [30, 58]. One of the major problems is data alignment, because SIMD hardware is usually limited to accessing 16-byte aligned addresses [45].

The use of intrinsics is a comfortable way of accessing the hardware but still requires expert knowledge of the target's instruction set and restrictions. No form of abstraction is given, alienating many programmers from using it. Still, many algorithms are implemented by hand in assembly code or intrinsics, or using prefabricated libraries [46].

The languages presented in this chapter have one key feature in common: *explicit parallelism*. Being able to express the concurrent execution of different statements by the programmer takes the burden of finding parallelism from the compiler. That again lets the compiler make best use of the available hardware resources for parallel execution.

One basic distinction between those data-parallel programming languages is *autonomy*. Some of the languages presented in the following section are merely an addition to an existing programming language that enables the use of parallelism. Others are encapsulated and form a separate part of the whole application.

First direct programming by inline assembly and compiler intrinsics will be discussed in Section 3.1. Then we take a look at StreamIT in Section 3.2, a well-known stream programming language, followed by OpenCL in Section 3.3, a new standard to unify data-parallel programming across a multitude of platforms. OpenMP has been used for over a decade now by parallel programmers and is introduced thereafter in Section 3.4. Section 3.5 gives brief summaries of other interesting data-parallel programming languages. Finally Section 3.6 discusses the presented languages in the context of scientific data-parallel programming.

3.1. Programming Assembler and Intrinsics

In today's commonly used compilers like the Gnu C Compiler gcc or Microsoft's Visual C++ compiler msvc, very few optimizations using the multi-media instruction sets are integrated. The rare automated usage of the vector instructions stems from the difficult problem of excluding data dependencies. For standard compilers this usually means that only very simple loops with statically known loop bounds can be SIMDized.

A programmer aware of the data layout has two possibilities to access the SIMD hardware: assembly coding or intrinsics. Assembly coding is probably the most effective way of programming SIMD hardware but requires expert knowledge not only of the SIMD hardware but of the whole processor. Implementing loops and other control structures can become tedious work. One has to keep track of available registers, exceptions, restrictions etc. Technical issues aside, writing assembly code is complicated and time consuming and the resulting program might not be understandable by anyone but the author.

Programming SSE or AltiVec using intrinsics is a fairly pleasant way compared to assembly coding. An intrinsic is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsic functions are

inherently more efficient than called functions because no calling linkage is required. Thus, each SIMD instruction is encapsulated with its own C-function and the operands have SIMD vector register data types. The disadvantage is the loss of control over hardware resources. Register allocation, for example, is left to the compiler. On the other hand one can be certain that the code generated from those compilers is close to optimal.

Program 3.1 shows the computation of a cross product of two floating-point vectors with 3 components with SSE. The example code is given as inline assembly in AT&T syntax, e. g. as `gcc` requests it. The last operand is target as well as source operand. Numbers following a percentage sign refer to the operands for the assembly sequence which are given at the end of the `asm` separated by colons. The shuffles swap the components in the right places so that they can be multiplied and finally subtracted.

```
__attribute__((aligned(16)))
  float a[3], b[3], c[3];
// a, b and c are 16-byte aligned
// ...initialize a, b

asm ("movaps %1,     %%xmm0\n\t"
     "movaps %2,     %%xmm1\n\t"
     "movaps %%xmm0, %%xmm2\n\t"
     "movaps %%xmm1, %%xmm3\n\t"

     "shufps $0xc9,  %%xmm0, %%xmm0\n\t"
     "shufps $0xd2,  %%xmm1, %%xmm1\n\t"
     "mulps  %%xmm1, %%xmm0\n\t"

     "shufps $0xd2,  %%xmm2, %%xmm2\n\t"
     "shufps $0xc9,  %%xmm3, %%xmm3\n\t"
     "mulps  %%xmm3, %%xmm2\n\t"

     "subps  %%xmm2, %%xmm0\n\t"

     "movaps %%xmm0, %0\n\t"
     : "=m" (*c)
     : "m" (*a), "m" (*b)
     : "xmm0", "xmm1", "xmm2", "xmm3");

// c contains now the normal...
```

Program 3.1: Cross product using SSE instructions in AT&T syntax

Intrinsics allow for easier usage of processor-specific enhancements because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages resources that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

The computation of the cross product with intrinsics can be found in Program 3.2. The intrinsic version offers great readability and lets the programmer focus on the important part, namely vectorizing his code.

```
#include <xmmintrin.h>
#define SHUFFLE(x,y,z,w) (((w)<<6) | ((z)<<4) | ((y)<<2) | (x))
__attribute__((aligned(16)))
  float a[3], b[3], c[3];
// a, b and c are 16-byte aligned
// ...initialize a, b

__m128 x, y, tmp, res;
x =  _mm_load_ps (a);
y =  _mm_load_ps (b);

tmp = _mm_mul_ps (_mm_shuffle_ps (x, x, SHUFFLE(1,2,0,3)),
                  _mm_shuffle_ps (y, y, SHUFFLE(2,0,1,3)));
res = _mm_mul_ps (_mm_shuffle_ps (y, y, SHUFFLE(1,2,0,3)),
                  _mm_shuffle_ps (x, x, SHUFFLE(2,0,1,3)));
res = _mm_sub_ps (tmp, res);

_mm_store_ps ((float*) c, res);

// c contains now the normal...
```

Program 3.2: Cross product with SSE compiler intrinsics

3.2. StreamIT

StreamIT[59] was developed at the Massachusetts Institute of Technology and first presented in 2002. It was designed with signal-processing algorithms in mind and abstracts the program as a stream graph, consisting of nodes or blocks and connections between them. The atomic block is called a *filter*. Filters have a single input and a single output. The body of filters, the actual code, is written in a Java-like language. The connection of filters works by putting them into composite blocks. There are three such composite blocks: pipelines, split-joins and feed-back loops.

The applications for which the languages aim to be suitable exhibit the following properties:

- ☐ The application operates on large data streams, nearly no reuse of data.
- ☐ Filters are independent and self-contained.
- ☐ During the operation of a stream application the stream graph - the sequence of filters - is generally constant.

- ☐ Small modifications to the stream graph are possible.
- ☐ The filters are allowed to have occasional out-of-stream communication.
- ☐ Efficient results are expected from the compiled stream application.

To get an impression of how a simple StreamIT program looks, let us examine Program 3.3. The reading and writing of input and output are only done in separate filters to show the pipelining construct `Convert`. The two filters `Read` and `Write` get a RGB value or write a float value back respectively. The consumption of a stream element is denoted by the keyword *pop* and the creation of an element by *push*. The computations are done in the `RGB2Grey` filter. It consumes one RGB stream element per tick and pushes one float value.

```
// Data declaration
struct RGB { float R, G, B; }

// The pipeline block
void -> void pipeline Convert {
  add Read;
  add RGB2Grey;
  add Write;
}

// Read/Write values
void -> RGB filter Read {
  work push 1 { push (get_rgb()); }
}
float -> void filter Write {
  work pop 1 { put_float(); }
}

// Conversion filter
RBG -> float filter RGB2Grey {
  work pop 1 push 1 {
    RGB input = pop();
    float grey = 0.3*RGB.R + 0.59*RGB.G + 0.11*RGB.B;
    push (grey);
  }
}
```

Program 3.3: Simple RGB to grey conversion in StreamIT.

The expression of stream computations by filters and their connection is quite intuitive but with a noticeable focus on DSP algorithms. The composite filters together with a message-passing system, e. g. for handling external events, offer a high expressiveness

for the target application domain. The concepts of two dimensional streams, as well as neighboring operations are not supported. The built-in *peek* operation lets the user access upcoming stream elements without removing them from the stream like pop. However, already passed elements cannot be accessed again.

The StreamIT compiler contains stream-specific analyses and optimizations[29] extraneous to standard compilers for imperative languages; this embraces, for example, the combination of adjacent nodes in the stream graph or shifting computations from the time into the frequency domain.

For a single core, the default backend of the compiler generates C++ files which can be linked to the runtime library. When compiling for multicore architectures or clusters, the compiler also generates different threads for the different cores or processors. SIMD parallelism is not exploited by the StreamIT compiler itself; the C++ compiler used to compile and link the generated files, however, might use simple SIMD parallelization optimizations.

3.3. OpenCL

The Khronos Group released the first specification of *OpenCL* (Open Computing Language) [18] in December 2008. Khronos is a conglomerate of many heavy-weight industrial partners such as AMD, Apple, Intel, IBM and NVidia. The idea behind OpenCL is to provide the first open, royalty-free standard for general-purpose parallel programming of heterogeneous systems. Those systems can be any combination of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs.

OpenCL is designed to support all kinds of applications ranging from high performance computing to embedded and consumer software products, by giving an abstraction that is said to be low-level, high-performance and portable. It has a close-to-the-metal interface enabling the development of platform-independent tools, middleware and applications.

The standard consists of a platform-independent programming language and an API for coordinating the computations across a heterogeneous system. From the specification, the following characteristics can be derived:

- ☐ Support of data and task-parallel programming models.
- ☐ The programming language is a subset of ISO C99 with extensions for parallelism.
- ☐ Definition of consistent numerical requirements based on IEEE 754.
- ☐ Introduction of configuration profiles for embedded devices and handhelds.
- ☐ Efficient cooperation with OpenGL and other graphics APIs.

OpenCL exhibits a clear distinction of four different models: the platform model, the execution model, the memory model and the programming model. The platform model describes the view of the hardware components of the system. It is subdivided into host and compute device(s). The latter then contains compute units which consist of

processing elements. The execution model defines how kernels work. Instances of kernels are called work-items which can be organized in work-groups. Those groups can then be arranged in a grid. The memory model distinguishes four kinds of memory: global, constant, local and private. While global and constant memory are self-explanatory, local memory belongs to a work-group and private memory can only be accessed by a work-item. A data-parallel, a task-parallel as well as a hybrid of those is supported by OpenCL. The task-parallel programming model is equal to an execution model where each work-group only contains one work-item.

The programming language in OpenCL is called OpenCL C and is a subset of C99 with extensions to express parallelism. Short vector data types are supported. All data types are aligned to their respective size in memory. Thus, a float4 vector type is automatically 16-byte aligned. Swizzling, the access and permutation of 4-vector operands, as well as scalar expansion, for example adding a scalar to a vector type which is interpreted as adding the scalar to each component of the vector, are also supported. Memory areas can be declared with a certain location, implementing the memory model mentioned above. The runtime system features a great amount of support function to steer the data flow and handle communication and distribution, etc.

The standard provides the kernel shown in Program 3.4 as a minimal example. It computes the dot product, done here by a predefined function `dot`, of two float4 vectors. The ID provided by the function `get_global_id()` refers to the work-item with which the memory locations of the data are associated.

```
// Computation of a dot product
__kernel void
dot_product (__global const float4 *a,
             __global const float4 *b,
             __global float *c)
{
  int gid = get_global_id(0);
  c[gid] = dot(a[gid], b[gid]);
}
```

Program 3.4: Simple dot product in OpenCL.

The expression of the computation is quite simple and intuitive to a C-programmer, but the code requires a lot of setup and post-processing to be done. The exhibition of this setup code exceeds the available space here and does not contribute much to the understanding of the functionality of OpenCL. It can be examined by the interested reader in the appendix of the OpenCL standard. There is no available compiler for OpenCL yet. The first to be released will be part of Apple's next operating system.

3.4. OpenMP

Simply put, *OpenMP* is an API for writing multi-threaded applications. The first OpenMP standard was published in 1997 for Fortran. In 1998 the specification for C/C++ followed. OpenMP 2.5 was the first version to unify the standards for the different branches in 2005. The latest version of OpenMP, 3.0, was released in 2008.

OpenMP implements a fork-join model of parallel execution. A program starts with a master thread which splits into several threads where the programmer marks a parallelizable region and joins them afterwards. These regions can be specified with the compiler directive

```
#pragma omp parallel
{
  // code to be executed in different threads
  //  simultaneously...
}
```

Thus, OpenMP is an extension to the two most popular scientific sequential programming languages by adding a set of directives. These directives ease the thread programming by a great amount and also offer synchronization and management of shared memory resources.

Consider the sample Program 3.5 which distributes the generation of five random numbers to five different tasks. The actual number of tasks can be set with the function `omp_set_num_threads()`. Additionally, the ID of the current thread can be read with `omp_get_thread_num()`. The directive `omp parallel` denotes the compiler that the follow code block is to be executed in parallel by invoking different threads.

```
#include "omp.h"

int main (int argc, char **argv)
{
  int numbers[5];

  omp_set_num_threads(5);
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    numbers[id] = rand(100);
  }

  return 0;
}
```

Program 3.5: Thread parallel execution in OpenMP.

OpenMP also includes the possibility to express loop level parallelism. This is done with the directive

```
#pragma omp for
for (i=0; i<max; i++)   //... loop body
```

It forces the compiler to share the work in the different loop iterations between different tasks. Program 3.6 shows a single loop attached with the worksharing directive. This program also shows the use of the memory protection. By declaring an assignment to a variable or memory location critical, it can only be accessed by one thread at a time. The effect using `omp for` shows in the resulting array. When the computations inside the loop are distributed the field `numbers` is filled with different task IDs. Omitting the worksharing directive results in the same task ID for all array elements, as the last task to access the field overwrites it.

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char **argv)
{
  int numbers[5];
  int i;

  omp_set_num_threads(3);
  #pragma omp parallel
  {
    #pragma omp for
    for (i=0; i<5; i++) {
        int id = omp_get_thread_num();
        #pragma omp critical
        numbers[i] = id;
    }
  }

  for (i=0; i<5; i++) printf ("%d\n", numbers[i]);
  return 0;
}
```

Program 3.6: Worksharing of loop iterations in OpenMP.

The parallelism introduced with OpenMP directives is on the thread level. It is best used for synchronous multi-processor programming. The integration of the directives into common programming languages encourages its use. Still, the exploitation of SIMD parallelism is not supported or in other words left to the underlying C,C++ or Fortran compiler. The possibility to express the independence of loop iterations could be used

for SIMDization but conflicts with the concept of distributing parallel computations to different threads.

3.5. Other Parallel Programming Languages

In the late 1980s and the early 1990s parallel programming was on the rise and so, various data-parallel programming languages emerged in that time. *C**[5] and *mpl*[7] were designed with highly parallel architectures in mind. *High Performance Fortran*[27] and *Sm-1*, later *NCX*[63] aimed for versatile parallel descriptions. Still, the programming model of those languages does not fit very well with the short vector SIMD hardware aimed at here. Because of the complexity introduced by many data-parallel programming languages, a big enough user base could not be established for continuous development. There are three further data-parallel programming languages worth mentioning; two of which, SWARC and 1DC, come with a compiler that is capable of generating efficient SIMD code for multi-media instruction sets. Brook, on the other hand, is well known for its GPU backend.

3.5.1. Brook

The stream computing language *Brook* was developed as part of the Merrimac project[10] for stream computing in general. As one possible target architecture, it supports GPUs of various kinds. The system for GPGPU programming was presented in 2003 and is called Brook for GPUs[3].

As in traditional stream programming, Brook lets its user specify data streams of predefined data types or user defined structs. The kernels are written in a C-like language and support special stream operations like reduction, scatter and gather operations.

The GPU backend of Brook produces *Cg*[34] code which then can be compiled and linked with the Cg tool chain. At its heights Brook was a preferred choice of GPGPU programmers[3].

3.5.2. RapidMind

RapidMind aims at easing the development for multi-core systems. It works slightly differently than the other languages discussed here. Its API is embedded into C++ code. Thus, the complete workflow of a C++ application to be parallelized with RapidMind remains unchanged. By including a header and linking the runtime library, RapidMind kernels can be integrated into the C++ code as the same syntax is used.

The parts of the application to be run in parallel are encapsulated in function calls to RapidMind programs. These programs then operate on data streams and distribute them at runtime to the different cores. A SPMD (Single Program Multiple Data) stream processing model is used for RapidMind, allowing for scaling to larger numbers of processing

[3]Brook+ is now ATI's streaming language.

entities. The runtime library is able to dynamically manage the workload between the different cores.

The RapidMind platform provides various types of backends. One for x86 CPUs from Intel and AMD, a GPU backend supporting various graphics processing units from NVidia and ATI, and a Cell BE backend to distribute the RapidMind programs on the SPEs. Special optimizations for SIMD processing units for the supported targets are not available.

3.5.3. SWARC

The *SWARC*[11] language is a vector-based language designed to write portable SWAR (SIMD Within A Register) code modules. Those modules then can be integrated into C code to form a complete application. The SWARC syntax itself is close to C and allows the specification of vectors of arbitrary size and precision. The compiler for the language is called Scc and is thought to support a multitude of targets. With the possibility of selecting different precisions, the compiler has to select between the conversion instructions and operations offered by the respective target. The compiler generates C code containing a form of inline assembly that can be linked together with the rest of the application. So far, the Scc is known to generate code for MMX and 3DNow![43].

3.5.4. 1DC

To best make use of the parallelism offered by multi-media extensions in the field of image processing, the data parallel language *1DC*[28] has been developed. 1DC is intended to succinctly describe parallel image filters.

The underlying programming model understands the hardware as a linear processing array (LPA) with as many processing elements as columns are in the image. From the different parallel processing methods, i. e. in which way the pixels are processed, only the row-wise is applicable for SIMD instructions as the indexed addressing required for the slant-systolic method or the autonomous method cannot be implemented efficiently.

The syntax of the language is understood as an enhancement of C. Six different operations, such as rotations or grouping operations essential to specifying image filters, are added. The *sep* data type is used to describe vectors that hold as many scalar elements as there are PEs in the LPA.

The compiler, named 1DCC, works in three stages. At first the 1DC description is translated into an intermediate representation called macro code. This is fed into a macro code optimizer and finally into the code generator. The compiler supports MMX and SSE instruction patterns and shows good performance on convolution filters, edge detection and other pixel operations.

3.6. Evaluation

Direct programming aside, this chapter presented a selection of high-level parallel programming languages which could be used to program multi-media extensions such as the Streaming SIMD Extensions or AltiVec. However, none of these languages comes with a compiler able to generate efficient code for those SIMD units. In conclusion let us take a look at the pros and cons of the various languages in the context of data-parallel programming of scientific algorithms.

StreamIT fully embraces the stream programming model looking like a graph description language for stream graphs. The different filter constructs let the user keep a good overview of his application. StreamIT misses neighboring operations, i. e. accesses to adjacent stream elements. Those operations are common to many scientific algorithms, especially in the field of image processing. This moves StreamIT applications deliberately into the DSP domain, making it not as attractive for scientific programming as it could be.

+ Drawn through stream programming model.
+ Clear program structure.
– DSP domain, no neighboring operations.

OpenCL aims to be the new standard for data-parallel programming. It gives solid abstraction models for all parts of the system. It is very ambitious with its comprehensiveness and is pushed by a major part of the software and hardware development industry. For now it is only the standard and applications have yet to be written and evaluated regarding their performance. The language itself also includes typical GPU operations like dot products and swizzling[4], leaving great opportunities for SIMD optimizations. All in all, this could probably become the way data-parallel applications will be programmed in the future although the setup, data distribution, and collection code are very extensive. For smaller applications it is likely to outsize the core algorithm itself.

+ Comprehensive standard aiming to unify data-parallel programming.
– Extensive setup code, discouraging implementation of small to medium sized algorithms.
– No compiler available yet, no programming experience.

OpenMP is perhaps the best known parallel-programming language but does focus more on task parallelism than on data-parallelism. The worksharing directive `#pragma omp for` could be a toehold for SIMD optimizations yet no efforts have been made there. While this fits the multi-media instruction sets, it would not benefit other targets such as GPUs for instance. The reason here is the fork-join-parallelism. Data would

[4]Rearrangement of elements of a vector.

need to be transferred at points marked for parallel execution and after those transfered back. This could happen several times in the same algorithm wasting too much time on back and forth copying of data, slowing the resulting application by a great amount.

+ Used for over a decade, widespread user community.

– Designed for task parallelism.

– Fork-join model not suitable for every SIMD hardware.

Brook and **RapidMind** (formerly SH) emerged in the GPGPU period in the years 2003 to 2005 when there was no proper interface provided by the hardware vendors. Brook+ is now ATI's streaming language, whereas RapidMind evolved into a commercial software product. Though different in appearance, both Brook and RapidMind offer the potential to exploit SIMD parallelism. In fact, the Cell BE backend and the x86 backend of RapidMind could be improved with the SIMDization of the data-parallel functions making them a viable choice for possible SIMD exploitation.

+ Heavily used in GPGPU programming.

+ Different approaches but well designed.

– No compiler backend for multi-media instruction sets.

1DC is specialized to image filters and only mentioned here for the sake of completeness as it possesses a working SIMD code producing compiler. Unfortunately, 1DC seems to have been discontinued and there is no compiler available (anymore).

+ Working SIMD compiler presented in the paper.

– Limited to the specification of image filters.

– Discontinued.

SWARC offers vector-based computations integrated into C code but the abstraction is not as nicely done as in other data-parallel languages. Despite standard operations, the control flow is designed around masks to exclude computations on certain vector elements. Concepts like scatter, gather, reduce or neighboring have to be implemented circumstantially by the programmer. As with 1DC, SWARC seems to have been discontinued with no available compiler.

+ Working SIMD compiler.

– Stream operations like scattering, gathering and reduction are not integrated in the language.

– Discontinued.

CHAPTER 4

CGiS

Since this work focuses on SIMD code generation from CGiS programs, the data-parallel programming language CGiS is covered in this chapter. CGiS aims to express scientific algorithms based on stream computing and thus directly exposes parallelism. Here, a closer look at the CGiS language will be taken, its key design goals, as well as its syntax and semantics. The chapter starts with an introductive example in Section 4.1 followed by the discussion of the design goals in Section 4.2. To delve deeper into the language, we will examine the grammar of a CGiS program with a top-down approach. Syntactically speaking, every CGiS program is seperated into three parts: INTERFACE section, CODE section and CONTROL section. Each of these is presented in Section 4.3 to 4.5. The concept of hints is explained in Section 4.6. Finally, Section 4.7 gives a summary of the language.

4.1. Overview

To give a first impression we examine the CGiS program presented in Program 4.1. It computes the alpha blending of an image onto an opaque background. Each point of the digital image background, called pixel, gets blended with the color values of an input image according to its alpha value. Alpha is a value between 0 and 1 and describes the image's opaqueness. The pixels here are stored in vectors of three floats, holding red, green and blue color values.

Every CGiS program starts with the keyword `PROGRAM` followed by its name and then is strictly divided into three sections, `INTERFACE`, `CODE` and `CONTROL`. The INTERFACE section is concerned with the naming and passing of input and output data. Here, two streams of arbitrary but equivalent size are specified. One only serves as input stream, thus can only be read; the other is marked as `inout` which allows reading and writing. A third scalar value is passed, holding the alpha value for the blending operation. The CODE section contains all the kernels of a CGiS program. The kernels operate on single elements of a stream. In Program 4.1, the CODE section includes

the kernel `blend` which computes the blending operation for a single background pixel. This example shows, amongst others, the possibility of scalar multiplication by simply multiplying `alpha` to a vector. The CONTROL section describes the iteration over the streams by `forall` loops which call the kernels from the CODE section. The calls are executed in parallel on the different stream elements.

```
PROGRAM alpha_blend;

INTERFACE

extern in    float3 image<SIZE>;
extern inout float3 background<SIZE>;
extern in    float  alpha;

CODE

procedure blend(in    float3 ipixel,
                inout float3 bg,
                in    float  alpha)
{
  bg = alpha * ipixel + (1 - alpha) * bg;
}

CONTROL

forall(in_pix in image, bg_pix in background)
  blend(in_pix, bg_pix, alpha);
```

Program 4.1: Alpha blending of an image on an opaque background

This small example provides a first glance at what a CGiS program looks like. There is an obvious closeness to C allowing easy access for most programmers, but most importantly the parallelism is explicit in the language by independent iterations over streams with forall loops.

The concept of having different sections for different parts of the program divided by script-like key words can be astounding. The purpose of these separating key words is not to distinguish CGiS from well-known programming languages but to give the program itself and also the way a programmer thinks about the implementation of his algorithm a clear structure. The parallelism is given by the CONTROL section, as it issues kernels to operate on independent stream values. The sequential operations then are performed inside the kernels specified within the CODE section. Data declaration is done in the INTERFACE section which is the connection to the outside world.

Figure 4.1 shows the usage pattern of CGiS. A source code file is fed to the compiler, which outputs code for the desired target and code for interfacing with the main application. Arrows denote in and output; dotted lines denote linkage. The filled rectangular

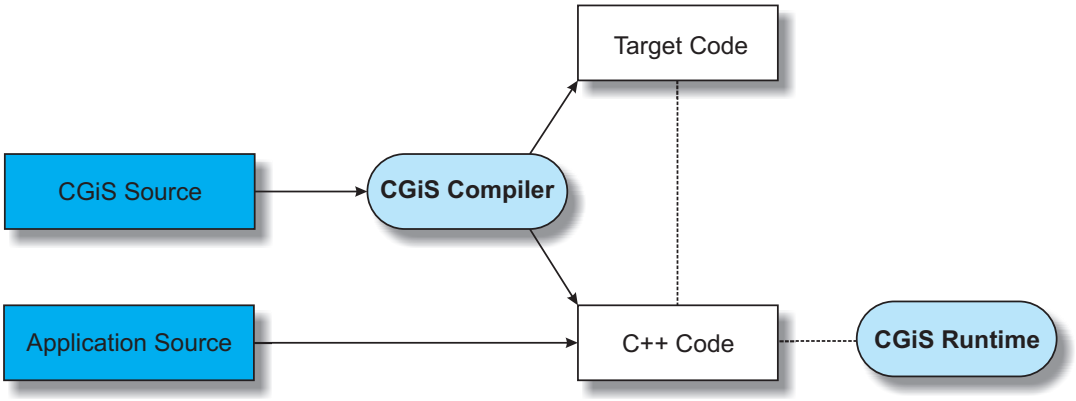


Figure 4.1.: The basic usage pattern of CGiS.

nodes are user-supplied code. The oval nodes are part of CGiS, and the other code components are generated by the CGiS compiler. The programmer interacts only with this interface code in a uniform way. For the alpha blending example, the C++ code to set up and run the generated CGiS code is shown in Program 4.2.

The CGiS compiler generates a set of setup, execution and cleanup functions which are all prefixed by the CGiS program name, here `alpha_blend`. At first a new CGiS program control structure, `blend`, has to be created. Depending on `TARGET_ARCH`, the appropriate initialization functions are called. The stream sizes must be passed and the initializing routines must be called. This results in the generation of needed temporary structures and buffers. Next, the pointers for input data are passed to the part generated by the CGiS compiler and the runtime library with the `set_data_` prefixed functions. For each variable declared in the INTERFACE section of a CGiS program, an adequate enumeration value is created to connect the application data with the CGiS part. All input data, qualified with `in` or `inout` in CGiS must be passed, e.g. the data of the image to be blended is pure input and is passed by `set_data_alpha_blend(CGiS_alpha_blend_image, app_image)`. Setting up data is finalized by calling `setup_data_alpha_blend`. The computations specified in the CGiS program are started by a call to the function `execute_alpha_blend`. It runs the iteration sequence of the forall-loops. Output data, denoted with the qualifier `inout` or `out` in CGiS programs, can be collected by the `get_data` prefixed functions. Finally, functions for cleaning up and deleting the internal temporaries should be called.

The final application binary is built from the C++ code of the application, the generated sources from the CGiS compiler and the runtime library.

In the following, the data-parallel language CGiS is examined further and in more detail starting with its design decisions motivating the appearance of CGiS programs.

```
// application data
float3 image[size]; float3 bg[size]; float alpha;

// ... application data initialization

// create a CGiS program command structure
CGiS_program* const blend = get_CGiS_program(TARGET_ARCH);
setup_size_alpha_blend(CGiS_size_alpha_blend_SIZE, app_size);

// register program and start initialization
register_program_alpha_blend(blend);
blend->init_all();
setup_init_alpha_blend();

// pass data pointers
set_data_alpha_blend(CGiS_alpha_blend_image, app_image);
set_data_alpha_blend(CGiS_alpha_blend_background, app_bg);
set_data_alpha_blend(CGiS_alpha_blend_alpha, &app_alpha);

// finish the data setup and create internal structures
setup_data_alpha_blend();

// start the alpha blending
execute_alpha_blend();

// get back the resulting data and clean up
blend->finish();
get_data_alpha_blend(CGiS_alpha_blend_background);
cleanup_alpha_blend();
delete blend;
```

Program 4.2: Application interface to the generated CGiS code.

4.2. Design Decisions

The designer of a programming language should have two sometimes contradictory intentions in mind. First is pleasing the user who should be encouraged to write a program by the intuitiveness of the language, its features, its expressiveness or its simplicity. Secondly the language has to be close enough to the targeted hardware to let compilers generate efficient code. Not all of those motivational characteristics inevitably go hand-in-hand. Let us examine the key decisions in the design of the language CGiS. The following list summarizes the corner pillars of the design of CGiS.

- **Familiarity:** the language should be familiar to an average programmer. We have designed CGiS with C in mind to shorten the adaptation process.

- **Readability:** a program written in CGiS is supposed to have a clear structure separating different aspects. Programmers, even when not familiar with the language, should be able to understand its purpose.

- **Usability:** a programmer should not need to know details of all target characteristics. Of course, a general knowledge of what is possible at all may be required. After all, every programmer needs to know some aspects of language implementation and hardware features to decide on efficient data structures or algorithms in high-level languages as well. Although CGiS aims at scientific programming, and indeed its whole reason for existence is the presumed time efficiency of implementations of time sensitive algorithms, a higher abstraction level than intrinsics and assembler would endorse widespread usage. High enough abstraction is also a prerequisite for portability.

- **Efficiency:** scientific programmers willing to use a non-standard programming language likely do so to increase the performance of their application. High performance can only be gained with the generation of efficient code. We want CGiS to be as light-weight as possible, so that its compiler is feasible to generate efficient code.

- **Portability:** one of our main goals was to accommodate the steady stream of new GPU architectures starting from the existing ones. The language must be robust to the evolution of GPUs for enabling the user to get the most out of new hardware without rewriting his code. This work takes the goal of portability a step further by opening up common SIMD hardware as targets for CGiS applications. A unified language is a prerequisite for portability: The programmer should not need to rewrite his implementation on an evolutionary step of hardware.

- **Adjustability:** with a wide range of target support, expert programmers must be given the opportunity to fine tune their application for a specific target without impairing the performance on other targets.

- **Expressiveness:** the user should be able to express a complete algorithm, including both the computation that is ultimately mapped to the target hardware and the control flow to be performed on the host CPU, in the same language. This is particularly important because it is a prerequisite for many of the remaining goals.

So the key for a consenting user base is balance. CGiS aims for it in different ways, e.g. it is a data-parallel programming language with explicit parallelism but its syntax and semantics are close to C avoiding alienation of new programmers.

Looking back at the first CGiS example in Program 4.1, some of the desired goals are reached at first sight. The purpose of the program is understandable to everyone familiar with C programming. The abstraction level of the language is high enough, no hardware peculiarities are exposed. The algorithm is expressed completely in CGiS, no pre or postcomputations need to be done in the main application. Of course this

program cannot speak for all possible data-parallel algorithms. Here it is merely a help to give a better insight into the design decisions.

An exhaustive discussion of the syntax and semantics of CGiS is presented in [32]. Here, the language is discussed in top-down approach by inspecting the three different sections in a CGiS program one after another. As seen in example Program 4.1 the three main sections are:

- **INTERFACE:** This part contains all the data that passes through a CGiS program.
- **CODE:** All the kernels in the underlying program are defined here. Kernels describe *sequential* computations on single stream elements.
- **CONTROL:** This section is a series of forall loops iterating over streams calling the kernels listed in the CODE section. The computations on each stream element are independent and can be executed in *parallel*.

The grammar of a CGiS program looks like this:

```
PROGRAM  ::= PROGRAM ID;
             INTERFACE INTERFACE_SECTION
             CODE CODE_SECTION
             CONTROL CONTROL_SECTION
```

Throughout the following sections, the remaining parts of the CGiS grammar are presented in the same manner. The production rules are highlighted in gray. Non-terminal symbols are in normal font only in upper-case; terminals are in bold.

```
GRAMMAR_RULE  ::= NON_TERMINAL terminal;
```

Possible alternatives for symbols are separated by a |. There are a few encapsulations to ease the description of the grammar rules.

- `[.]+`: at least one occurence of the encapsulated symbols must be present.
- `[.]*`: an arbitrary number of encapsulated symbols can be present.
- `[.]?`: stands for optional content.

The non-terminal symbol `ID` will be used in various places whenever a variable name should be placed. The naming rules for variables are the same as in C.

4.3. INTERFACE Section

This section contains all the data that is processed in the program. In the context of stream programming one can think of it as an abstraction of the sources and the sinks in a stream graph[57].

4.3.1. Basic Data Types

CGiS supports the following data types for data declared as part of the INTERFACE section:

```
TYPE      ::= BASE_TYPE | ID;
BASE_TYPE ::= int | int2 | int3 | int4 | uint | uint2
            | uint3 | uint4 | float | float2 | float3
            | float4 | bool | bool2 | bool3 | bool4
USER_TYPE ::= struct { COMP_LIST } ID ;
COMP_LIST ::= COMP_DEF, COMP_LIST | COMP_DEF
COMPDEF   ::= BASE_TYPE ID
```

The short vector types are common to many data-parallel and especially GPGPU languages, which is why they are also integrated into CGiS. Also CGiS allows structs of arbitrary length.

4.3.2. Data Declaration

There are two different kinds of variables in CGiS: *streams* and *scalars*. Throughout the remainder of this chapter a CGiS scalar refers to a non-stream variable. This means a scalar can also be one of the basic short-vector types. Scalars declared in the INTERFACE section are constant across iterations over streams.

The basic declaration of a variable consists of a location specifier, a flow specifier, a type, a name and an optional stream qualifier. A stream declaration always has stream qualifiers ('<', '>') attached to the stream name. Either one or two sizes must be given for the dimensions, depending on whether the stream has one or two dimensions. They can also be symbolic constants. The location specifier tells the compiler where the variable's value is to be taken from, either an external source or local. The flow specifier gives information about the data flow of the variable, namely whether the data is only read, written or both read and written.

```
INTERFACE_SECTION ::= [IFACE_VAR_DEC]+
   IFACE_VAR_DECL ::= ISCALAR_DECL | ISTREAM_DECL;
     ISCALAR_DECL ::= LOCATION IFLOW SCALAR_DECL
      SCALAR_DECL ::= TYPE ID
     ISTREAM_DECL ::= LOCATION IFLOW STREAM_DECL
      STREAM_DECL ::= TYPE ID<DIM_SPEC>
         LOCATION ::= extern | intern
            IFLOW ::= in | out | inout
         DIM_SPEC ::= [XSIZE | _] | [[XSIZE, YSIZE] | [_,_]]
```

The variables declared in the INTERFACE section are only visible in the CONTROL section. There are no such things as global variables known from common programming languages.

C-like pointer types are also not available in CGiS. Memory is always represented as arrays which can be indexed by numerical values. Thus, it is not possbile to dynamically allocate memory.

4.4. CODE Section

The CODE section contains the description of the kernels in a sequential C-like language. Again, in comparison with a stream graph, this section describes the nodes of the graph.

Each kernel is started by the keyword **procedure**, followed by a list of parameters passed to the kernel invocation. The definition of a kernel very much equals the definition of a C function; though procedures do not have return values, as possibly many output values are produced per call. CGiS uses call-by-value-result semantics[51].

A procedure definition has the following grammar:

```
CODE_SECTION ::= [PROC_DECL]+
   PROC_DECL ::= procedure ID ( PARAM_LIST )
                 {
                   [STMT]+
                 }

   PARAM_LIST ::= [PARAM_DECL, PARAM_LIST] | PARAM_DECL
        PFLOW ::= IFLOW | reduce
   PARAM_DECL ::= PFLOW [STREAM_DECL | SCALAR_DECL]
```

Parameters of procedures also require a flow specifier which tells whether the value of the corresponding parameter is only read, written or both. In that manner, input and output parameters can be arbitrarily listed in the procedure header. This allows for procedures having multiple output values declared in a uniform way. Additionaly, the variable can be specified as `reduce`. This is a special flow specifier used for implementing a reduction operation.

Reduction

The reduction operation is a binary operation applied to all elements of a stream reducing it to one scalar. The typical reduction procedure is shown in Program 4.3. It computes the sum of all stream elements.

4.4.1. Statements

In the specification of the CODE section there is no difference if the kernel is called from another kernel or from the CONTROL section. The body of a procedure consists of a non-empty list of statements. There are six different kinds of statements: calls, conditions, loops, blocks, assignment series and data accesses.

```
INTERFACE
extern in  float reduce_me<_>;
extern out float result;

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

CONTROL
forall(d in reduce_me)
  sum(d, result);
```

Program 4.3: Reduction of the stream `reduce_me`.

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

A *call* in CGiS is different from a call in C, in that it does not produce an explicit return value and thus cannot be the right side of an assignment. Each procedure called must be specified in the CODE section.

Conditions are if-then or if-then-else constructs for diverging control flow. They are specified and used as in C. CGiS supports three kinds of *loop constructs*: for, while and do. The for and the while-loops perform their termination test before every new iteration whereas the do-loop tests for termination after each iteration. In connection with loops, the statements *break* and *continue* can also be used with the same functionality as in C. Both can only be used inside a condition. Opening a new *block* opens a new environment allowing the usage of variables local to that block. Shadowing of variables is possible. A *series of assignments* is a non-empty comma-separated list of assignments. An assigment is required to have a possibly masked variable on the left side and an expression on the right. Expressions are presented in Section 4.4.2. Masking is a form of excluding certain components of a vector from an operation and is explained further in Section 4.4.3. There are five different assignments, four of which implicate an arithmetic operation along with the variable on the left side as the destination and first operand. For example `a += b` is equivalent to writing `a = a + b`. *Data accesses* are memory loads and stores. There

are three different kinds: lookups, gathers and write-backs. All of these are explained below in Section 4.4.4.

4.4.2. Expressions

The right side of an assigment in CGiS is called expression. An expression can be subdivided into five kinds and is related to an operation or an atomic construct like a variable or a constant. Unary, binary and ternary expressions refer to the number of operands an operation has, while a function expression refers to a number of predefined mathematical functions such as *sin* or *ln*. An expression can also be enclosed in brackets to override precedencies of operations.

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

There are various arithmetical operations in CGiS which can either be expressed by symbols, predefined functions or other operator keywords. Most of these operators are known from mathematics or other programming languages, however CGiS features a few operations worth mentioning in more detail:

- **hor** and **hand** are horizontal logical operations performing the logical 'and' or the logical 'or' on the components of a single vector operand returning a scalar result.
- Shifts and rotations can be performed by the operators $<<$ or $<<<$ for left shifting/rotating and $>>$ or $>>>$ for right.
- $\#$ is a cross product operator for variables of type float3.

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

- $\backslash$ is a dot or inner product operator.

$$a \; b = a_x b_x + a_y b_y + a_z b_z$$

Rank	Operation
highest	swizzling, functions, unary −
.	* / # \
.	<< >> <<< >>>
.	+ −
.	< <= >= >
.	== !=
.	max min
.	not ~
.	&
.	^
.	|
.	and
.	or
lowest	? :

Table 4.1.: Precedences of operators in CGiS

- ☐ Swizzling, which is basically a composition of a new vector from elements of a given one, is further explained in Section 4.4.3

Naturally, there is a default precedence order for the operators in CGiS. It is shown in Table 4.1.

4.4.3. Masking and Swizzling

Masking and swizzling are two concepts alien to common C programmers. Both stem from the original idea of designing CGiS as a GPU programming language and reflect their frequent usage as a hardware feature of graphic cards ALUs.

The procedure `foo` in Program 4.4 serves as an example for masking and swizzling. This program part also shows *scalar expansion*. In CGiS scalars can be assigned to a vector resulting in an expansion of the scalar to a vector of the appropriate number of components. The assignment `c = a.x;` results in each component of `c` having the value of `a.x`.

Masking

Masking is a way of selecting components of a vector type to be modified, thus excluding others from the assignment. The mask can either be static by explicitly naming components or dynamic by attaching a boolean variable.

```
MASKED_ID ::= ID . [x|y|z|w]+[a]
            | ID:ID
```

[a] Special masking rules apply.

```
procedure foo (in float3 a, in bool4 dmask, out float4 b)
{
  // scalar expansion to float3
  float3 c = a.x;

  // masking of b, excluding first component
  b.yzw = [1.0, 2.0, 3.0] + c;

  // masking of b, depending on a variable
  b:dmask = [0.0, 1.0, 2.0, 3.0];

  // swizzling of a, reverting component order
  b += a.wyzx;
}
```

Program 4.4: Masking and swizzling in CGiS.

The statically masked assignment to `b` in Program 4.4 equals to the sequence

```
b.y = 1.0 + c.x;
b.z = 2.0 + c.y;
b.w = 3.0 + c.z;
```

Here the first component of `b` is ignored.

In CGiS the usage of masks follows some restrictions:

- ☐ Masks are ordered. It is not possible to change the order of the components as in `b.zy = [2.0, 1.0];`.
- ☐ There are no gaps in masks[1]. The programmer is not allowed to write something like `b.yw = [1.0, 3.0];`.

As masks are only used for excluding components, it is impossible to create masks with more components than the original vector inhibits. Double-mentioning of components in masks is also prohibited as the result of the operation is then unknown.

The dynamic mask `b:dmask` excludes the components that have a corresponding false value in the boolean vector `dmask`. The boolean vectors used for dynamic masking have to match the component number of the vector to be masked.

Swizzling

Swizzling is an operation that creates a new temporary vector of arbitrary components of the swizzled one.

[1]This is a restriction for SIMD code generation.

```
SWIZZLE ::= ID . [x|y|z|w]+[a]
```

[a]Special swizzling rules apply.

Considering the swizzling example in Program 4.4 the assigment `b += a.zzyx;` equals to the sequence

```
b.x += a.z
b.y += a.z
b.z += a.y
b.w += a.x
```

This example shows the great flexibility swizzling offers. It is possible to create arbitrary vectors resembled out of components of a given vector. Here the vector a is reverted and, to match the size of vector b, the last component of a is doubled.

Contrary to masking there are no restrictions on swizzling but the fact that it only works on the predefined short vector types.

4.4.4. Data Accesses

There are three different kinds of data accesses:

- ☐ **Lookups:** indexing of streams.
- ☐ **Gathers:** neighboring operations on the current stream element.
- ☐ **Write-backs:** stores to a specific memory location.

They can be specified as follows.

```
DATA_ACCESS ::= LOOKUP_SPEC | GATHER_SPEC | WRITEBACK_SPEC
```

Lookups

Lookup operations correspond to indexing operations in other languages. The looked-up stream is considered an array which can be indexed by a variable. That variable holding the position can be of type integer or floating-point; the latter results in a runtime conversion to integer. A stream can also be looked up several times with different indices and resulting variables. Simple pairs of variable name and position can be used for that. There is no restriction on the number of lookups performed. Depending on the dimensions of the stream, one or two position variables are needed to complete the lookup operation.

```
LOOKUP_SPEC  ::= lookup ID : [ABS_POS]+ ;
ABS_POS      ::= ID < ID > | ID < ID, ID >
```

Program 4.5 shows the use of a simple lookup or indexing operation. Two streams of 16 elements each are iterated over `indices` and `results`. The first contains a set of indices that should be derived from the stream `vals`. The derived values are then to be saved in the second stream.

In the procedure `array_lookup`, with the keyword **lookup** after the parameter declaration an indexing operation on the stream `whole` is signaled. Also, the index `position` has to be passed as a parameter to the procedure. The result of the lookup operation is placed in the variable `wanted`.

The usage of a lookup statement also implicitly defines the variable of the stream's base type, the result of the lookup operation is stored in. All lookups take place at the beginning of a procedure and the defined variables are read-only. The user is responsible to care for avoiding out-of-bounds accesses.

```
PROGRAM indexing;

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

CODE
procedure array_lookup (in float data<_>, in int position,
                        out float value)
{
  lookup data: wanted<position>;
  value = wanted;
}

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

Program 4.5: The stream `indices` is looked up.

Gathering

In contrast to lookups, gathering operations work relative to currently processed stream element. That is why they are also called neighboring operations. They are useful in many algorithms varying from image processing (blurring, edge detection etc.) to scientific simulations such as boiling simulation of liquids or cellular automatons. As the gathering operation is relative to the current element, the iterator of the stream must be passed as parameter to the function using gathers. A stream iterator is the variable that always holds the stream element which is processed in the current iteration. It is linked to its stream and its position in the stream. This means that a kernel using the gathering operation has to be called directly from the CONTROL section.

In CGiS, gathering operations are specified very similar to lookups. After the keyword **gather**, the stream iterator to be gathered from must be given. Again, this stream iterator also needs to be a parameter to the procedure hosting the gathering operation(s). Separated by **:** follows the name of the newly defined variable that should hold the result of the gathering operation. The offset to the current stream position is then given in <, >. Lists of gathered variables of arbitrary length are possible as well.

```
GATHER_SPEC ::= gather ID : [REL_POS]+;
REL_POS ::= ID < INT_VAL > | ID < INT_VAL, INT_VAL >
```

As an example, consider Program 4.6 which holds a simple gathering operation as it could be used in Conway's game of life simulation[2]. Here it is only checked whether a cell has more than 6 neighbors. If so, it dies, i. e. its value in the stream is set to 0. The stream `cells` is a two-dimensional stream containing integer values, here 0 or 1. In the CONTROL section the stream iterator `c` is passed to the procedure `perish`. This procedure gets the parameter `current` on which the gathering operation is performed. The eight adjacent neighbors of the variable current are stored to the respective variables, e. g. `tl` gets the value of the stream element that is left and above the current; denoted by the offsets `<-1,1>`.

```
PROGRAM part_of_life;
INTERFACE
extern inout int cells<_,_>;
CODE
procedure perish (out int current)
{
  gather current: tl<-1,1>,  t<0,1>,  tr<1,1>,
                   l<-1,0>,            r<1,0>,
                  bl<-1,-1>, b<0,-1>, br<1,-1>;
  int sum = tl+t+tr+l+r+bl+b+br;
  if (sum > 6) current = 0;
}
CONTROL
forall (c in cells)
  perish (c);
```

Program 4.6: Simple gathering operation in CGiS.

Though simple at first glance, the gathering operation internalizes a few details to consider. The relative accesses to stream elements can be out-of-bounds. Looking at the example Program 4.6 already the first element of the stream causes five out-of-bounds accesses. There are two solutions to this problem,

- ☐ wrap-around and

[2] A simple cellular automaton where cells die or come alive depending on the number of adjacent alive cells.

- ☐ closest match or clamping.

The first one is to consider the data as a ring (one-dimensional stream) or a sphere (two-dimensional stream). If you reach behind the first element you get to the last element of the stream. The second is used by default in CGiS. The access is clamped in the way that it always matches the stream boundaries. For example the variables `l`, `tl` and `t` in Program 4.6 would have the same value as `current` for the first element of the stream. The checks for clamping are free on GPUs as it is implemented in hardware, thus it is the default handling of out-of-bounds accesses.

Also, the gathering operation induces one restriction to the CGiS language. It can only be used in kernels that are called directly from the CONTROL section because of the connection to the iterator. This connection is lost through subsequent calls made from inside the CODE section.

Write-back

Random writes or stores to arbitrary stream positions are realized with the write-back operation. Their semantics is just the opposite of lookups with the slight relaxation that the indexing variable does not need to be a parameter of the procedure. As an example for the write-back operation, consider Program 4.7.

```
WRITEBACK_SPEC ::= writeback ID : [ABSPOS] ;
        ABSPOS ::= ID < ID [, ID ]*>
```

```
procedure store (out int S<_>, in int3 a, out int sum)
{
  int pos = a.z; int val = a.x;
  sum = a.x + a.y;
  if (sum > 256) {
    writeback S: val<pos>;
  }
}
CONTROL
forall (c in cells)
  perish (c);
```

Program 4.7: A write-back operation in CGiS.

The effect of a statement '`writeback S: val<pos>;`' is to store the value `val` at the index `pos` in the stream `S`. Also, a sequence of value-position pairs is allowed. Should the same position be used multiple times, the results of the write-back are undefined for that position.

4.5. CONTROL Section

The CONTROL section is the part in a CGiS program where the parallel computations are issued. This section mainly consists of a series of forall loops iterating over streams specified in the INTERFACE section. The bodies of those loops may only contain calls to kernels defined in the CODE section. The CONTROL section features the following control statements:

- **forall:** initiates the iteration over specified streams.
- **show:** enables the possibility to visualize the results of streaming computations on GPUs. This is useful for algorithms where the interpretation of the computed data is to be shown as a picture anyway.
- **intrinsic:** an intrisic can be used to execute special functions on streams, integrated in the runtime library, such as matrix multiplications.

In this more compact description of the CGiS data-parallel programming language, the discussion of intrinsics as well as the show directive are omitted here. The interested reader can find more information in [32].

```
CONTROL_SECTION ::= [CONTROL_STMT]+
   CONTROL_STMT ::= FORALL_STMT | SHOW_STMT | INTRINSIC
    FORALL_STMT ::= forall ( [ FA_PDECL , | FA_PDECL ] ) FA_BODY
       FA_PDECL ::= ID  in ID
        FA_BODY ::= CALL | { [CALL]+ }
      SHOW_STMT ::= show ( ID ) ;
      INTRINSIC ::= [ matvecmul | matmatmul ] (ID,ID,ID )
```

Compared to the general stream programming model[57], the first restriction CGiS poses becomes obvious. The number of stream elements consumed and produced is always one. This restriction, though, is not a real burden but merely a way to enforce a certain amount of structure within the data streams to be processed. The programmer can define arbitrary data structures to shape the look of a stream element.

4.5.1. Semantics

Instead of re-specifying the semantics of the CONTROL section, only a brief intuitive explanation of how the parallelism works is presented here. The complete specification – again – can be found in [32].

Programming parallel hardware with purely sequential programming languages is futile, as uncovering hidden parallelism is too difficult. Therefore data-parallel programming languages emerged exposing parallelism more or less clearly. CGiS exhibits its parallelism by declaring all calls inside forall loops as independent. Thus, executing iterations of a forall loop in different ordering must not change the results of the computations. To ensure this behavior, a few terms must be set:

- Calls inside forall loops are distributive. So, a series of calls inside a forall loop body is equivalent to a series of identical forall loops with one call in their body each.

- Data updates occur after the kernel computations are done for each element of the iterated stream(s).

Both terms result from CGiS' closeness to the stream programming model. Strictly speaking, a kernel has input and output streams that are consumed and produced element-wise. So each kernel consumes whole streams and produces whole streams. Thus, subsequent kernels operate on complete output streams of previous kernels. This explains the distributivity of the kernel calls inside forall bodies.

The second term stems from the introduction of *inout* streams which are not supported in the orginal streaming model. Reconsider the gathering example in Program 4.6. An immediate update of the stream data would result in different computations on the same stream element depending on when the iteration reaches it, hence the alleged parallel computations would not be independent. Forcing the data updates to take place after the full processing of the stream lets them fit into the model again. Specifying a stream as inout is intended only to save storage space and shorten the description of certain algorithms.

4.6. Hints

The concept of placing *hints* at certain points in the program enables the experienced programmer to guide the optimizations of the CGiS compiler.

```
HINT_SPEC ::= #HINT ( [HINTLIST] )
HINT_LIST ::= HINT | HINT, HINT_LIST
     HINT ::= [PROFILE :]? ID [= INTCONST]?
```

Hints are introduced by the phrase **#HINT**. They only contain a simple keyword, or a keyword together with an assigned number. Optionally, a profile identifier can set the hint active only for the specified hardware or hardware group. Depending on the location of the hint, several possibilities for guidance arise.

- Loop counter hints: it is possible to declare a maximum count for a loop iteration variable by using the keyword `max_count`. Also, loop counters can be declared `uniform` which means they are independent of the current stream iteration.

- Avoid texture reuse (GPU only): the hint `no_texture_reuse` ensures that no texture is used for other streams in external programs.

- Guide control flow conversion: the conversion of conditionals, such as if-statements, can be enforced with `force_conversion` or disabled with `no_conversion`.

- Inlining can be avoided by using `never_inline`.

Hints can be a very powerful tool to increase the efficiency of the generated code. They should be used with care though, as for example the avoidance of inlining can induce severe performance penalties.

4.7. Evaluation

This chapter presented the CGiS language, a data-parallel programming language for scientific algorithms. With this application domain in mind, the syntax and semantics of the language embrace the targeted design goals very well.

The description of kernel computation is very close to C, ensuring familiarity. The clear structure of every CGiS program, separating it into data declarations, sequential code and parallel execution improve readability. While due to the high abstraction level, there are no hardware specific restrictions in CGiS guaranteeing usability, the concept of hints at various program spots enables a decent amount of adjustability.

The language constructs available and the possibility to express a whole algorithm in a single language together with the predefined functions fit the purpose of scientific programming, which fullfills the expressiveness design goal, allowing for high portability of the applications written with CGiS. With the exposure of parallelism and the lack of side-effects, CGiS compilers can achieve high efficiency of the generated code.

CGiS is based on the stream programming model and applies a few restrictions to it. In contrast to standard stream programming the computations on the different stream elements are independent. Instances of the same kernel can be executed in parallel accordingly. As the language lacks global variables, the problem of shared memory and competing accesses to it does not arise. Also, internal states for kernels are not supported. With the demand for data-parallel loops, states inside kernels depending on stream computations are not possible. This curbs the number of algorithms expressable with CGiS in a way that the resulting domain very well fits to parallel hardware such as GPUs and SIMD CPUs.

CHAPTER 5

The Compiler Framework

This chapter deals with the compiler framework of the CGiS system. The system consists mainly of four components:

- the CGiS language,
- the compiler `cgisc`,
- the runtime libraries and
- the infrastructure

The CGiS programming language has been covered in the previous chapter. Section 5.1 gives an overview of the system and explains the basic usage of the compiler. Following the traditional structure of a compiler, Section 5.2 presents the frontend and the intermediate representation, while in Section 5.3 the middle-end and the integrated program analyses are discussed. The GPU backend is briefly shown in Section 5.4 and the SIMD backend(s) in Section 5.5 is shown in more detail. In Section 5.6 the runtime system is discussed. The remaining parts of the infrastructure can be found in Section 5.7.

5.1. Overview

A typical compiler[61] consists of a frontend, a middle-end and a backend. While the frontend parses the input and translates it into an intermediate representation, the middle-end performs optimizations and the backend's task is to generate code for the selected target architecture. A coarse grain view of the tasks of the CGiS compiler, called `cgisc`, is given in Figure 5.1.

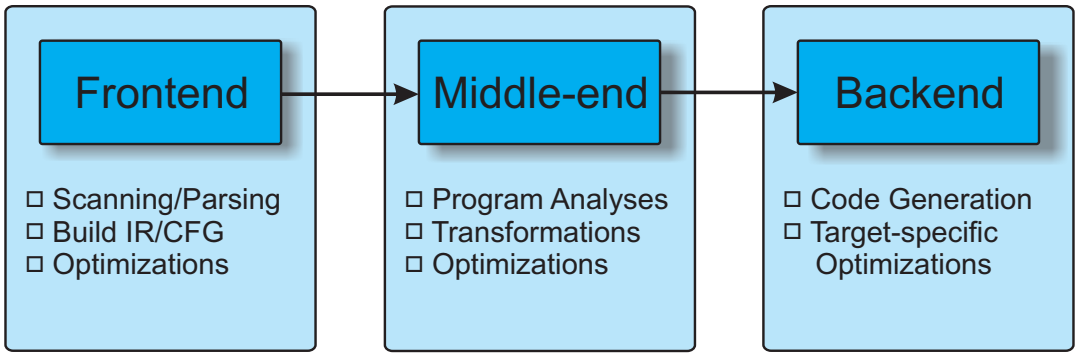


Figure 5.1.: Coarse grain overview of the CGiS compiler.

5.1.1. Invocation

The CGiS compiler is a command line application. Its general invocation looks as follows:

```
cgisc [-switch | --verb_switch] -P <profile> <source.cgis>
```

Besides *optional switches*[1] the invocation of `cgisc` demands a *profile* and a *source file*. A profile selects the target architecture. Depending on the profile, the compiler enables and disables certain optimizations. Profiles are further described in Section 5.1.2. The compiler supports many switches to guide compilation and extract additional information about the compilation process. It is possible to enable various kinds of different debugging output as well as selectively disable certain compiler optimizations. Among others, it is possible to gain information about the control flow graph, i. e. the internal representation of the program, during the compilation process at various compiler stages. This kind of visualization is explained in Section 5.2.3.

`cgisc` itself is designed to run on different platforms and on different operating systems. It can be compiled on any system featuring a `gcc` and a building environment with `automake`. Also, it can be compiled using Microsoft's Visual Studio and its C/C++ compiler `msvc`. The needed project files are part of the CGiS distribution.

Let us recall the alpha blending program (Program 4.1) from Chapter 4. Assume we want to compile for SSE hardware and get some insights into the control flow graph after the program has been parsed.

```
> cgisc -a P -P SSE alpha_blend.cgis
```

creates this set of files:

[1] The switch `-h` or `-help` displays the full list of options.

`alpha_blend.h`	Header file to be included by the main application code.
`alpha_blend.cpp`	This file contains the setup code as well as the routines for starting the iterations. All program dependent macros and wrapper functions are in here.
`alpha_blend_simd.inc`	This file contains the translated CGiS kernels as C++ functions with intrinsics.
`alpha_blend.parse.gdl`	Visualization of the CFG after parsing.

The project can be compiled with a standard C++ compiler and linked against the main application and the appropriate runtime. For this demonstration we use the Gnu C Compiler `gcc`, e. g.

```
> gcc -msse3 -Wall -DKIND=CGiS_SSE
        -I<path_to_runtime_header> alpha_blend.cpp
> gcc alpha_blend.o main.o -o ab -L<path_to_runtime_lib>
        -lsseruntime
```

In this example the `gcc` needs the appropriate instruction sets enabled. This can be done by activating the `-m<sse|sse2|sse3>` switch to generate SSE code. The resulting application will then use SIMD execution for the computations specified in the CGiS program. The compiling and linking process varies for the different target architectures. For example the Cell BE requires binary images to be uploaded to the SPUs. These images can be assembled with shell scripts that are also genereated by `cgisc`.

The SIMD backend generates code for the target architecture in form of compiler intrinsics. These can be embedded in normal C/C++ code and represent one or more assembly instructions. Programming with those intrinsics has been presented in Section 3.1. Excerpts of the SSE code resulting from the compilation of the alpha blending example can be found in Appendix B. To better understand the SIMD code, the remainder of this section describes the code transformations and optimizations required to produce efficient SIMD code.

The reasons why intrinsics are used over assembly or even binary code are twofold. First of all encapsulation of these intrinsics in functions is a comfortable way of integrating SIMD code into C/C++ code. The output of the SIMD backend is human readable. Secondly, the backend is freed from the burden of various standard optimizations and compiler stages that are present in C/C++ compilers anyway, e. g. register allocation or common subexpression elimination. Those are left to the C/C++ compiler used to compile and link the whole project, in our example compilation here the `gcc`.

5.1.2. Profiles

The CGiS compiler uses profiles to select the target architecture. Depending on the target, the compiler chooses different transformations, optimizations and code generation

Backend Category	Profile	Architecture
GPU	NV30	NVidia NV30 GPUs
	NV40	NVidia NV40 GPUs
	G80	NVidia G80 GPUs
	CUDA	NVidia's GPGPU programming language CUDA
SIMD	SSE	Any x86 or x87 compatible architecture featuring Streaming SIMD Extensions (SSE, SSE2, SSE3)
	AVEC	PowerPC architectures containing AltiVec Units
	CELL	Cell Broadband Engine (Requires IBM Cell SDK version 2.0 or higher.)

Table 5.1.: Supported profiles of `cgisc`

patterns. There are two different backend categories, one for GPUs of various generations and one for different SIMD CPUs. Each target architecture belongs to one of those categories. Table 5.1 summarizes the profiles currently integrated.

5.1.3. Vectors, Scalars and Discretes

Throughout this chapter analyses and optimizations on CGiS programs are explained in the context of SIMD code generation. We have to distinguish the following types of variables:

- A *vector* is a standard CGiS variable as used in kernels. It can have up to four[2] components and be of any CGiS base type.
- An *iterator* is a vector variable that is initialized with a stream element and passed to a kernel.
- A *CGiS scalar* is a variable of small vector data type (up to 4 components) that is passed to a CGiS program from the outside, thus is specified in the CONTROL section. It does not change its value during a forall-iteration over streams.
- A *CPU scalar* is a variable that can be held in a CPU register during SIMD execution such as loop counters of loops with statically known loop bounds.
- A *scalar* can be a CGiS scalar or a CPU scalar, depending on the context.
- A *discrete* is a vector with only one component, i. e. it could be kept in a scalar CPU register.

At first glance this differentiation seems odd as the common usage of the term "scalar" is a description of non-vector variables. In CGiS a scalar is a vector that does not change its value across stream iterations such as an iterator. Thus, scalar here is the same as "not part of a stream".

[2]Restricting the number of components for the basic vector data types stems from CGiS' GPU heritage.

The origin of this re-interpretation of scalar also stems from CGiS' GPU heritage. With small vectors as base types, there was no need to discern vectors with different numbers of components. For SIMD CPUs this plays a significant role as we will see in the following.

5.2. Frontend and Intermediate Representation

The main task of the frontend is to translate the CGiS source program into the intermediate representation (IR) of the compiler. To have a better understanding of the frontend's task, let us examine the IR first and then have a closer look at the frontend itself.

5.2.1. Intermediate Representation

The intermediate representation (IR) internal to the CGiS compiler is a set of C++ classes representing a control flow graph (CFG). The classes are only explained as far as needed to understand the remaining sections and get a rough overview. All classes that belong to the IR are prefixed with `Cir` for CGiS Intermediate Representation. Lists, maps and vectors are all implemented with the Standard Template Library[26] of C++.

The main class to which all others are ancillary is `CirProgram`. An instance thereof contains the options selected for the current compilation process, the underlying profile and administrates the list of procedures specified in the CODE section as well as the calls to some or all of those procedures from the CONTROL section. These calls are represented by a class called `CirForall`. It holds the call to the respective kernel as well as the mapping of the parameters.

The procedures are represented by a class called `CirFunction`. It contains all necessary information such as parameter list and mapping, the subgraph of the intra-procedural control flow as well as other properties. This subgraph is a directed graph of `CirBlocks`. Every procedure has got a unique entry node and a unique exit node; both are represented as special blocks.

The blocks consist of possibly empty lists of `CirOperations`. Depending on their kind, these operations have a possibly empty list of source and target operands of type `CirOperand`. The operands can be constants or variables, thus of type `CirConstant` or `CirSingular`.

To sum up, a control flow graph in CGiS' intermediate representation is described as follows. A program contains a list of functions and a ordered list of calls. The calls correspond to the ones placed inside forall-loop bodies in the CONTROL section of the CGiS program and are in the same order. The functions, as specified in the CODE section, contain directed graphs of basic blocks which have lists of operations attached. The operations then have their list of operands which can be either variables or constants.

Depending on the current stage of the compilation process, different classes inherited from those presented above are used. For representing a CGiS program after it has been

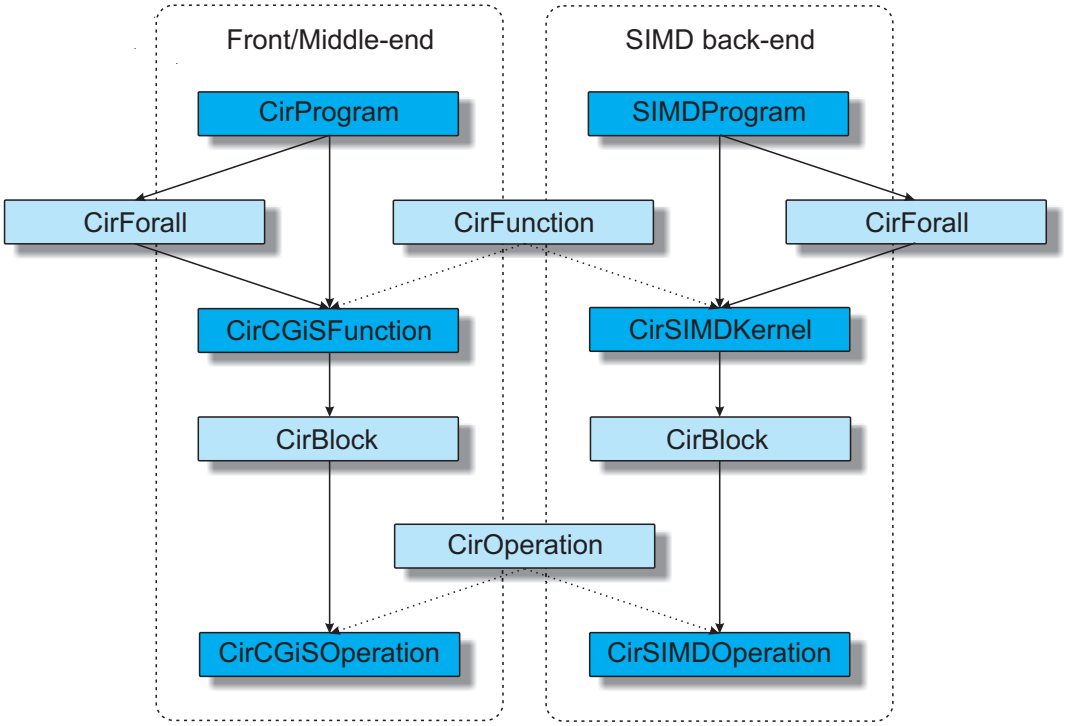


Figure 5.2.: Intermediate representation classes for middle-end and SIMD backend.

parsed, classes with the infix `CGiS` are used, e. g. `CirCGiSOperation` for operations as they are in the source code. When the compilation progresses and becomes target specific in the backend, the program, the functions, the operation etc. are translated into their target equivalents, i. e. `CirCGiSOperation` becomes `CirSIMDOperation`. The main classes in the different stages are shown in Figure 5.2 for the SIMD backend. Dotted lines denote inheritance while solid lines stand for superordinate classes.

5.2.2. Tasks of the Frontend

The order of the various tasks of the frontend of the CGiS compiler is displayed in Figure 5.3.

Parsing

Scanning and parsing is done with a standard flex/bison[31] combination. The parser builds the control flow graph of the CGiS program. Type checking is performed here to ensure correctness for the further stages. In case of implicit casting in the source code, appropriate casting operations are inserted.

During parsing, struct variables and nested structs are split into their components. This decomposition enables a uniform way of handling parameters and operations reducing them to the integrated short vector types. There are no pointers in CGiS so any assignment automatically passes the value. Regarding the intermediate representation,

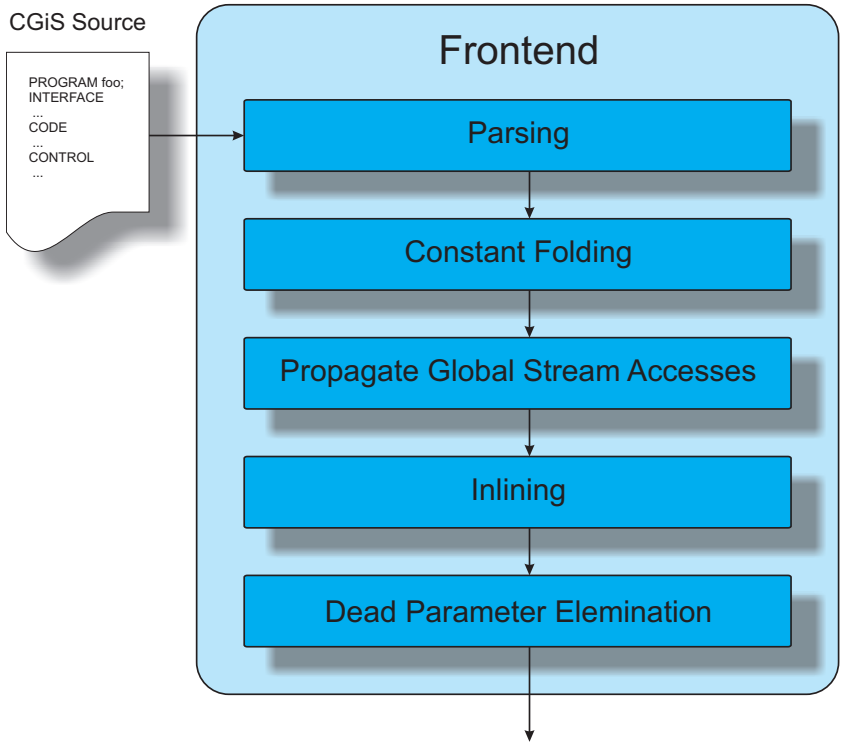


Figure 5.3.: Tasks of the frontend.

new variables are created, prefixed by the name of the struct variable, followed by a `$` and suffixed by the component name. This is done recursively to break down all nested structs.

In Program 5.1 the structs `a` and `b` in the assignment are broken down until only variables of the predefined types are left. The `$` splits the different levels within the newly created name for the variables leaving it still recognizable by a human reader.

Constant Folding

The frontend does a form of early constant folding. Operations on constants that can be statically evaluated are executed immediately and the resulting constant replaces the operation.

Constants used in SIMD code have to be loaded into SIMD registers at runtime. As the SIMD execution computes multiple operations in parallel, single constants are stored as quadruples, also meeting the alignment constraints. For example, if a loop variable is incremented by one every iteration, the constant array `{1, 1, 1, 1}` is stored at a 16-byte aligned address. Those constant arrays are declared as global data in the generated code, accessible to all kernels. So, all constants only occupy one array and multiple storing of the same constant can be avoided. In this way it is guaranteed that the memory footprint of constant data is as small as possible.

```
struct {
  float cx, cy;
} coord_t;

struct {
  coord_t coord;
  float value;
} point_t;

// let a and b be of type point_t
// the CGiS assignment
a = b;

// becomes the sequence
a$coord$cx = b$coord$cx;
a$coord$cy = b$coord$cy;
a$value = b$value;
```

Program 5.1: Decomposition of structs by the parser.

Propagation of Global Stream Accesses

This step was introduced with the writeback operation in CGiS programs (Section 4.4.4). Writebacks, in contrast to lookup and gather operations, do not require the accessed stream to be present as a kernel parameter. So this is the only case where streams can be written in kernels without the kernel locally knowing the stream.[3]

Thus, in this stage of the frontend, the writeback operations in the kernels are connected to the global stream data from the INTERFACE section they access.

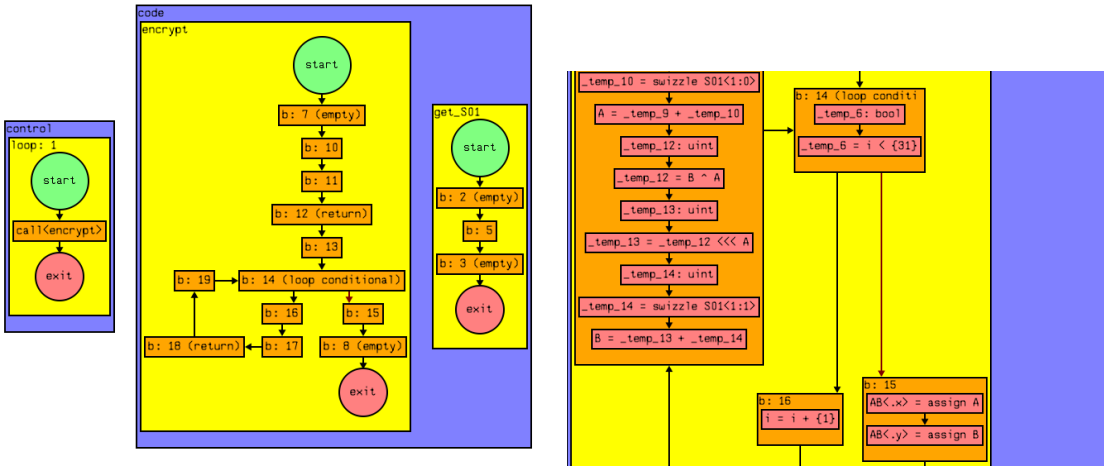
Inlining

Inlining is used in many compilers to increase the performance of the generated code. Instead of calling a procedure, its body is inserted at the call site. Passing of parameters and copying back of results are not necessary. When inlining, a compiler must ensure the program semantics are preserved. In some cases inlining can enable other optimizations as longer local code sequences can be addressed.

As mentioned, inlining can enable other optimization and this is also true for the SIMD backend. The later presented program transformation kernel flattening needs full inlining to be applied. Inlining also increases the performance of SIMD code as the passing of SIMD register contents via stack operations results in very expensive memory loads and stores. For some GPU backends inlining is mandatory[4], whereas it is optional for SIMD code generation.

[3] As this contradicts the concept of lookup and gather operation, writebacks are likely to change with the next version of the CGiS grammar.

[4] NV30 and NV40 generations.



(a) Control flow graph. (b) Unfolded basic blocks

Figure 5.4.: Visualization of the control flow graph and detailed instruction view.

Dead Parameter Elimination

In CGiS dead code elimination is the removal of operations that do not influence parameters passed to the outside of the program, i. e. all variables with an `out` flow specifier. In this early stage of the compiler, full dead code elimination is not reasonable as often dead code is introduced by other optimizations. The *dead parameter elimination* only removes parameters from procedure headers that do not contribute in any way to the computations of the `out` parameters. These parameters usually turn up when structs are passed to procedures and not all parts of the struct are read or written.

5.2.3. Visualization

The CGiS compiler is equipped with a GDL[50] interface. GDL, or Graph Description Language, is a textual representation of a graph tailored to be viewed with aiSee[17]. Having a graphical view on the control flow graph at any point during the compilation process is a very comfortable way to understand the functionality of transformations and optimizations. It also greatly eases troubleshooting.

Each stage of the compiler, as well as the results of the internal program analyses, can be visualized. Figure 5.4 shows the control flow graph of the rc5 encryption example, Program 6.2.[5] The encryption algorithm consists of one forall-loop with a call to the kernel `encrypt`. This kernel then calls `get_S01` to do lookup operations inside the encryption loop.

Figure 5.4(a) shows the view on the control flow graph right after parsing is done. The control section consists of exactly one call to the kernel `encrypt`. The CODE section contains two procedures `encrypt` and `get_S01`. Only the basic blocks and their connections are shown; all kernels have unique start and exit blocks. The blocks

[5]The option `-a P` passed to `cgisc` triggers the GDL output after parsing.

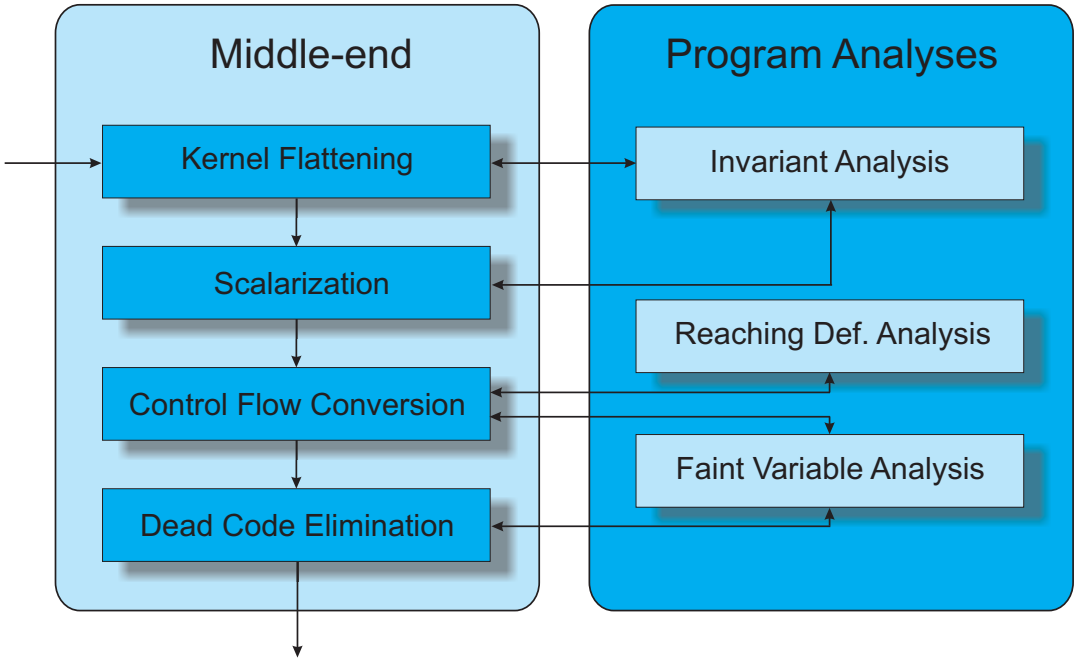


Figure 5.5.: Tasks of the middle-end.

containing instructions can be unfolded as seen in Figure 5.4(b). Now the single CGiS statements are visible.

5.3. Middle-end

The middle-end of the CGiS compiler performs transformations and optimizations on the control flow graph containing CGiS operations and thus is independent of the selected target architecture. However, not all transformations are reasonable for all backends. In Figure 5.5 the tasks of the middle-end for a target that is part of the SIMD backend are given. Some of the tasks require information about the program which can be gathered by program analyses. The analyses needed here are shown in the right part of Figure 5.5. The program analyses are generated from analysis specifications with the program analyzer generator PAG.

First, we will introduce a few basics of program analysis to clarify the descriptions of analyses later. Then the connection to the PAG framework will be described as well as the three generated analyses: invariant analysis, reaching definitions analysis and faint variable analysis. After that, the four tasks of the middle-end are explained: kernel flattening, scalarization, control flow conversion and dead code elimination.

5.3.1. Program Analysis Basics

The middle-end's job is to transform the source program as delivered by the frontend into a more efficient version without changing its semantics. If, for example, a certain

program point is not reachable, the code generation for that point can be omitted. To be able to do such optimizations, information on the source program is needed. This information can be derived by program analyses. These analyses are used to statically determine program properties which hold for every execution of the program. The CGIS compiler has a dedicated program analysis framework. Certain properties of programs which lead to optimization opportunities can be determined by data-flow analysis.

Data-flow analysis is based on the concept of abstract interpretation[9, 8]. In the context of computer programs, the idea is to select a set of properties of interest and reduce the concrete semantics of the program to abstract ones. These abstract semantics then describe only the effects on the properties of interest. In general, the computation of properties of a program is not possbile in finite time and space due to the Halting Problem[55]. By skillful abstractions the problem can be simplified making a solution computable at the price of lost precision. Let us illustrate this with a simple example: for an arbitrary program we are interested in the signs of the integer variables, not in their concrete value. So in our abstraction, there are only three possible abstract values left for a variable $(+, -, 0)$. The complexity of the operations is greatly reduced as, for example, for the multiplication. Only the new sign needs to be computed, e. g. for $-$ and $+$ the resulting sign is $-$, instead of actually multiplying two integer values and then examining the sign. The concrete semantics of the program are reduced to abstract semantics. Precision is lost in case of an addition, for a negative and a positive sign the resulting sign cannot be deduced.

An algorithm to determine program properties used in data-flow analysis is the MFP (Minimum FixPoint) algorithm. A program is represented by its control flow graph (CFG), nodes correspond to statements and edges to possible control transitions. Each has a transfer function attached describing how information is propagated along this edge. The CFG, along with the transfer functions, form a recursive equation system which can be solved iteratively by the MFP algorithm. To formalize this algorithm we need to introduce a few terms.

- A *Control Flow Graph (CFG)* is a directed graph $G = (N, E, s, e)$ with a finite set N of nodes and a set $E \subset N \times N$ of edges. $s \in N$ is called start node and $e \in N$ end node. If an edge (n, m) exists, then m is called *successor* of n and n is called *predecessor* of m. The start node s is required to have no predecessor and e no successor. Also the graph has to be connected, so every node must be reachable from s, and e is reachable from any node.

- A *path* p from node n_0 to node n_k in a $CFG(N, E, s, e)$ is a sequence of edges of the form $p = (n_0, n_1), (n_1, n_2), ..., (n_{k-1}, n_k)$. An edge can be contained in the path multiple times if loops are present in the program.

- A partially ordered set $(A, \sqsubseteq)$ is called a *complete lattice*, iff every subset of A has a *greatest lower bound* ($\bigsqcap$) and a *least upper bound* ($\bigsqcup$). The elements $\bot = \bigsqcap A, \top = \bigsqcup A$ are called bottom and top elements of A. The application of the greatest lower bound to two elements $\bigsqcap\{a, b\}$ is written as $a \sqcap b$. Analogously, $\bigsqcup\{a, b\}$ is written as $a \sqcup b$.

- An *ascending chain* $(x_i)_i$ is a sequence $x_0, x_1, ...$, such that $x_j \sqsubseteq x_{j+1}$ holds for all j. A chain $(x_i)_i$ is called *strongly ascending*, iff for all j, x_j and x_{j+1} are different. A chain eventually stabilizes iff there exists an index j such that for all indexes $n > j$, $x_j = x_n$ holds. Similar definitions apply to descending chains.

- Functions from a set A to a partially ordered set B are usually ordered by the pointwise ordering: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x) \forall x \in A$. Let A and B be complete lattices. A function $f : A \to B$ is called monotone, iff $\forall x, y \in A : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \in B$. The function f is called distributive, iff for $\forall x, y \in A : f(x \sqcup y) = f(x) \sqcup f(y)$. All distributive functions are monotone.

- A *data flow* problem is a CFG together with a complete lattice D of abstract values, a family of transfer functions, and a start value i in D. D is then called the *abstract domain*. The transfer functions express the abstract semantics of the CFG. They assign a meaning function to every edge of the CFG. $tf : E \to D \to D$, i. e. $\forall e \in E : tf(e) : D \to D$. The path semantics along some path is the composition of the transfer functions tf along the path (and the identity for the empty path):

$$\begin{aligned} [\![\epsilon]\!]_{tf} &= id \\ [\![e_0, \ldots, e_n]\!]_{tf} &= [\![e_1, \ldots, e_n]\!]_{tf} \circ tf(e_0) \end{aligned}$$

- The *Minimum Fixed Point (MFP) solution* is the least function in the pointwise ordering $MFP : N \to D$ satisfying the following system of equations:

$$\begin{aligned} MFP(s) &= i, \\ MFP(n) &= \bigsqcup \{tf(e)(MFP(m)) | e = (m, n) \in E\} \text{ for } n \neq s. \end{aligned}$$

The MFP solution is computable by a fixed point iteration if all transfer functions are monotone and every ascending chain of the lattice D eventually stabilizes. The algorithm consists of the inductive computation of a sequence $(F_k)_k$ of functions $F_k : N \to D$. The function F_0 maps the start node s to the start value i, and all other nodes n to the least element of the complete lattice D. The function F_{k+1} is computed from F_k as follows using the equation system for MFP:

$$\begin{aligned} F_{k+1}(s) &= i, \\ F_{k+1}(n) &= \bigsqcup \{tf(e)(F_k(m)) | e = (m, n) \in E\} \text{ for } n \neq s \end{aligned}$$

If the fixpoint, $F_{k+1} = F_k$ is reached, the algorithm stops. F_k is then the MFP solution. $F_k \sqsubseteq F_{k+1}$ is implied by the monotonicity of the transfer functions. Hence, $(F_k(n))_k$ is an ascending chain for every node n. If ascending chains in D eventually stabilize, the chain $(F_k)_k$ of functions eventually stabilizes as well, i. e. the algorithm terminates.

The program analyzer generator PAG[35] offers the possibility to generate a program analyzer using the MFP algorithm from a description of the abstract domain and the

abstract semantic functions[41]. These descriptions are usually given in two high level languages which support the descriptions of complex domains and semantic functions.

The domain can be constructed from simple sets such as integers by pre-defined operators (power sets etc.) or user-defined functions. The semantic functions are described in a functional language which combines high expressiveness with efficient implementation[36]. Additionally, a join function to combine two incoming values of the domain into a single one has to be written. This function is applied whenever a point in the program has two or more possible execution predecessors.

To use PAG generated analyses within `cgisc`, a frontend for PAG has to be provided by hand, `cirfront`. It implements routines to allow PAG to access the internal control flow graph and the connected data structures.

5.3.2. Program Analyses Interface: cirfront

PAG implements a work-list algorithm to propagate data-flow values along the edges of a control flow graph until a fixed point is reached. The control flow graph internal to `cgisc` is designed to be easily mappable to the one needed by PAG. A frontend for PAG to access the CFG has to

- ☐ provide access to a CFG as needed by PAG and means to traverse the graph.
- ☐ offer data structures and functions to access control flow graph items such as instructions, operands and their properties.
- ☐ syntacticaly integrate into the analysis specification language Optla.

`cirfront` fulfills these requirements by providing access to the control flow graph (CFG) and the abstract syntax tree (AST) of the operations. Functions are implemented to obtain and access parts of the syntax tree and traverse the CFG. The frontend is designed after the requirements for *Interface II* for PAG version 2.0[36]. This interface allows the specification of transfer functions for various points in the control flow graph, i. e. at call sites, at outgoing edges or inside blocks from statement to statement.

Overview

`cirfront` consists of the following files:

`iface.*, iface-inline*`	contain the functions described in the PAG manual such as stream operations and initialization
`iface2.cpp`	contains the access functions to the attributes declared in `pagoptions.optla`
`syn`	describes syntax trees under instructions
`pagoptions`	describes the features of syntactical lists
`pagoptions.set`	describes the possible types of nodes that can be matched in an optla file
`pagoptions.optla`	describes the attributes that can be used in an optla file
`context*`	special handling of different contexts in analyses
`testcfg.*`	implements some consistency checks for the PAG CFG and writes GDL output
`edges`	contains all edge types, that can be encountered
`gen_c_streams.pl`	generates wrapper for accessing CGiS internal structures as streams (`c_streams.[c|h]`).

Optla specification

The specification of a program analysis in the functional language Optla consists of three parts:

- A problem section: here the main properties of the analysis are specified, such as the name, the direction, the abstract domain and the initial values.
- A transfer section: all modifications of the data-flow value are given here. For different aspects of the CFG different transfer functions can be used, e. g. at edges between basic blocks, between instructions of a basic block or at routine starts.
- A support section: this part is reserved for the specification of additional helper functions, for example processing of lists of operands of instructions etc.

When specifying an analysis in Optla, several data structures are accessible at different points in a CFG. For example at a node in the CFG, the targets and the operands present at this node can be accessed, while at a block only its type can be accessed. The reserved names for these structures and their types as well as other predefined functions can be found in `pagoptions.optla`, presented in Figure 5.3.2. Each CFG item is introduced in capital letters and its accessible contents are listed below, e. g. a statement inside a basic block is named NODE. It contains an instruction, a list of targets, a list of operands and an unique ID. Variables as well as constant data is of type `KfgData` which maps to `CirCGiSOperand` in the context of `cgisc`.

```
GLOBAL                          NODE2_POSITION

graph:          Kfg#            block2:          KfgBlock#
                                instruction2:    KfgInstruction#
ROUTINE                         position2:       snum#
                                op_id2:          snum#
routine:        KfgRoutine#     targets2:       *KfgData#
parameters:    *KfgData#        operands2:      *KfgData#

BLOCK
                                EDGE_POSITION
block:          KfgBlock#
node_type:      KfgNodeType#    edge:            KfgEdge#
                                edge_position:   snum#

NODE
                                PROBLEM CirFrontend
instruction:    KfgInstruction#
targets:       *KfgData#
operands:      *KfgData#        SUPPORT
op_id:          snum#
                                exists     :: KfgData -> bool;
                                operand_id :: KfgData -> unum;
POSITION                        is_variable:: KfgData -> bool;

position:       snum#
```

Figure 5.6.: The file `pagoptions.optla` defines the reserved names and types for accesses in the CFG and the predefined functions.

Syntactical lists can be used in Optla in the same way as in ML[44], for example. `cirfront` implements these lists for accessing operands and targets of an operation as well as function parameters. In Optla specifications, these lists can be iterated with the common list access operations such as `[h::t]`. The implementation of syntactical lists greatly improves the readability of the analysis specification. The following code snippet from a CGiS analysis specification shows the use of these lists:

```
add_params :: varset, *KfgData -> varset;
add_params (flow, [!]) = flow;
add_params (flow, h::t) =
  if (!is_external_scalar(h)) then
    add_params (flow ~ operand_id(h), t)
  else
    add_params (flow, t)
  endif;
```

The function `add_params` is part of the support section of an analysis and takes a set of operand IDs and a list of CFG items as parameters and returns a new set of operand IDs. Its purpose is to add the parameters of a kernel to a given set of variables. The ~ operator corresponds to a set addition in Optla. Thus, all elements of the incoming parameter list that are not external scalars are added to the given set `flow`.

In the transfer section of the analysis specification, the function to add parameters can be used at the start of each routine:

```
TRANSFER_EDGE
_, _, _:
    if node_type = node_start then
        let vars <= @ ; in
            lift(add_params(vars, parameters))
    else
        @
    endif;
```

This part of a transfer section shows the modification of the data-flow value at certain edges. The user can specify at which types of edges the modification should be applied via pattern matching. As only wildcards (_) are used for the edge types, here the modification can take place at every combination of edge types. `node_type` refers to a property of the start node of the edge. Without going into further detail now the presented transfer function matches every edge that starts at a routine start node and adds all parameters found at that node to the data-flow value passed along the edge. More information can be found in [36].

5.3.3. Implemented Program Analyses

To generate a program analysis with PAG two specifications are needed: a domain specification and a specification of the abstract transfer functions. In the following we will explain the analyses that are part of the CGiS compiler's middle-end from a high-level point of view. We focus only on the description of the abstract domain and the transfer functions. To explain the implemented analysis, their data-flow values and their transfer functions better, we introduce the following notations:

Let k be a kernel called from the CONTROL section and V the set of variables used in the kernel. P_i are the parameters with `in` flow specifier and P_o are the ones with `out` flow specifier. Parameters with the `inout` specifier are in both sets. N is the set of all program points. For any assignment s at program point $p \in N$ the set of target variables is denoted T_s and the set of operands is denoted O_s. The transfer function at statements θ maps data-flow values and program points to data-flow values, thus applying the changes the statement at a program point makes on the examined data. *Dom* is the concrete domain of the underlying analysis. For all the analyses presented here the abstract domain is created by lifting the concrete domain to a lattice with $\top$ and $\bot$.

$$\theta : (Dom, N) \rightarrow Dom$$

For each of the following analyses the specification of the data-flow value and the transfer functions will be given. As a running example consider Program 5.2. With this program we will show the different program properties determined by our analyses.

```
PROGRAM analysis_props;
INTERFACE
extern in int3 X<NUM>;
extern out int2 Y<NUM>;

CODE
procedure properties (in int3 a, out int2 b)
{
  int3 c = 2 * a;          // (0)
  int i = 0;               // (1) i is invariant
  for (; i < 10; i++) {    // (2)
    int d = i;             // (3)
    c += [1,1,1];          // (4) c reaches this from (0) and (4)
  }
  b = c.xy;                // (5)
  c = [0,0];               // (6) c is faint
}

CONTROL
forall (x in X, y in Y)
  properties (x, y);
```

Program 5.2: Analysis-determined variable properties.

Invariant Analysis

The aim of the *invariant analysis* is to mark variables that firstly do not directly contribute to the computations of variables that are passed to the outside, i. e. have the flow specifier `out` and secondly do not get modified by non-scalar input variables. The latter demanded property states that the value of the marked variable does not differ for different stream elements as the forall-iteration progresses. The values of those variables are then invariant regarding the current forall-iteration.

As an example examine Program 5.2. The variable `i`, defined at program point (1), is used as a loop counter with a static loop bound. No matter what tuple of `(a,b)` is passed to the kernel `properties`, `i` changes its value in the same way. Invariance is a property that is kernel-wide and only holds for variables in kernels called from the CONTROL section of the CGiS program.

The main motivation behind the invariant analysis is to keep loop counters in general purpose registers of the CPU, i. e. not in SIMD registers. This enables significantly cheaper termination checks and increments. To determine the invariance property for variables of a procedure, two program analyses have to be performed. The first one is a forward analysis that starts with a set including the function parameters with the flow property `in`, but excluding scalar variables as they do not change across the forall iterations as well. Every variable whose value depends on a variable in the set also gets included in the set. Remember that the targets at statement s are denoted T_s, the operands O_s, kernel parameters with in-flow P_i and with out-flow P_o. The initial value of the carrier is

$$CF_{init} = P_i$$

and the transfer function is

$$\theta(CF, p) = \begin{cases} CF \cup T_s & \text{if } CF \cap O_s \neq \emptyset \\ CF & \text{otherwise} \end{cases}$$

The second analysis runs backwards over the control flow graph and starts with the set of function parameters that have the `out` flow specifier. Each variable that is an operand of an assignment to a variable in the set gets inserted as well. The initial value of the carrier is

$$CB_{init} = P_o$$

and the transfer function is

$$\theta(CB, p) = \begin{cases} CB \cup O_s & \text{if } CF \cap T_s \neq \emptyset \\ CB & \text{otherwise} \end{cases}$$

As we look for invariance which is a kernel-wide property and not restricted to single program points, we need to combine the resulting sets of the analysis via set union. For the forward analysis this is the set at the end node and for the backward analysis it is the one at the start node. Each variable not contained in the set then is invariant regarding forall-loop iterations. The set of invariant variables for kernel k then is

$$I_k = V \setminus (CF_k \cup CB_k)$$

A variable $v \in V$ is called *invariant* if $v \in I_k$.

Regarding the PAG specification the carrier in both cases is a lattice of integers which can be mapped to variables of the analyzed kernel. The combine function applied at control flow joins is a set union for both analyses.

Reaching Definitions Analysis

The *reaching definitions analysis* determines for each program point which assignments may have been made without being overwritten, when program execution reaches that point. An assignment reaches a program point p if p is reachable from the program point p' where the assignment is made. No further assignment to the same variable may be performed later on the path from p' to p. The results of a reaching definitions analysis can be used for constant folding and provide essential knowledge for control flow conversion.

In Program 5.2 two definitions of the variable c reach program point (4), one from the definition (0) and through loop iteration from itself (4). The carrier is a lattice over a set of integer tuples. The first integer of each tuple is the id of the assigned variable and the second is the id of the program point at which the assignment was performed. The analysis inserts at each program point p a tuple of the variable's id and the program point into the set for each written variable. This new definition then overwrites all definitions already in the set for the variable v. The combine function is also a set union. The initial value of the carrier is the empty set.

$$RD_{init} = \emptyset$$

and the transfer function is

$$\theta(RD, p) = \begin{cases} (RD \setminus (v, p')) \cup (v, p) & \text{if } v \in T_s \\ RD & \text{otherwise} \end{cases}$$

A variable $v \in V$ reaches program point p if $v \in RD_p$.

Faint Variable Analysis

The *faint variable analysis* computes for each program point p if the value of a variable is not needed at any program point reachable from p. If the variable is not faint, it is called *alive*.

Assignments to dead variables can be excluded from the program code, since their removal does not change the semantics of the program. If such an assignment has been removed from the program, the faint variable analysis has to be rerun because other variables might only have been used in that assignment. As an example, reconsider Program 5.2 in (6) the assignment to c is obsolete as c is never used again. Thus, c is faint at (6).

The domain of the faint variable analysis is a lattice over a set of unsigned integers. The analysis is a backward analysis and the data flow information at all program points but the end node is initialized with $\perp$. At the end node we add all the parameters with out-flow into the carrier as these are used outside of the CGiS program. The analysis proceeds by adding variables that are needed to compute the final values of these parameters or intermediate results. Thus, for each operation, the analysis checks whether the operation's targets are contained in the carrier. If so, the targets are removed

and the operands are added. Or in other words: if all targets of an operation are faint, none of the operands is marked as alive. Otherwise, all targets of the operation are marked as faint and all operands are marked as alive. The initial value of the carrier is

$$F_{init} = P_o$$

and the transfer function is

$$\theta(F,p) \quad = \begin{cases} (F \setminus \{(t,p) : t \in T_s\}) \cup \{(o,p) : o \in O_s\} & F \cap T_s \neq \emptyset \\ F & \text{otherwise} \end{cases}$$

A variable $v \in V$ is called *faint* at program point p if $v \notin F_p$.

5.3.4. Kernel Flattening

The main challenges in generating efficient SIMD code are how to arrange data and how to meet the alignment requirements of data accesses. Our solution to this alignment and data layout problem is *kernel flattening*.

Kernel flattening is a code transformation on the intermediate representation. It processes a single kernel and splits all stream elements and variables into one-component vectors or *discretes*. This also includes operations on those variables: every operation is copied and executed on each former component of the variable. Program 5.3 shows the flattening operations applied to a CGiS procedure `yuv2rgb`, transforming YUV color values into RGB values. Bear in mind that operands of a multiplication with a discrete value are replicated to match the number of components of the target or operation. Here for the computation of `RGB`, each of the components of `YUV` is accessed individually but gets replicated to a 3-component vector to match the number of the sums' components. The function `yuv2rgb_f` is the resulting function. The parameters `YUV` and `RGB` and the constant vectors are split into three discretes each. The assignment to `RGB` and the computations are split as well. So each component becomes a single discrete or parameter and all vector operations are replaced by discrete ones.

The procedure transformed by kernel flattening can be executed in parallel. After compound variables have been broken down to discrete ones, these can be subjected to SIMD vectorization. Four consecutive elements of each one-component vector stream can now be loaded into one vector register, and immediate constants are replicated into a vector. Because the original data elements of the stream possibly have small vector data types, data either has to be reordered during execution or beforehand. The SIMD backend supports local and global data reordering. While global reordering is basically a reordering in memory, local reordering inserts code that reorders these elements in registers at the beginning of the function and at the end. For the previous example the possible stream access patterns are shown in Figure 5.7. Sequential execution accesses one YUV-triple per iteration. Global reordering splits the YUV-stream into three streams. Thus, in each iteration, four elements of each former component can be loaded into a vector register and processed. Local reordering takes the stream as it is and inserts permutation operations at the start and the end of the flattened procedure.

```
procedure yuv2rgb(in float3 YUV, out float3 RGB)
{
  RGB = YUV.x + [ 0, 0.344, 1.77 ] * YUV.y
              + [ 1.403, 0.714, 0] * YUV.z;
}

procedure yuv2rgb_f (
              in  float YUV_x, in  float YUV_y, in  float YUV_z,
              out float RGB_x, out float RGB_y, out float RGB_z)
{
  float cy = 0.344, cz = 1.77, dx = 1.403, dy = 0.714;
  RGB_x = YUV_x +              dx * YUV_z;
  RGB_y = YUV_x + cy * YUV_y + dy * YUV_z;
  RGB_z = YUV_x + cz * YUV_y;
}
```

Program 5.3: Kernel flattening applied to the procedure `yuv2rgb`.

Per default, the CGiS backend uses local reordering, but the programmer can force global reordering by hints. Global data reordering requires input stream data to be loaded before and output stream data to be stored after execution. Thus, the higher reordering costs with respect to local reordering are amortized only if the reordered stream is processed several times with gathers and lookups. Data reordering is further discussed in Section 5.5.2.

Lookup and gather operations are split as well. In case of global reordering, the gathers and lookups are straightforward because only alignment has to be taken care of. As for local reordering, on gather operations possibly more data has to be loaded than just the bytes of the required stream element. This is because the data can be spread across several SIMD blocks[6] and a SIMD load instruction always loads full blocks. The number of blocks loaded and reorganized can be kept minimal though because all the offsets of the elements gathered around the current are statically known. On the other hand, data-dependent lookups result in four different non-SIMD loads and the reconstruction of a vector. With too many of these lookups the benefit of vectorizing the function might be negated.

As another example consider the the rc5 encryption in Program 6.2 on page 110. Here, the application of kernel flattening means that the `inout` parameter `AB` is split into an `inout` parameter `AB_x` and an `inout` parameter `AB_y`. All operations are made discrete enabling SLP execution. Data reordering instructions are inserted allowing stride-one access to `AB_x` and `AB_y`. From the perspective of data layout and alignment, four elements can be processed in parallel.

[6]These blocks are 16 byte for all the examined architectures.

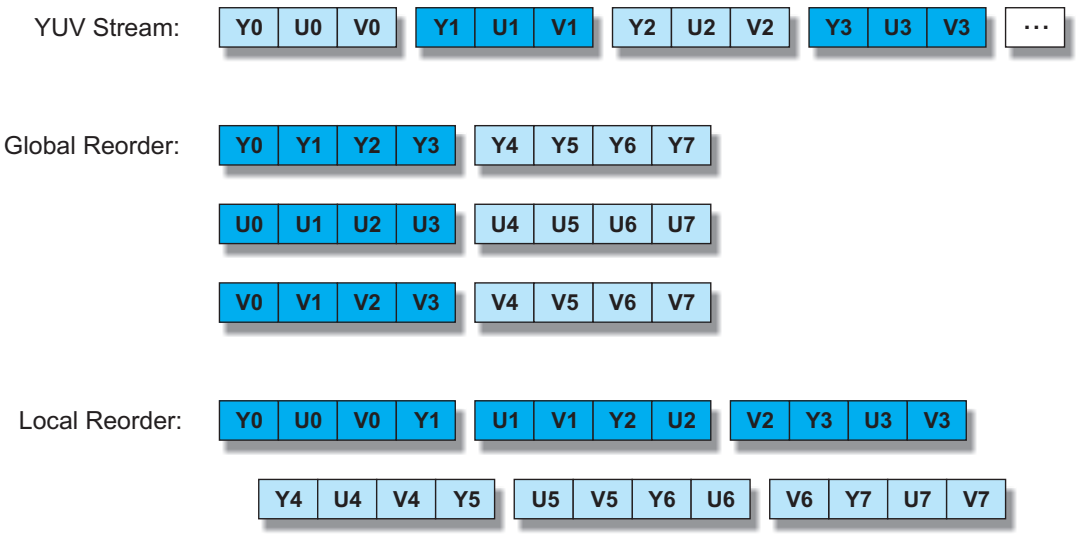


Figure 5.7.: Stream access patterns for global and local reordering of the YUV stream.

5.3.5. Scalarization

The optimization called *scalarization* tries to keep certain variables in scalar registers, i. e. they should not be held in SIMD registers later on in the translation process. On first glance this might sound contrary to SIMDization but is, in fact, a way to speed up SIMD code.

Many algorithms feature loops with statically known maximum loop bounds or loops whose maximum iteration count depends on a scalar variable passed to the kernel containing the loop. Let us assume now that this kernel is called from the CONTROL section. This means there is at least one stream which is iterated over and this kernel is called for each element of this stream. The aforementioned invariant analysis determines which variables are not influenced by any other source than scalar parameters or constants. Should this loop counter now be invariant for every invocation of the kernel, the loop body is always executed the same amount of times. With that knowledge it is possible to have a scalar loop counter with simple loop bound check even for parallel execution of the kernel. Basically all variables that are invariant regarding the forall-iterations are candidates for scalarization.

The reason why loop counters are especially interesting is because the boundary checks can be very expensive depending on the target architecture. The Streaming SIMD Extensions feature an instruction that translates the result of a SIMD comparison into an intermediate value, `MOVMSKPS`. For AltiVec no such conversion is possible, however, in some occasions the status register can be queried for an "all zero"-flag. This flag indicates that on a vector compare, all comparisons resulted in a false.

To clarify the idea consider Program 5.4. The integer variable `steps` is a scalar parameter to the kernel `apply`. `steps` is constant across all calls to this kernel. Thus the for-loop in `apply` gets executed exactly the same time for every `(v,f)` tuple, i. e. stream element. This means that the loop bound check can be performed with a scalar

```
PROGRAM scalarify;
INTERFACE
extern inout float velocities[_];
extern in float forces[_];
extern in int steps;

CODE
procedure apply (inout float vel, in float force in int steps)
{
  dt = 0.1;
  for (int iter=0; iter < steps; iter++) {
    vel += force * dt;
  }
}

CONTROL
forall (v in velocities, f in forces)
  apply (v, f, steps);
```

Program 5.4: Scalarization of the loop counter `iter`.

loop counter.

For another scalarization example let us revisit Program 6.2. The lookup function `get_S01` only depends on the scalar `i`. With the invariant analysis introduced on page 75 it can be determined that `i` is constant across all elements processed in parallel. CGiS also allows the user to annotate invariant variables to guide the compiler. Each of the elements processed in parallel calls the lookup procedure with the same integer parameters (`where`) in the same order. This enables flattened parallel execution of the kernel. The value to be looked up can be provided by the lookup in `get_S01` with only one SIMD load and one or two permutations.

5.3.6. Control Flow Conversion

The three main control flow constructs of CGiS are procedure calls, conditionals and loops. The control flow conversion tries to modify and replace these constructs to allow SIMDification of diverging control flow. All transformations of the control flow conversion are executed on the intermediate representation of the CGiS program.

By default, calls are fully inlined for SIMD targets although it is possible to force separate functions by using compiler hints[7]. We found that generating true calls increases the runtime of the application. At the call site values to be passed as parameters are usually pushed on the stack and loaded back after the called routine returns. Because of the size of the registers this can be very expensive.

The motivation behind the application of control flow conversion is to be able to

[7]This can restrict the application of other optimization.

execute the bodies of if-then-else statements and loop statements in parallel. In the context of CGiS this means executing kernels called from the CONTROL section in parallel, i. e. for successive stream elements iterated by the forall loop. Control flow conversion for CGiS programs includes if-conversion and loop-conversion.

The transformation called *if-conversion* is a way to convert control-dependencies into data-dependencies and is common to vectorizing compilers[64]. The basic idea of traditional if-conversion is to attach different guards to all assignments in the true and the false branch of the if-construct to be converted. The control flow change by the if-construct can then be removed. Consider the code snippet in Figure 5.8.

```
if(c) {
  x = a;
else {
  x = b;
}
```

(a) if-construct

```
g0 = c;
[g0] x = a;
g1 = !g0;
[g1] x = b;
```

(b) guarded assignments

Figure 5.8.: Replacement of an if-control-structure by guarded assignments.

The if-construct on the left is converted into the guarded assignments on the right side. These assignments to `x` are executed depending on the guards but the control does not change and reaches them both.

On the SIMD hardware examined here, the setting is slightly different as guarded execution of statements is not supported. To be able to remove the control flow change another approach must be taken: a mask is generated from the condition. All assigments in the conditional branches are executed on copies of the original variables and select operations at the end of the branches are used to write the right value back depending on the mask. Here, the masks are the results of vector compare operations. These component-wise operations yield a vector that contains all 0s at an element if the comparison failed for that element, all 1s otherwise. The Allen-Kennedy algorithm of [64] has to be adapted in the following way to match the target hardware addressed by the SIMD backend.

As depicted in Figure 5.9, let I be a basic block containing an if-statement with condition C_I and its associated mask M_I. For simplicity, we consider only a simple conditional body, with one block T_I in the `true`-branch and one block F_I in the `false`-branch. The control flow join is denoted J_I. Let L_I be the set of variables alive at J_I, W_T the set of variables written in T_I and W_F is the set of variables written in F_I. The algorithm which inserts the additional operations required for the if-conversion is given in pseudo-code in Figure 5.10.

During the if-conversion phase, for each I the sets S_T and S_F are determined. For each control flow branch, copies of the variables written and live after the branch are inserted at the beginning of the respective branch. After the end of a branch, select instructions (like ϕ-functions from SSA [39]) are inserted which select the new value for the written variable depending on the generated mask. For nested control flow structures the masks have to be combined and evaluated appropriately.

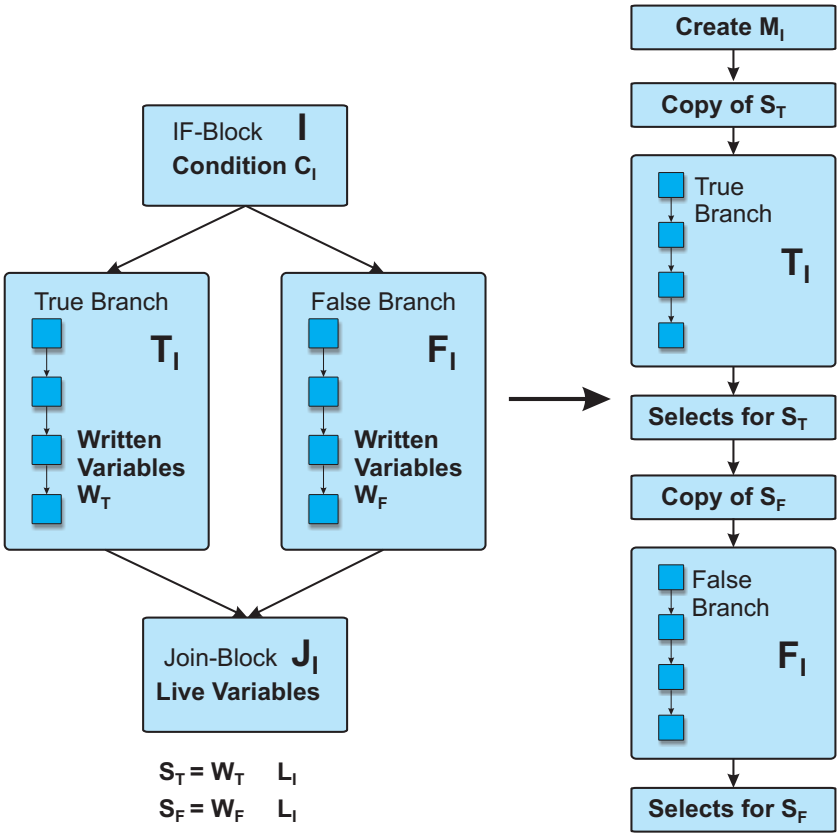


Figure 5.9.: Basic if-conversion in `cgisc`.

The control flow conversion of loops is quite straight-forward. For the loop condition, a mask is generated as well, and the loop iterates as long as the mask is not completely 0 (signifying that *all* elements of the tuple have finished iteration). If the mask is completely 0, then the loop can be exited.

Conversion of nested control flow statements is also supported. When the mask of a condition is generated for a nested statement, it is always combined with the mask of the control flow statement via binary `and`.

The loop modification statements `break` and `continue` are also part of the CGiS language and need to be supported by the control flow conversion. If one of these statements is encountered, there must be a preceeding if-statement (see Section 4.4), which is associated with an if-mask. That if-mask, the masks for all enclosing if-statements within the same loop, and the loop-mask are combined via binary and-not.[8] In case of a break statement, an additional loop control mask needs to be introduced to identify the disrupted loop. The following algorithms do not modify any conditions, they *only create or modify masks*.

Let i be a nesting level with an associated mask $\mathcal{M}_i$. Assume a loop statement L at nesting level l and a conditional at level n with $n > l$ containing a continue statement

[8]The binary operation andn(a,b) is equal to $a \wedge \neg b$.

```
Use the faint variables analysis to determine L_I
Use the reaching definitions analysis to determine W_T and W_F
Build intersections S_T = W_T ∩ L_I and S_F = W_F ∩ L_I
Foreach v_T ∈ S_T
   insert v_T' = v_T at the beginning of T_I
   insert select(v_T, v_T', M_I) at the end of T_I
Foreach v_F ∈ S_F
   insert v_F' = v_F at the beginning of F_I
   insert select(v_F', v_F, M_I) at the end of F_I
```

Figure 5.10.: Pseudo code for additional insertion of copies and select operations used in if-conversion.

associated with the loop statement at level l, i. e. there are no other loop statements in between. The pseudo-code in Figure 5.11 describes how masks are updated for continue statements. Any code following a `continue` at the same nesting level has already been removed by the parser, as it is unreachable anyway.

```
Control flow transformation reaches continue at nesting level n
M_l = {M_i : i ∈ [l..n − 1]}
If continue is in else−branch
   insert M_c = not(M_n)
Else
   insert M_c = M_n
Foreach M ∈ M_l
   insert M = andn(M, M_c)
```

Figure 5.11.: Pseudo code for conversion of continue.

The conversion of `break` follows the same algorithm as for `continue` but additional information needs to be passed to the update of the loop mask as computations in further iterations are aborted for all elements affected by the break statement. Basically, an additional mask is introduced that remembers which elements need to stay deactivated. In Figure 5.12 the conversion for `break` is shown in pseudo-code. Each loop only needs one control flow mask and it is updated every time a break is reached.

Keep in mind that a mask associated with a break or a continue statement is a combination of the masks of the upper nesting levels via binary `and`. Thus, it can never happen that an update of the loop mask with the break-mask disables elements for which the iteration should continue.

To illustrate the control flow conversion, consider Program 5.5. We can see a while-loop at nesting level 0 and two nested if-statements. The loop itself is not removed by the conversion but its condition is changed as the eval function only returns false if the condition `C0` is false for all elements executed in parallel. The function `get_mask`

```
Control flow transformation reaches break at nesting level n
M_l = {M_i : i ∈ [l..n − 1]}
insert intialization of M_b = 0 before L
insert M_l = andn(M_l, M_b) after the creation of M_l
If break is in else−branch
  insert M_b = or(M_b, not(M_n))
Else
  insert M_b = or(M_b, M_n)
Foreach M ∈ M_l
  insert M = andn(M, M_b)
```

Figure 5.12.: Pseudo code for conversion of break.

creates masks from conditions. At the location of each previous if-statement, a mask is created and combined with the mask of its upper nesting level. As a break statement is present, the loop gets an additional break-mask, `bmask`. It remembers for which elements the iteration has been terminated, and modifies the mask of the loop right after its creation. At the location of the break statement, all masks of the upper nesting levels are updated. For the condition `C0` controlling the loop iterations, the mask `m0` is created. As it is created in each iteration it needs to be updated with the break mask immediately. After that the first if-statement is converted by creating a mask `m1` for condition `C1` and a copy for `x` because it is written inside the if-branch and still alive after the if-branch. The same is now done for condition `C2`. `m2` is created and `x` is copied again. When the break is encountered, the break-mask `bmask` is updated, as are all the masks of the nesting levels above up to `m0`. The if-branch for condition `C2` ends now and the select statement is inserted for `x` and its copy `x2`. After the unchanged assignment to `x` the if-branch for `C2` closes and the select for `x` and its copy `x1` is inserted. It is visible that none of the assignments in the original program needed to be changed.

5.3.7. Dead Code Elimination

The dead code elimination optimization tries to remove statements from the control flow graph that do not contribute to the results passed to the outside. There are two possibilities as to how such dead statements can occur: they are introduced by the programmer or they result from preceding optimizations.

As an example, for dead code consider Program 5.6. The only variable carrying results is `b` which is of type `float2`. `c`, on the other hand, is of type `float3` and holds the result of a 3-component multiplication of `a` and a constant. The assignment to the third component of `c` is dead as the value of `c.z` is not used for the computation of any variable with the flow specifier `out`.

`cgisc`'s faint variable analysis now determines after kernel flattening that `c_z` is faint and that the assignment to it is obsolete. Thus it can be removed by dead code elimination. The dead statement is introduced by kernel flattening. Of course it is

```
// Cx are conditions in a CGiS program
while (C0) {
  if (C1) {
    if (C2) {
      x = 0;
      break;
    }
    x += 1;
  }
}

// after control flow conversion
bmask = get_false_mask();
while (eval (C0)) {
  m0 = get_mask (C0);
  m0 = andn (m0, bmask);
  m1 = m0 & get_mask (C1);
  x1 = x;                      // copy of x for C1
  m2 = m1 & get_mask (C2);
  x2 = x;                      // copy of x for C2
  x = 0;
  bmask |= m2;                 // break mask update
  m0 = andn (m0, bmask);
  m1 = andn (m1, bmask);
  x = select (x, x2, m2);      // select for C2
  x += 1;
  x = select (x, x1, m1);      // select for C1
}
```

Program 5.5: Control flow conversion example.

present in the source code as well, but if a short vector variable is considered an entity, the multiplication of 2 with the vector `a` is not dead.

The results of the dead code elimination can be visualized using the GDL interface. The results of the faint variable analysis and the dead code elimination for the procedure `dead` are given in Figure 5.13. The dead assignment is marked and an additional unfoldable info box displays the lifeness information on the variables known at the program point.

```
procedure dead_code(in float3 a, out float2 b)
{
  float3 c = 2 * a;
  b = c.xy;
}

// after flattening:
procedure dead_code_f(in float a_x, in float a_y, in float a_z,
                      out float b_x,  out float b_y)
{
  float c_x = 2 * a_x;
  float c_y = 2 * a_y;
  float c_z = 2 * a_z;
  b_x = c_x;
  b_y = c_y;
}
```

Program 5.6: Dead assignment to c.z.

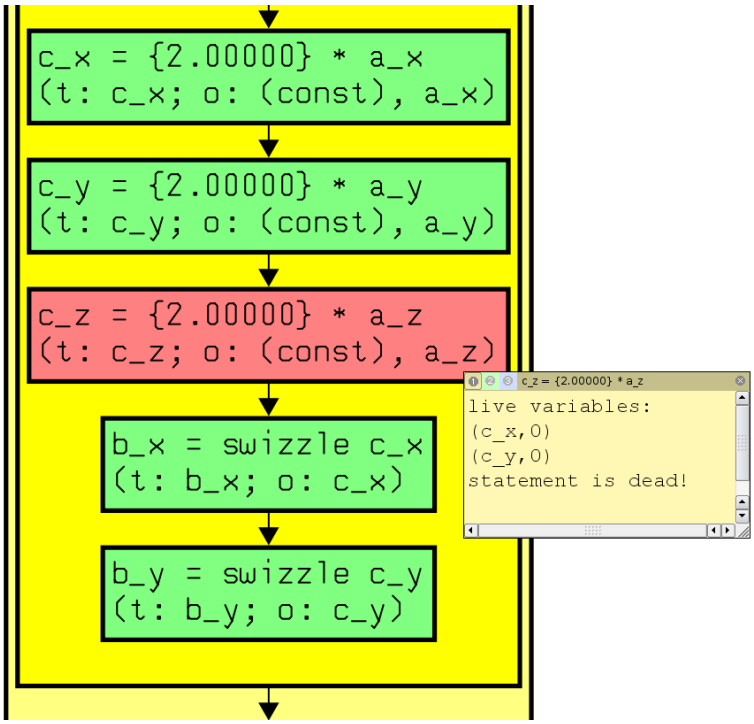


Figure 5.13.: Results of the dead code elimination.

5.4. GPU Backend

The GPU backend of the CGiS compiler has been developed by Philipp Lucas. In contrast to SIMD CPUs Graphics Processing Units offer many parallel computing units but are not as easily accessible. While the focus of the SIMD backend lies on mapping parallelism to the hardware and meeting the data alignment requirements, the main tasks to be accomplished for successful GPU code generation differ.

Recent graphics processing chips from NVidia, e. g. the G80 series, offer many CPU features such as loops and procedure stacks for calling subroutines yet still no parameter stacks. But the life of a GPGPU (General Purpose Programming on GPUs) programmer has not always been that comfortable. When we started out with the implementation of the CGiS compiler the GPU targets were the NV30 and NV40 series. By that time GPUs were programmed via pixel shaders, small assembly programs uploaded onto the GPU by a graphics API like OpenGL or DirectX. The CGiS compiler uses these pixel shaders as output format.

The frontend part of the CGiS compiler is shared by both backend categories. Middle-end optimizations are only partly applied and the code generation in the backends themselves then is handled in a completely independent way. The most prominent objectives of the GPU backend are:

- ☐ Packing data into textures. As there was neither an abstraction for GPUs nor the memory of GPUs a graphics API was used to operate the memory. All the data has to be organized in textures, which are small-sized chunks of GPU memory accessible by texture operations.

- ☐ Register allocation. The pixel shader programs are on assembly level, thus the writer himself or the compiler has to take care of allocating registers for operations.

- ☐ Optimizations. The kind of optimizations differ for the GPU backend. Copy elimination[39], the process of removing superfluous assignments, and Superword Level Parallelism are two optimizations only implemented in the GPU backend. The SIMD backend relies on kernel flattening and the optimizations of the C/C++ compiler.

NVidia opened up their hardware for general purpose programming with their in-house language CUDA[42]. CUDA lets the programmer write kernels in a C-like language with certain restrictions. The GPU backend of `cgisc` supports code generation for NV30, NV40 and G80. A CUDA backend has also been integrated for testing and comparison purposes.

5.5. SIMD Backend

It is the backend of a compiler that finally translates the optimized intermediate representation into target specific code. For `cgisc` there are two different backend categories,

SIMD and GPU. The SIMD backend of the CGiS compiler comprises the following targets:

- ☐ Intel's Streaming SIMD Extensions
- ☐ Freescale's AltiVec
- ☐ IBM's Cell Broadband Engine

The tasks of the SIMD backend can be seen in Figure 5.14. Its input is the control flow graph modified by the middle-end optimizations. First, the code generation is initiated by matching sequences of statements of the IR with instruction sequences of the target architecture. This also includes peephole optimizations. As SIMD hardware has to meet data-alignment requirements, and previous transformations such as kernel flattening may require the data to be reordered a data reordering phase follows the code generation. Depending on the architecture and the used data accesses, loop sectioning – an adaptation of the iteration sequence – might be applied to increase data cache performance. For the Cell BE thread management features must be included in the generated code: the PPE functions as distributor and assigns threads to the Synergistic Processing Elements. This is done in an additional backend phase that is only active when code for the Cell BE is to be generated.

5.5.1. Code Selection and Peephole Optimizations

As in the GPU backend the code selection in the SIMD backend is also handled by pattern matching with OORS[15]. The generated pattern matcher matches sequences of CGiS instructions inside basic blocks in the intermediate representation and replaces those with instruction sequences of the selected target architecture. Those sequences are later written out in form of compiler intrinsics.

At this point a few peephole optimizations are possible, the most prominent of which is exploiting the multiply-add instruction of AltiVec. The instruction set does not feature a "normal" integer multiplication; multiplications always take x-bit arguments and produce $(2 \times x)$-bit results. For example, the `vec_mule` can multiply the even elements of two vectors containing 16-bit integers. Standard multiplication without changing precision is done via `vec_madd` a vector multiply-add operation. During the pattern matching phase of the code generation, the matcher looks for sequences of multiplications and additions to find a suitable pair to fill the needs for `vec_madd`.

Program 5.7 shows the OORS pattern that searches for opportunities to collapse a multiplication and an addition into a single operation. The search pattern looks for an addition of which one of the operands is the target of a former multiplication. The $\star$ allows an arbitrary number of instructions in between, not passing basic block boundaries. There are no restrictions to this rule so its condition is always true. OORS supports costs for rules to let the matcher favor some rules over others trying to minimize global or local costs depending on the selected option. The `replacement` field describes the resulting operation replacing the other two. Here the ternary AltiVec operation `vec_madd` is created including the translation of the original operands to SIMD ones.

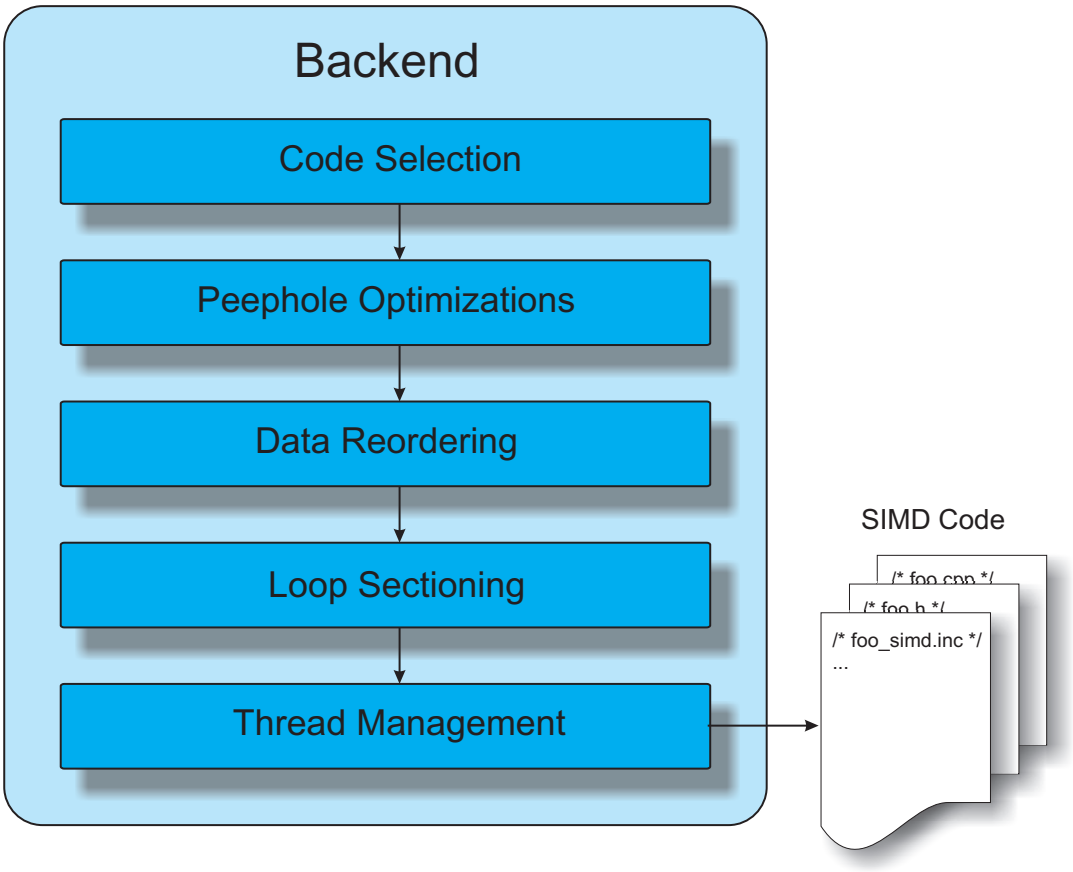


Figure 5.14.: Tasks of the SIMD backend.

Translation of Masks and Swizzles

In graphics hardware the accessing of single or multiple components in any order is possible, such as `a.xy = b.zw`. These operations are called masking and swizzling and are also reflected in CGiS.

The multi-media instruction sets examined in this thesis have slightly different operations to handle both operations. For the Streaming SIMD Extension there are three instructions that let the user select different components of two vectors to build a new vector: `shuffps` and `unpack[h|l]ps`. Both have been introduced in Section 2.3.1.

Single swizzles are easily realized with shuffles as both operands can be the same register. The following sample code is used to explain the translation of swizzles.

```
int4 a = [0,1,2,3];
int4 b = a.xzzx;    // (1)
a.x    = b.w;       // (2)
a.yz   = b.yw;      // (3)
```

The appropriate SSE intrinsics for statement (1) are:

```
rule madd
{
  search:  [ CirCGiSBinOp ($$->opcode() == OP_BIN_MUL &&
                           target_is_temp($$->target())),
             *,
             CirCGiSBinOp ($$->opcode() == OP_BIN_ADD &&
                           ($$->operand1() == $1->target() ||
                            $$->operand2() == $1->target()))
           ]
  condition: { return true; }
  cost:      { return 1; }
  replace: [ CirSIMDTerOp (OP_AVEC_MADD,
                 simdkernel($1),
                 simd_op($3->target(),   $1, 1),
                 simd_op($1->operand1(), $1, 1),
                 simd_op($1->operand2(), $1, 1),
                 simd_op(($3->operand1() == $1->target() ?
                          $3->operand2() : $3->operand1()),$1,1),
                 $1->mask())
           ]
}
```

Program 5.7: OORS rule for the MADD peephole optimization.

```
__m128i a = _mm_set_epi32 (0,1,2,3);
__m128i b = mm_shuffle_epi32 (a, a, _MM_SHUFFLE(0,2,2,0))
```

Hereby the intrinsic `_mm_set_epi32` creates an integer vector from its parameters and `_mm_shuffle_epi32` creates a new vector from selected components of its first two parameters. The macro `_MM_SHUFFLE` creates an appropriate intermediate constant for the component selection. In this example the selection corresponds to the swizzle in (1). Should the assignment contain masks as well, multiple shuffle operations can be needed to create the wanted vector as for statement (2) and (3):

```
__m128i tmp0, tmp1;

// statement (2)
//  build up tmp0 = [a.x a.x b.w b.w]
tmp0 = mm_shuffle_epi32 (a, b, _MM_SHUFFLE(0,0,3,3));

a = mm_shuffle_epi32 (tmp0, a, _MM_SHUFFLE(0,2,2,3))

// statement (3)
//  build up tmp0 = [a.x a.x b.y b.y]
tmp0 = mm_shuffle_epi32 (a, b, _MM_SHUFFLE(0,0,2,2));
```

```
//  build up tmp1 = [b.w b.w a.w a.w]
tmp1 = mm_shuffle_epi32 (a, b, _MM_SHUFFLE(3,3,3,3));

a = mm_shuffle_epi32 (tmp0, tmp1, _MM_SHUFFLE(0,2,0,2))
```

For AltiVec masks, swizzles can be effectively realized with the permute operation. Arbitrary elements can be selected from the two input vectors to create the new vector, `vec_perm` is presented in Section 2.3.2. The three statements can be simply realized with one permute operation each:

```
static vector signed int a = {0, 1, 2, 3};

// statement (1)
static const vector unsigned int perm1 =
       {0x00010203, 0x0c0d0e0f, 0x0c0d0e0f, 0x00010203};
vector signed int b = vec_perm (a, a, perm1);

// statement (2)
static const vector unsigned int perm2 =
       {0x1c1d1e1f, 0x04050607, 0x08090a0b, 0x0c0d0e0f};
a = vec_perm (a, b, perm2);

// statement (3)
static const vector unsigned int perm3 =
       {0x00010203, 0x14151617, 0x1c1d1e1f, 0x0c0d0e0f};
a = vec_perm (a, b, perm3);
```

Generating AltiVec permute operations is much simpler, though one has to keep in mind that because the permute mask has to reside in a vector register, an additional SIMD load is necessary. This load is not required for SSE shuffles as the immediate is part of the assembly instruction.

5.5.2. Data Reordering

Data-alignment is the most protuberant problem when it comes to SIMDization (see Section 2.5.1). The aforementioned solutions are replication, padding or shuffling. In the context of kernel flattening, shuffling can be done globally or locally.

The CGiS compiler does not support replication of input or output data as we feel it is not efficient. Usually in stream processing large amounts of data are processed. Quadrupling data is too space and time consuming to get high performance because the SIMD parallelism contained in multi-media instruction set is not high enough to make this solution rewarding.

For all kernels that cannot be subjected to the kernel flattening optimization, e. g. because they contain gathering operations, `cgisc` supports *padding*. This includes a pre

	Kernel flat. local reorder	**Kernel flat. global reorder**	**Padding**	**Replication**
add. space in words	-	$n \cdot c$	$4c$	$3c \cdot n$
add. SIMD memory ops	-	$2n$	$2n$	$3n$
elem. proc. in parallel	4	4	1	1
additional operations	$2c$	-	-	-
additional registers	$(c-1)+1$	-	-	-

Table 5.2.: Space and time consumption by the different solutions to the alignment problem.

and post-processing step where data is re-arranged in memory to match the alignment requirements by inserting dummy elements.

Meeting the alignment by shuffling data is possible by reordering data locally at the start of the procedure or globally in the same way as padding is implemented. We support shuffling only in connection with kernel flattening.

Table 5.2 gives an overview of the different approaches to satisfy alignment. Hereby it is assumed that an input stream of a CGiS base type with $1 \leq c \leq 4$ components is processed that has n elements. For output streams, additional copy operations have to be appended after the stream data has been computed. In most cases kernel flattening with local reordering is the most efficient solution, as it does not include any additional memory operations. Each stream element is read only once. The additional operations introduced by local reordering are all shuffle or permute operations. Their number depends on the number of components of each stream element. The increase of the register pressure for this solution is mostly a problem for SSE with only 8 vector registers available but it still outperforms the heavy use of memory copies induced by the other solutions. AltiVec hardware features 32 vector registers so register pressure hardly becomes an issue.

In general the data-reordering phase only instantiates global reordering of stream data if necessary and otherwise inserts local reordering operations at the start and the end of procedure bodies for flattened kernels. It also assures the correct re-establishment of the original order after procedure body; this holds for every reordered stream and for temporary streams, e. g. streams with the flow specifier `inout`.

5.5.3. Loop Sectioning

Many data parallel algorithms, especially in image processing, require the gathering of nearby data elements. One example for such an image processing algorithm is the

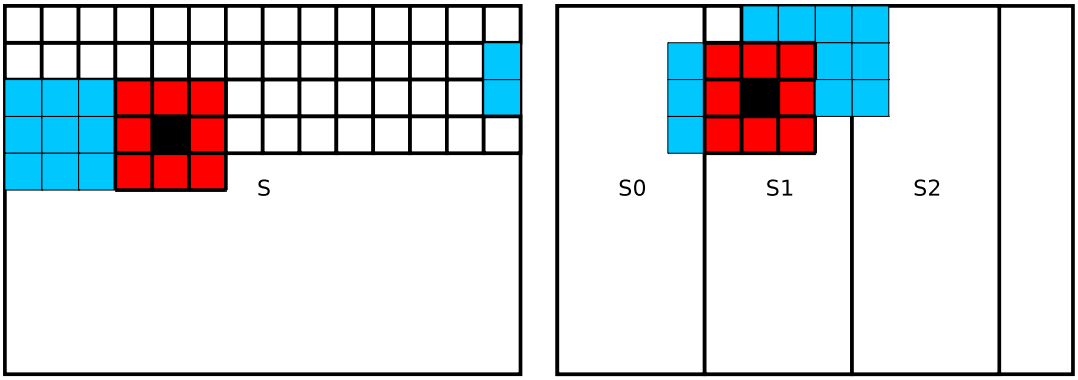


Figure 5.15.: Adaption of the iteration sequence for gathering.

Gaussian blur described in Section 6.2.4. CGiS supports gathering operations that let the programmer access stream elements relative to the current position in the stream. When iterating over a two-dimensional stream column-by-column or row-wise, it is possible that data elements already loaded and present in the data cache are evicted and have to be loaded again. To best make use of cached data, the CGiS compiler can adapt the iterations over the stream by dividing the field into smaller strips that better match the cache size and organization of the processor. This optimization was inspired by [4]. With the smaller width of the strips, there are more accesses to already cached data.

This optimization aiming at increased cache and runtime performance is not applicable for the Cell BE when using SPEs. These processing units only feature uncached local storage, thus making adapted access sequences unnecessary. Also this adaptation is possible only on architectures which allow a direct control of the iteration order. As for example GPUs do not allow this detailed iteration control, the GPU backend of `cgisc` cannot make use of this optimization.

As an example, consider Figure 5.15. A two-dimensional field S is processed, and for each element its 8 immediate neighbors are gathered. The blue (light) squares are data elements that have been loaded in former iterations and are thus present in the data cache. The element loaded in the current iteration is colored black, and the gathered elements are red (dark). In the left part, the iteration sequence is simply row-by-row. The right part shows the same field subdivided into smaller strips S_i. Each S_i is also processed row-wise. But with the reduced row width, the cache hit rate is increased because of better data locality.

The size of the strips is determined by the cache size and the memory requirements as follows. In a two-dimensional data field the strips can run either in the direction of the first dimension (X-axis) or the second dimension (Y-axis). To determine which axis fits best, we investigate the access pattern of the gather operations. The dimension which gives rise to the most data accesses defines the run direction. We assume that the two-dimensional stream is stored row-wise in memory. The strips then run column-wise. Iteration is row-wise inside the strips. For each stream, o determines the maximum of iteration lines or rows crossed by the access pattern, e. g., in Figure 5.15 o is 3. o is the

sum of maximum absolute offsets in strip direction plus one for the current line. For a given parallel kernel-execution k, the CGiS compiler decides the width of the strips S_k from the cache size C, the cache line size l and the size of the stream elements read and written. (Different architectures with different cache sizes are selected at compile-time.) δ is a constant number that represents the local data that is needed in each iteration such as intermediates and other stack data. Assume that the kernel k accesses stream elements of size a_i and the gathers for a_i cross o_i lines. These parameters are statically known and result in a simple heuristic for computing the strip width:

$$S_k = \left\lfloor (C - \delta)/(\sum o_i \cdot a_i) \right\rfloor_l$$

$\lfloor \rfloor_l$ rounds down to the nearest multiple of l. S_k does not need to be constant across a whole program but is adapted to each specific kernel execution.

Should the size of the field not match a multiple of the strip width, the remaining elements are processed by normal iteration.

Boundary Checks

Data accesses relative to the currently processed stream element are widely used in image processing algorithms and physical simulations. A point in a grid is modified according to its relation to its neighbors. While on GPUs the boundary checks come "for free" with the hardware, on CPUs range checks have to be performed to avoid segmentation faults and wrong data accesses. These checks are very time consuming and thus have a great impact on performance. For example in a demosaic filter, almost 50% of the computation time is spent in the gathering function. To improve the performance of gather operations the stream iteration is split in two. In the first iteration only the elements not accessing out-of-bounds neighbors are considered and range checks are omitted. In the second processing the remaining elements are iterated with range checks. For arbitrary large fields, the performance gain is roughly 25%.

As the access pattern of the gathers is statically known, the elements that need to be checked for violating the stream boundaries can be determined at compile time. For a two-dimensional stream the set of stream elements needing boundary checks can be thought of as a frame. In Figure 5.15 the width of this frame is one. In general the width of the top strip is the maximum gather distance in vertical direction, minimum gather distance for the bottom strip. Left and right strips are determined by the horizontal gather distance.

5.5.4. Thread Management

In this part of the SIMD backend the code generation varies greatly for the different targets. This stems from the fact that the Cell BE is a multi-core processor of whose processing elements can be controlled by assigning threads. To explain the way the threads are created, we will firstly examine the translation of the forall statement which is common to all three targets. Then we will proceed to the generation of threads for the Cell BE and have a quick look at the appearance of SPU programs.

Forall-Loops

Forall-loops in CGiS programs express the explicit parallelism of the iterated streams. In such a loop each stream gets its iterator and only calls to kernels specified in the CODE section may be stated. The iterator(s) can be passed as parameters to these kernels. It is assumed that all streams processed in one forall-loop body have the same number of elements. Preserving the semantics, a CONTROL section can be seen as a series of forall-loops, each containing exactly one call to a kernel. Furthermore it is assumed that the called kernels and all subsidiary kernels are either fully flattened or process one stream element at a time. For the SIMD backend, the translation of forall-loops has the following steps for each loop in the series:

- ☐ Determine the possible parallelism for each call and adapt stream data if necessary: if the called kernel is flattened, stream data is expected in original order. In any other case all data of iterated streams must be padded except for vector types matching the allignment.
- ☐ Include the computed iteration sequence from the loop sectioning optimization if applicable.
- ☐ Issue the rebuilding of the original layout after the last forall-loop or if needed by the next forall-loop.
- ☐ Generate the corresponding C++-loop(s).

Threads

The Cell BE is a multi-core architecture consisting of a PowerPC processor and eight synergistic processing units. An application using these units needs to do the following tasks to run programs on an SPE:

- ☐ Upload a binary image of the program to be run on the SPE.
- ☐ Set the SPE program counter to the start of the binary.
- ☐ Issue the start of the SPE program.
- ☐ Wait for the program to finish.

This commonly is done by creating threads and distributing them to the different SPEs. The thread management is done on the PPE. The development kit provided by IBM includes a library called `libspe2`[22] which works together with the `libpthread`[40] to support thread handling.

CGiS offers two ways to distribute the threads to the SPEs: static or dynamic load balancing. With static balancing, the idea is to split the iteration sequence equally between all the available SPEs at the start and wait until all are finished. Dynamic balancing splits the iteration into smaller chunks and assigns a chunk as soon as a SPE is unoccupied, i. e. a work-list distribution. The latter alternative imposes a communication

overhead but can increase overall performance for algorithms where the computation time varies for different stream element because of dynamic loops for example. The computation of the Mandelbrot set in Program 6.3 exhibits such dynamic loops.

For AltiVec and SSE, the backend only generates single-thread applications. New generations of the Intel architectures feature two cores, so implementing threads for SSE targets might be profitable as well. In this work only the Cell BE is considered a multi-core architecture.

Generation of SPE Programs

As SPEs can work independently from the PPE, it is not sufficient to translate a kernel into code suitable for SPEs but a small SPE application has to be generated by `cgisc`. Such an SPE program needs to complete the following steps:

- ☐ Fetch the parameter structure from the main memory.
- ☐ Copy the data to be processed into the local storage via DMAs.
- ☐ Start the iterations by calling the fully inlined kernel.
- ☐ Copy results back into the main memory.

When the PPE creates a thread to trigger SPE execution, it can pass one address to the SPE as a program parameter. For CGiS programs this address is a pointer to a structure in the main memory that holds addresses from which input data should be loaded and to which output data should be written. When the SPE starts, it firstly fetches this parameter structure.

Let us quickly recapture the properties of the direct memory accesses presented in Section 2.4. DMAs are asynchronous and always transport a multiple of 128 bytes which corresponds to a cache line in the Cell BE's main memory. With the development kit provided by IBM comes a library for managing these DMAs. This library contains functions to tag the memory accesses and wait for certain tags to complete before continuing the program.

In order to achieve the best performance, the CGiS compiler combines several kernel calls into one SPE program. One call would correspond to exactly one iteration in a forall-loop. In our implementation, the SPEs always execute multiple forall-iterations. The number of these iterations is chosen in a way that the data loaded matches exactly a multiple of the cache line size. To further hide the latency of the DMAs double buffering is implemented; while the data for the current iteration are processed the input data for the next iteration are fetched and the output data of the previous iteration is stored.

Program 5.8 shows the simplified SPE program of the alpha blending example, generated for the CGiS Program 4.1 on page 40. In the program an outer and an inner loop can be seen. The inner loop handles the calls to the procedures, while with the outer iterations double buffering is implemented. The outer loop issues the load of the data for the next iteration, switches tags for the DMAs and waits for the data to be processed by the inner loop. The load of this data has been issued by the previous outer loop

```
int main (unsigned long long speid __attribute__ ((unused)),
          unsigned long long parms_ea)
{
  ...

  // get the parameter struct
  mfc_get(&params, parms_ea, sizeof(params), tag_id[0], 0, 0);
  DMA_WAIT(tag_id[0]);  // tag_id contains 2 reserved tags

  // local storage data
  vector float ipixel_ls[2][24];
  ...

  // DMAs for in data
  unsigned long long ea = (unsigned int) params.ipixel;
  mfc_getb (ipixel_ls[0], ea, 384, tag_id[0], 0, 0);
  ...

  // begin outer iterations...
  char current = 0; char next = 0;
  for (outer=0; outer<iters; outer++) {

    // DMA iterations for in data
    ea = ((unsigned int) params.ipixel) + (outer+1)*384;
    mfc_getb (ipixel_ls[next], ea, 384, tag_id[next], 0, 0);

    // switch the DMA tag
    current = next; next = 1 - current;
    DMA_WAIT(tag_id[current]);

    // begin inner iterations
    for (i=0; i<8; i++) {
      blend_f_Cell (&(ipixel_ls[current][i*3+0]),
                    &(ipixel_ls[current][i*3+1]),
                    &(ipixel_ls[current][i*3+2]), ...);
    }

    // DMAs for out data
    ...
  }

  // let the last stores finish before exiting:
  DMA_WAIT(tag_id[current]);
}
```

Program 5.8: Simplified SPE program for the alpha blending example.

iteration. Let us now examine the allocation and filling of the variable `ipixel` of the original CGiS program. `ipixel` is of type `float3` and thus one instance of it occupies 12 bytes of memory.[9] The procedure `blend` has been flattened, so four iterations are executed per call in parallel by SIMDization needing 48 bytes of data. For this data a buffer of 96 bytes in the local storage is reserved because of double buffering. This buffer is called `ipixel_ls`. The SPE program issues eight of those four-way executions and thus it needs 384 bytes per DMA load for the variable `ipixel`.

The number of outer iterations is set by the CGiS compiler and depends of the type of load balancing selected. The storage of output data is handled in the same way as input data. An SPE program finishes as soon as the last output data have been stored.

5.6. Runtime System

The runtime system of CGiS consists of two parts, the runtime libraries and the program specific routines for setting-up, starting the computations and getting back the results.

Each application containing routines created with the CGiS system needs to be linked against the appropriate runtime library. A runtime library for a SIMD target comprises the following functionalities:

- ☐ Stream management functions: raw data passed to the CGiS part of an application gets packed in a stream class that provides various access and manipulation functions.
- ☐ Simulation functions for features not supported by the hardware: CGiS has special functions and operations that cannot always be mapped to SIMD operations. In that case the runtime provides simulation or approximation functions.
- ☐ Debugging and utility functions: functions for the combination of different masks, evaluation of conditions and debug output of SIMD register are kept in the runtime library as well.
- ☐ Setup and cleanup functions.

For each CGiS program, a set of routines is created individually and also belongs to the runtime system. These routines allow the programmer to interface with the generated SIMD code. Examples of these functions can be seen in Program 4.2 on page 42. Setup functions are used to pass the stream data to the CGiS data structures. The iteration can be started via an execute function and a provided function to receive the results can be called to copy the data back. The following list comprehends the tasks the generated functions comply:

- ☐ Create a CGiS program and initialize the data structures.

[9] On all presented target architectures the width of the single-precision floating-point data type is assumed to be 32-bit.

- ☐ Set up the stream data by passing it to the CGiS functions.
- ☐ Start the stream iterations.
- ☐ Collect the results.
- ☐ Clean up intemediate data and delete the created program and stream structures.

5.7. Remaining Infrastructure

Apart from the compiler `cgisc` and the runtime libraries, a testing environment is also part of the system. This environment contains a large set of regression tests for every language feature available. A set of more complex examples is also included, for the SIMD backend these are discussed in the following chapter. The GPU examples are not re-showcased in this work.

Additionally, some support libraries are used to ease up the work with CGiS applications. These include an utility library for various support and debugging routines, a bitmap library used in many tests and examples to load, store and modify image data and an OpenGL library for the GPU backend.

5.8. Acknowledgements

The initial version of the compiler and all parts of the GPU backend have been designed and implemented by Philipp Lucas. He also instantiated the regression tests and maintains the developing of `cgisc` for Windows. The autoconf build-system was created and is supported by Gernot Gebhard, who also is responsible for some parts of the auxiliary libraries. Cell BE applications created with CGiS depend on the libraries for SPE control and communication which are part of the Cell BE Software Development Kit, version 1.3. All SIMD runtime libraries contain parts of the libsimdmath, an open source library implementing trigonometric functions, exponential and logarithmic functions for SIMD hardware.

CHAPTER 6

Examples and Experimental Results

Rewriting algorithms in CGiS or learning a new data-parallel programming language is time-consuming and requires good motivation. This chapter tries to provide this motivation by showing a set of examples and their performance evaluation. The application domains of the examples are widespread to demonstrate the usability of CGiS for many problem domains. Applications from the fields of physical simulation, encryption, mathematical exploration, image processing, financial simulation and computer graphics are given. The examples vary in complexity and pose different challenges for efficient execution on SIMD hardware. With each example, its peculiarities and the performance data for the test environments are given and the results are discussed.

First, the testing environments are described in Section 6.1. For each of the SIMD hardware architectures introduced in Chapter 2, a sample system was configured. Section 6.2 presents the different CGiS programs and their performance on the different host systems. An overall interpretation of the results regarding the efficiency of the generated SIMD code is given in Section 6.3. Finally, an outlook on the possibilities of further development of the CGiS compiler to increase performance of the generated code is presented in Section 6.4.

6.1. Testing Environment

For each application the test is executed on the host system with standard non-SIMD execution and the results are compared to SIMD execution. Because PowerPC architectures are designed differently than the wide-spread x86, comparison between different host systems is not significant. For the Cell BE we compare three results to standard non-SIMD execution: SIMD execution on the PPE, SIMD execution with static load balancing on SPEs and SIMD execution with dynamic load balancing via worklists on the SPEs.

First, the three different testing environments are introduced. It is worth mentioning that the Intel Core2Duo and Core i7[25] as well as the PPE in the Cell BE feature

Feature	Core2Duo E8400	Core2Duo E6300	Core i7 920
Cores	2	2	4
Clock rate	3.00GHz	1.86GHz	2.67GHz
FSB	1333MHz	1066MHz	1066MHz
SIMD unit	SSE1,2,3,4.1		
L1 Cache	1MB	1MB	32kb per core
L2 Cache	6MB	2MB	256kb per core
L3 Cache	-	-	8MB
RAM	2GB DDR3	2GB DDR3	12GB DDR3
OS	Ubuntu 9.04	Ubuntu 8.04	
Kernel version	2.6.29		
gcc version	4.3.2	4.3	4.2.4
icc version	-	11.0	-

Table 6.1.: Overview over the SSE test systems features.

Feature	PowerPC G4 PPC7450	Playstation 3 PPE	Playstation 3 SPE
Cores	1	1	6 SPEs
Clock rate	733MHz	3.2GHz	
FSB	133MHz	EIB (1.6GHz)	-
SIMD unit	AltiVec	AltiVec	close to AltiVec
L1 Cache	32KB	32KB	-
L2 Cache	256KB	512KB	-
RAM	512MB DDR	256MB	256KB LS
OS	Mac OSX	Yellow Dog Linux 6.0	
Kernel version	8.8.0	2.6.22	
gcc version	4.0.1	4.1	

Table 6.2.: Overview over the AltiVec and Cell test system features.

hyper-threading, dual-cores or quad-cores. The CGiS compiler does not make use of this potential for two different reasons. First of all, the measurements become cleaner. Each processor has an operating system with different tasks running in the background. Dual-issuing interferes with those tasks and spreads the measurement range. Secondly, the comparison against standard scalar execution is better, i. e. the performance difference is only a result of using CGiS: examples not dominated by memory latencies would achieve further speedups by threaded execution while algorithms where memory accesses are the bottleneck would not profit much from threading anyway. In all the following examples, the code on the Core2Duo, the i7 and the PPE is executed on a single core, by single-threaded execution.

In Table 6.1 the characteristical data of the testing systems featuring SSE hardware is gathered. Table 6.2 holds the corresponding data for the AltiVec hardware and the

Cell BE. In the following, the single systems are put forward in more detail.

6.1.1. Intel Core2Duo and Core i7 Architectures

The first SSE machine we investigate here is a 64bit Intel Core2Duo. Intel's Core2Duo E8400, code-named Wolfdale, has two x87 cores which run at 3.00GHz. It is a 45nm architecture with 6MB level 2 cache and a frontside bus rate of 1333MHz. It features all Streaming SIMD Extensions up to version 4.1. The system contains 2GB of dual-channel DDR3 memory and is running an Ubuntu Linux, version 11. All tests were compiled with `gcc`, version 4.3.2, with optimization level 2.

To examine the behavior of SSE execution on different compilers we tested all our examples on a Core2Duo E6300 with the Intel C Compiler, `icc`, in direct comparison to gcc 4.3. The E6300 belongs to the same CPU family as the E8400, it is an older model though based on 65nm technology supporting lower clock speed. It only runs at 1.86MHz for both cores.

Our last computer for testing CGiS on SSE hardware is the Intel Core i7 architecture. This is a quad core with hyper threading support running at 2.667MHz. The gcc used here is slightly older than its relatives, version 4.2.4. This architecture is the first not bound to alignment constraints. Transfering data into XMM register from memory or vice versa does not require a 16-byte aligned address. Unaligned accesses are as fast as aligned ones. Still, this does not render our transformations and optimizations unnecessary. Depending on the data layout, remember the YUV stream from Program 5.3 on page 79, even with unaligned loads data is not processible without transformations in SIMD registers.

6.1.2. PowerPC G4

All tests for the AltiVec backend of the CGiS compiler were run on system with a PowerPC G4. The G4 is the fourth generation of Apple's 32-bit PowerPC series. The type G4 stands for a subset of closely related PowerPCs and the one examined here is the MPC7450[52]. It was introduced with the 733 MHz Power Mac G4 in January, 2001.

The PowerPC 7450 is the only major redesign of the G4 processor. It features a longer pipeline, 32KB L1 cache, 256KB on-chip L2 cache, and introduces up to 2MB external L3 Cache. Also the AltiVec hardware was upgraded with this revision. In earlier designs there were two different AltiVec units: one handled permute operations and the other the arithmetics. With the introduction of the PPC7450 two arbitrary AltiVec instructions can be dispatched at the same time. The frontside bus runs at 133MHz and the system contains 512MB RAM. The operating system, Mac OSX, is based on Free BSD with a Darwin kernel version 8.8.0. The installed gcc is version 4.0.1.

6.1.3. Playstation 3

Sony's PlayStation 3 video game console contains the first production application of the Cell processor, clocked at 3.2 GHz and featuring seven out of eight operational SPEs.

Only six of these seven SPEs are accessible to developers as one is reserved by the OS. Graphics processing is handled by the NVidia RSX 'Reality Synthesizer', which can output resolutions up to 1080p. The PlayStation 3 has 256 MB of XDR main memory and 256 MB of DDR3 video memory for the RSX. As of now there is no way of accessing the RSX for general purpose programming.

The PlayStation is running a Game OS from its firmware but allows the installation of a third party OS. Yellow Dog offers a Linux for the PS3, which is installed on the test system in version 6.0. To free as many resources as possible in this testing environment, this Linux has been configured to only run the needed applications, omitting for example the X server. The gcc coming with this release is version 4.1.

The two different approaches for distributing the computations on the SPEs are static and dynamic distribution as described in the previous chapter. As only six SPEs are available, static distribution divides all computations evenly among those six. The dynamic version uses equally sized packs of computations and hands them to an SPE whenever it is ready, i. e. as soon as an SPE is unoccupied it gets started again with one of the packs to be processed. The default size of a pack when the kernel flattening optimization is active is 64 calls to the kernel. This means that a total of 256 stream elements is processed in one pack.[1]

6.2. Applications

In the following we will examine a set of six sample applications. For all tests containing floating-point computations we assume IEEE 32-bit single-precision. For all stream data, we assume that the first stream element of the stream is 16-byte aligned. The runtime library of CGiS provides memory allocation functions to match this alignment requirement. In all the presented numbers the costs of reordering data, before and during processing, is included.

6.2.1. Particle

This first example is a toy particle system which moves particles through 3-dimensional space over an arbitrary amount of time steps using Euler integration[20]. The movement and the change in velocity of the particles is based on the Newton equations. The force of a particle is the product of its mass and its acceleration. Its acceleration is the change in velocity divided by the change in time, and its velocity is the change in position divided by the change in time.

$$f = m \cdot a, a = \frac{dv}{dt}, v = \frac{dx}{dt}$$
$$dv = \frac{f}{m} \cdot dt, dx = dv \cdot dt$$

In this simplified form, position and velocity vectors of a particle are computed discretely by

[1] A flattened kernel always executes 4 stream elements in parallel.

$$
\begin{aligned}
p(t) = \quad & p(t-1) + v(t-1) \cdot dt \quad && = p_0 + \sum_{i=0}^{t-1} v(i) \cdot dt \\
v(t) = \quad & v(t-1) + \tfrac{f}{m_p} \cdot dt \quad && = v_0 + \sum_{i=0}^{t-1} \frac{f}{m} \cdot dt
\end{aligned}
$$

with f and m being constant per particle.

The CGiS program for the simulation is shown in Program 6.1. The three streams `positions`, `velocities` and `masses` hold the positions of the particles in a 3-dimensional space, their velocities, and their masses, respectively. The procedure `sim` then computes the new position and velocity after a certain discretized time interval.

```
PROGRAM particle;

INTERFACE

extern inout float3 positions<PARTICLES>;
extern inout float3 velocities<PARTICLES>;
extern in float masses<PARTICLES>;
extern in float steps;
extern in float dt;
extern in float3 force;

CODE

procedure sim (inout float3 pos,
               inout float3 vel,
               in float mass,
               in float dt,
               in int steps,
               in float3 force)
{
  for (int i = 0; i < steps; i++) {
    pos += (vel*dt);
    vel += (force*dt/mass);
  }
}

CONTROL

forall(p in positions, v in velocities, m in masses)
  sim(p, v, m, dt, steps, force);
```

Program 6.1: Simple Euler particle system simulation.

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
102k	248	33	98	14	466	51	252	47
204k	487	52	198	27	934	98	416	75
307k	735	68	295	38	1399	145	698	79
409k	982	90	394	50	1871	193	859	119
512k	1228	112	492	64	2338	241	1102	156
∅ speedup		9.841		7.419		9.535		6.804

Table 6.3.: Results of particle test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
102k	3828	294	4214	27	59	101
204k	7622	584	8439	53	111	198
307k	11443	883	12743	80	163	301
409k	15534	1196	16892	106	215	439
512k	19427	1462	21123	133	268	539
∅ speedup		13.048		157.730	76.848	41.465

Table 6.4.: Results of particle test for AltiVec and Cell BE.

Characteristics and Performance

The particle system simulation is a very simple example. The computations are restricted to standard floating-point arithmetics, no control flow branches or additional data loads are present.

Positions, velocities and masses are chosen randomly, dt and f are arbitrary. For this test set, we assume a number of 64 time steps. The performance results of the particle test are given in Table 6.3 and Table 6.4. The left row holds the size of the input set. For every target architecture the CGiS version of the program runs faster than its scalar version. All execution times scale linearly with the size of the input set. This example is also part of the tutorial in the Cell BE SDK3.0 and has the following characteristics:

- ☐ Simple floating-point arithmetics.
- ☐ Inner loop depending on a CGiS scalar.
- ☐ Optimizations applied: kernel flattening, scalarization of loop counter.

SSE

The SSE version of the particle simulation shows a large speedup by more than a factor of 9 on the Core2Duo machines with the gcc compiler. The third column in Table 6.3

holds the measurements for the Intel C++ Compiler comparing scalar execution to SIMD execution. There are two interesting observations about the numbers in this column. Scalar as well as SIMD execution is much faster than the gcc versions on the same machine, namely around a factor of 4.5. There are mainly two reasons for this performance discrepancy:

- icc uses faster instructions for the floating-point computations.
- It also produces more efficient code in terms of applied optimizations.

Let us investigate these two claims a bit further. At first we take a closer look at the scalar code generated. Intel's compiler uses the SSE instruction in scalar mode to compute the floating-point operations. The SSE instruction set features every SIMD floating-point operation also in a scalar version. This version only computes the operation for one floating-point value instead of four, but is therefore not bound to alignment requirements. The Gnu C compiler, on the other, hand uses the slower but much more precise scalar floating-point unit. To achieve the 32-bit IEEE single-precision floating-point compliance, the results of computations are always written back to memory.

In the scalar implementation of the particle example, icc manages to unroll the inner loop over the components where for each component x, y and z the position and velocity gets updated:

```
;; for(k=0; k<3; ++k){...
...
;;   vel[j][k] += 1/mass[j]*force_dt[k];
movss     %xmm5, 8(%edx,%edi)
divss     (%eax,%ecx,4), %xmm6
addss     (%edx,%esi), %xmm6
movss     %xmm6, (%edx,%esi)
divss     (%eax,%ecx,4), %xmm7
addss     4(%edx,%esi), %xmm7
movss     %xmm7, 4(%edx,%esi)
divss     (%eax,%ecx,4), %xmm3
incl      %ecx
addss     8(%edx,%esi), %xmm3
...
```

We can see that the scalar SSE instructions are used, enabling much faster single-precision floating-point computations.

To verify the second claim we take a closer look at the generated SSE code from the intrinsics. Here the other interesting difference between gcc and icc becomes obvious: although both SSE executables run on the same machine, the icc generated code runs almost 4 times faster than the gcc generated one. As the instruction selection is basically done by the CGiS compiler with intrinsics, another compiler task must be responsible for the performance gap: this is the register allocation. In Appendix C the interested reader can find snippets from the assembler code generated by gcc and icc. Two differences can

be seen. The Gnu C compiler has much more spilling code in form of stack operations than icc. Each of these operations writes or reads 16 bytes to or from memory. As this code is part of the loop body of an inner loop in the context of stream processing, none of the stored intermediate data is used multiple times disabling any performance increase by caching. The other difference is the successful loop-invariant code motion of the icc. The term `(force*dt/mass)` from Program 6.1 is moved out of the loop over `steps` significantly reducing the number of division and multiplication operations. Though it is a pretty simple example with easily accessible parallelism in the C++ implementation, both compilers fail at automatic vectorization.

On the Core i7 the results are similar, even though with a speedup factor of 6.8 a bit less than on the Core2Duo processors. The complete results of the Core i7 can be seen in the fifth column in Table 6.3.

AltiVec

The results of the particle test on the G4 are kept in the second column of Table 6.4. A remarkable speedup of 13 could be achieved by the CGiS implementation. This is because of AltiVec featuring a multiply-add operation. The CGiS compiler is able to replace a multiplication and a subsequent addition twice reducing the number of computation operations.

Let us at take a look at the scalar results of the test for the PPE in the third column. Though the PPE runs at higher clock speed and has a faster memory connection, the program on the older G4 runs faster. For example for 102k particles the G4 needs 3828ms, while the PPE needs 4214ms. This unusual performance difference results from the distinct floating-point division operations: the `fdiv` instruction of the MPC7450 takes 14 to 35 cycles of latency which is less than half the latency of its PPE counterpart instruction, resulting in faster scalar code.

Cell BE

In the tutorial coming with the Cell SDK3.0, this example was used to demonstrate the successive parallelization of the original algorithm. While the tutorial shows the by-hand SIMDization and parallel threaded execution, here only the specification of the CGiS program results in very similar code with a huge amount of saved time on the programmer's part.

For 102k particles scalar execution needs 4214ms, while the AltiVec implementation only needs 27ms. The high speedup of 157.73 gained for AltiVec execution on the PPE in comparison to scalar execution stems from the high latencies of the `fdiv` instruction in the PowerPC core. It consumes 74 cycles and stalls the floating-point pipeline which can propagate back to dispatch as the operation is recycled in the FPU issue queue.

The performance gains for the SPE are 76.848 for static load balancing and 41.465 for dynamic. The PPE version of this example performs exceptionally well compared to the SPEs. This results from the DMAs being slower than the cached memory accesses

of the PPE. There are not many computations inside the kernel making up for the slow DMAs.

If one takes a closer look at the differences between the static and the dynamic load balancing, it becomes apparent that the static balancing method outperforms the dynamic one by a factor of 1.7. The inner loop inside the kernel `sim` is constant for all elements so every single SPE will take more or less the same time to compute the same amount of data. The static version splits the input data in 6 parts and initializes the SPEs. The dynamic version, on the other hand, takes smaller chunks, distributes them, waits for ready SPEs and continues until all data is processed. Thus, dynamic scheduling introduces additional communication overhead with no performance gain.

6.2.2. RC5

The next example investigated is the RC5[48] encryption. It was published by Ron Rivest in 1994. This encryption method belongs to the class of *block cipher encryptions*. Input data is divided into equally sized *blocks* and encoded with a *key sequence* over a certain number of *rounds*. The size of the data blocks can be 32, 64 or 128 bits; key lengths of 0 to 2040 bits are supported and the number of rounds can go up to 255. The underlying example uses a block size of 64 bits, a key length of 16 bits and the number of encryption rounds is set to 32 (RC5-64/16/32).

Alongside block cipher encryption there is also stream cipher encryption. The difference between these two is that the internal state of the algorithm changes as the encryption moves to the next data block for stream cipher encryption. This makes algorithms like the A5/1[1] stream cipher used in cellular phones not realizable in CGiS, as all stream data is processed in parallel. In block cipher encryption all parts of the input data are processed independently thus making this class of encryption algorithms perfectly suitable for CGiS.

Program 6.2 contains the CGiS code for the RC5-64/16/32. The procedure `encrypt` implements the encryption of the stream `ABs`. The procedure `get_S01` returns the key value from the stream `S` for the given round number. In a pre-computation step, the encryption key is expanded to match the number of rounds.

Characteristics and Performance

The implementation of the rc5 block cipher in CGiS features a loop with a static number of iterations in the encryption function. A special rotate operation as well as lookups, i.e. random access of stream data, are present. In short:

- ☐ Logical computations on unsigned integers.
- ☐ Static loop with 31 iterations per element.
- ☐ Lookup operations inside the main loop.
- ☐ Optimizations applied: inlining of lookup procedure, kernel flattening, scalarization of loop counter and lookup operation.

```
PROGRAM RC5;

INTERFACE
extern in uint2 S<32>;         // Expanded key
extern inout uint2 ABs<SIZE>; // The stream to be encrypted.

CODE
procedure get_S01(in uint2 S<_>, in uint where, out uint2 s)
{
  lookup S: temp<where>;
  s = temp;
}

procedure encrypt(inout uint2 AB, in uint2 S<_>)
{
  uint2 S01;
  uint i = 0;

  get_S01(S,i,S01);   // Get S[0] and S[1].
  uint A = AB.x+S01.x, B = AB.y+S01.y;
  while(i<31) {
    i = i + 1;
    get_S01(S,i,S01); // Get S[2*i] and S[2*i+1].
    A = ((A^B)<<<B) + S01.x;
    B = ((B^A)<<<A) + S01.y;
  }

  AB.x = A; AB.y = B;
}

CONTROL
forall(AB in ABs) encrypt(AB,S);
```

Program 6.2: Rc5 encryption.

In Table 6.5 and Table 6.6 the performance results of the test are presented. The leftmost column always holds the size of the input data, i.e. the length of the message to be encoded. All runtimes in scalar execution as well as the runtimes of the CGiS implementations scale linearly with the size of the input.

SSE

Clearly an implementation with CGiS is not profitable here since the scalar version of rc5 runs 12% faster on average. As an example for the measurements done for gcc, examine the rightmost column of Table 6.5. For the smallest message length of 768k bytes the scalar execution takes 118ms while the SSE version takes 138ms. This is exemplary for

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
768k	85	99	173	143	123	141	118	138
1536k	163	182	340	273	234	275	190	250
2304k	225	266	503	418	349	406	276	333
3072k	298	349	667	537	467	537	352	421
3840k	377	428	839	671	581	671	416	554
∅ speedup		0.867		1.231		0.865		0.808

Table 6.5.: Results of rc5 test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
768k	189	65	308	41	199	386
1536k	377	125	617	82	393	790
2304k	568	187	925	123	586	1201
3072k	782	250	1233	164	779	1547
3840k	950	336	1543	205	973	1908
∅ speedup		2.974		7.504	1.573	0.791

Table 6.6.: Results of rc5 test for AltiVec and Cell BE.

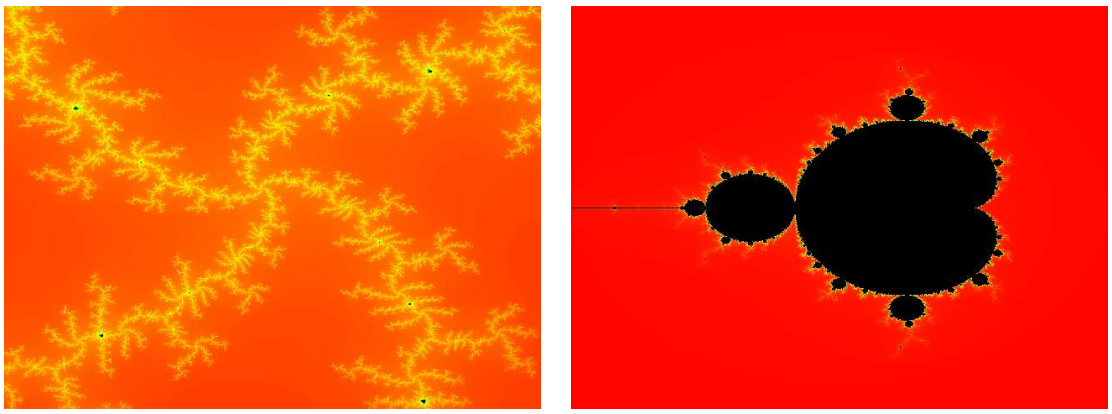
all gcc measurements on SSE hardware. The problem with SSE is the missing rotation operation and the lack of appropriate instruction to construct or simulate it. As a result, the CGiS compiler inserts compensation code which stores the data to be rotated in memory and uses scalar instructions to process it. Then again, the data needs to be loaded back into XMM registers.

With the Intel compiler, even a small performance gain becomes visible. Unfortunately, this can not be attributed to the excellent code generated by the CGiS compiler but rather to the suboptimal scalar code generated by icc. On page 22 such a rotation operation is defined in C and exactly corresponds to the scalar compensation code inserted by the CGiS compiler. The Gnu C compiler generates a much more efficient sequence of instructions than the icc which is responsible for the faster gcc generated code here.[2]

AltiVec

As expected, the AltiVec version shows good performance gains. The second column in Table 6.6 lists the measurements for the different message sizes on the G4. The average speedup is 2.974. The AltiVec operation `vrlb` introduced in Section 2.5.2 amortizes the costs of the lookup and reorder operations needed to re-arrange data. Taking a closer look at the generated code gcc is not able to recognize the scalar rotation and thus does

[2]The compensation code is used in both scalar and SSE execution as well.



(a) Balanced computations. (b) Varying complexity.

Figure 6.1.: Mandelbrot computations

not use the built-in hardware rotation. As the rotation is a series of logical and shift operations, it is rather hard to detect for a C++ compiler.

Cell BE

The PPE encryption shows an average speedup of 7.5. Since the AltiVec instruction sets for the G4 and the PPE are similar, it also profits from the hardware rotation. As an example measurement, examine the the last entry in the scalar column of the Cell BE part of Table 6.6. For a message of 3840k bytes scalar execution takes 1543ms which is rather slow compared to the much older G4 which only needs 950ms to encrypt the same message. The AltiVec execution on the PPE, on the other hand, processes the message in 205ms which is more than 50% faster than the encryption on the G4 with AltiVec (336ms).

For the SPEs the static load distribution beats the dynamic one. In fact, a performance gain is only visible for the static version with 1.573 on average. The lookup operation introduces significant performance loss: there are two possibilities to realize these stream lookups on SPEs. One would be inserting a DMA to the main memory for each lookup, the other is to copy the whole stream to be looked-up into the local storage. In the rc5 example the stream looked up is of relatively small size, so the CGiS compiler issues a full copy of the stream to the SPEs with the start of each threaded execution. Still this copying imposes additional memory operations which cannot be amortized enough.

6.2.3. Mandelbrot

The Mandelbrot set[60] is a very well known mathematical set. It is often visualized with colorful pictures most people have seen. In Figure 6.1 two different Mandelbrot sets are visualized. Those sets can be computed with the following sequence.

Let $c \in \mathbb{C}$. The sequence $(z_n^c)_{n\in\mathbb{N}_0}$ is defined as $z_0^c = c$ and $\forall n \in \mathbb{N} : z_n^c = (z_{n-1}^c)^2 + c$.

The *Mandelbrot Set* is defined as $M := \mathbb{C}\ c \in \mathbb{C} : lim_{n\to\infty}|z_n^c| = \infty$. For computing the picture of a Mandelbrot set for some $Z \subset \mathbb{C}$, the recurrence $z_n^c = (z_{n-1}^c)^2 + z$ is computed for each $z \in Z$ until either $|z_n^c| \geq 2$ or the loop counter exceeds a predefined maximum value. Colors are assigned to the points depending on the outcome of this process.

Program 6.3 holds the CGiS implementation of the computations of the above sequence. The procedure `mandelbrot` is called for each element of the streams `RE` and `IM` holding the real and imaginary part of the complex numbers corresponding to pixels in an image. The variable `max_iter` holds the maximum iteration count at which computation is stopped if the breaking condition has not been reached.

Characteristics and Performance

The kernel of the implementation of the Mandelbrot set computation contains a tight data-dependent loop with a possibly large number of iterations. The iteration of this loop might terminate prematurely for some elements, should the condition for halting be met. In the investigated example, the iteration maximum is set to 500. Parallelizing the loop requires full control flow conversion as to ensure correct termination of the iterations. The CGiS compiler fully converts the control flow into data-dependencies and produces statements for masking and selecting as described in Section 5.3.6, assuring that each element is iterated only as often as wanted. To exploit SIMD parallelism, kernel flattening can then be applied.

Summed up, this example is characterized by

- ☐ Simple floating-point operations.
- ☐ Data-dependent loop iterations.
- ☐ Optimizations applied: control flow conversion, kernel flattening.

To create the image in Figure 6.1(a) we have chosen the following parameters for the computation $Z = \{(x, y) : x \in [0.27525, 0.28371],\ y \in [-0.6101, -0.6015]\}$. The resulting image can be seen in Figure 6.1(a). Tables 6.7 and 6.8 hold the performance results of the test runs. Obviously the application suits the CGiS implementations well though speedups vary across the different targets.

Increasing the loop iteration maximum causes the speedups of the CGiS versions of the example to raise by a small amount. The inner loop is then executed more often and since it does not reload any data, the SIMD versions can profit from the data locality and the induced dominance of computation operations.

SSE

For the computation of the Mandelbrot set an approximate speedup by a factor of 3 would be the theoretical expectation, given the fact that control flow conversion introduces additional operations that a scalar implementation does not feature. There are only few and simple floating-point computations present that can be parallelized.

```
PROGRAM mandelbrot;

INTERFACE
extern in float RE<SIZE>;
extern in float IM<SIZE>;
extern out float OUT<SIZE>;

CODE
procedure mandelbrot (out float point, in float re, in float im)
{
  float curz=1;
  float tempx=re, tempy=im;

  int round=0;
  int max_iter = 500;

  while(round<max_iter)
  {
    float temp=tempx;
    tempx=tempx*tempx - tempy*tempy + re;
    tempy=2*temp*tempy+im;

    if(tempx*tempx + tempy*tempy >= 4) {
      // We have reached the condition for halting
      curz=-round-1;
      round=max_iter;
    } else {
      round=round+1;
    }
  }

  point = curz;
}

CONTROL

forall(point in OUT, re in RE, im in IM)
  mandelbrot(point, re,im);
```

Program 6.3: Computation of the Mandelbrot set

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
320 × 240	93	15	50	15	47	19	40	20
640 × 240	176	29	96	31	102	37	61	35
640 × 480	352	55	205	58	191	74	116	58
1280 × 480	709	108	407	110	383	133	225	110
1280 × 960	1373	170	825	211	755	259	418	195
∅ speedup		6.598		3.485		2.696		1.985

Table 6.7.: Results of Mandelbrot test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
320 × 240	127	57	88	35	40	73
640 × 240	263	110	177	69	73	138
640 × 480	535	220	348	137	139	225
1280 × 480	1052	432	701	269	264	541
1280 × 960	2117	864	1362	540	522	1063
∅ speedup		2.384		2.553	2.431	1.315

Table 6.8.: Results of Mandelbrot test for AltiVec and Cell BE.

The results for the different SSE machines can be seen in Table 6.7. At first glance, the measurements seem abnormal as they vary from 1.985 for the Core i7 to 6.598 for the Core2Duo E8400. However, if one takes a closer look, it is the scalar implementation that causes these speedup differences. On all test systems the SSE implementation is almost equally fast. The cores with higher clock rate perform slightly better. The E8400 (second column) has the highest performance gain due to the worst scalar execution.

AltiVec

The AltiVec version on the G4 achieves an average speedup of 2.384, which lies in the expected range. The measurements for the different image resolutions in Table 6.7 show a constant linear increase in the execution times for scalar and AltiVec execution. Even though the use of the `vmadd` instruction enables higher performance the additional operations from the control flow conversion reduce the performance gain.

Cell BE

The PPE implementation of the Mandelbrot computation shows the same performance benefits as the AltiVec version on the G4 with an average speedup of 2.553 (PPE tab in the Cell BE column of Table 6.8). Again the additional operations such as the creation of masks and the select operations reduce the performance gain.

size	Cell BE			
	scalar	**PPE**	**static**	**dynamic**
320 × 240	1392.2	686.8	1439.6	697.2
640 × 240	5556.2	2666	5587	2336.2
640 × 480	12496.4	5920.8	12462.8	5493.6
1280 × 480	22230	10511.6	22001.2	9167.4
1280 × 960	34722.4	16300.6	34216	14309
∅ speedup		2.093	0.997	2.299

Table 6.9.: Results of alternate Mandelbrot test on the Cell BE.

The results of the different SPE implementations might seem astounding at first. One would expect the dynamic version to be faster for this example, as the number of iterations of the inner loop varies for different stream elements. But with an average speedup of 2.431 the static version is 80% faster than its dynamic counterpart with only 1.315.

There are two reasons for this discrepancy: the threshold for the iteration depth is very low with only 500 and the iteration depth does not vary enough across the whole set. To investigate this claim further we have rerun the test with different input settings: the plane is now $Z = \{(x, y) : x \in [-2, -1.5], y \in [1, 1.5]\}$ and we have an increased maximum iteration count of 10000. The measurement results are kept in Table 6.9. Within this setting, the usefulness of dynamic load balancing shows. While the static load balancing can hardly reach the scalar execution time, the dynamic version achieves a performance gain by a factor of 2.299. As we can observe in Figure 6.1(b) the colors and thus the iteration sequence only differs in small parts of the image. This means that some of the SPEs terminate their computation very early while others compute for a long time. For the second setting the communication overhead of the dynamic distribution is more than compensated. Throughout all the other examples in this chapter this overhead cannot be amortized.

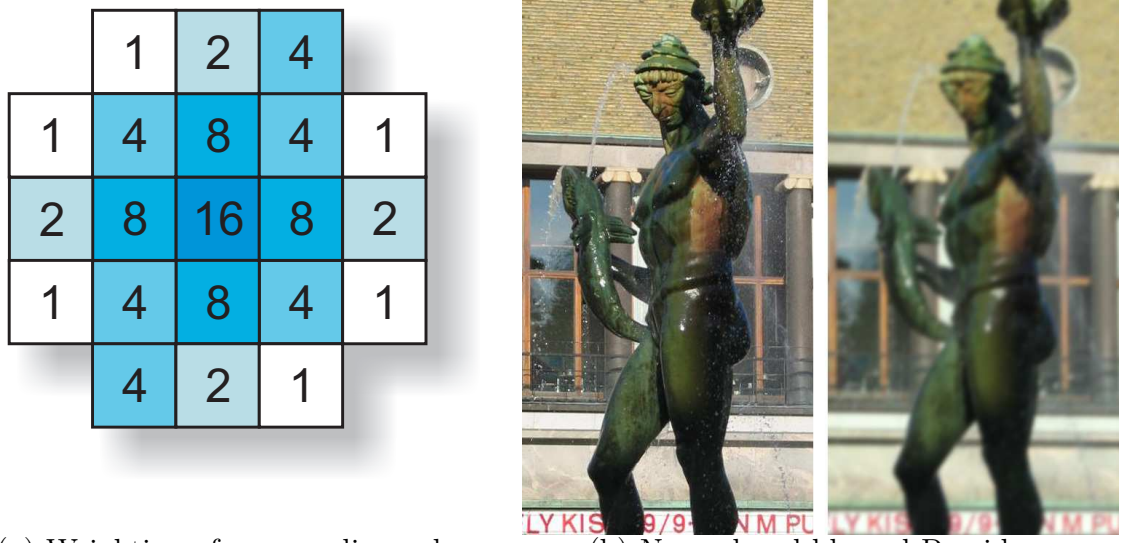
6.2.4. Gaussian Blur

Gaussian blur[21] describes the blurring of an image by a Gaussian function. It is a widely used effect in imaging software. Applying such a blur filter to an image is typically done to reduce image noise and detail. The visual effect of this blurring technique is a smooth blur resembling the view of the image through a translucent screen.

In a two-dimensional space, the Gaussian function looks like this

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

with σ being the standard derivation of the Gaussian distribution. This formula produces a surface whose contours are concentric circles with a standard normal distribution



(a) Weighting of surrounding colors. (b) Normal and blurred Poseidon.

Figure 6.2.: Weighted blur pattern and its application to the Gothenburg Poseidon.

from the center point. Values from this distribution are used to build a convolution matrix which is applied to the original image. Each pixel's new value is set to a weighted average of that pixel's neighborhood. The original pixel's color values have the highest Gaussian value, thus receiving the heaviest weight. The surrounding pixels receive smaller weights as their distance from the original pixel increases.

In practice such a convolution matrix is an approximation of the Gaussian values, and usually instead of the whole image only a small area around the current pixel is examined. An example of such a simplified matrix is shown in Figure 6.2(a). In Figure 6.2(b) a picture of the Poseidon statue in Gothenburg is shown as well as an application of the blur filter.

In Program 6.4 the CGiS implementation of the Gaussian blur is shown. The convolution matrix used complies with the one from Figure 6.2(a). The stream declaration of the image data is attached with a hint for SIMD architectures telling the CGiS compiler that the stream elements, though declared as `float3`, are individually aligned for SIMD processing, i. e. the data stream has been padded already. The algorithm starts with a set of gather operations followed by simple arithmetic combination of the gathered data.

Characteristics and Performance

In this example the loads of the adjacent pixel color values dominate the computation done. For architectures with cached memory the CGiS compiler uses loop sectioning to achieve a better cache performance. The experimental results are given in Table 6.10 and Table 6.11.

In the current version of the SIMD backend gathering operations and kernel flattening are mutually exclusive. Our bitmap library for loading and storing image data provides

```
PROGRAM gaussian;

INTERFACE

extern inout float3 IMAGE<BLUR_X,BLUR_Y> #HINT(SIMD:prealigned);

CODE

procedure blur (inout float3 ppixel)
{
  gather ppixel:
        tl<-1,-1>, t<0,-1>, tr<1,-1>, l<-1,0>, r<1,0>,
        bl<-1,1>, b<0,1>, br<1,1>, ttl<-1,-2>, tt<0,-2>,
        ttr<1,-2>, llt<-2,1>, ll<-2,0>, llb<-2,-1>, rrt<2,-1>,
        rr<2,0>, rrb<2,1>, bbl<-1,2>, bb<0,2>, bbr<1,2>;

  ppixel =  0.16 * ppixel
           +0.096 * (t + r + b + l)
           +0.048 * (tl + tr + bl + br + ttr + bbl)
           +0.024 * (tt + bb + rr + ll)
           +0.012 * (rrt + rrb + llt + llb + bbr + ttl);

  ppixel = ppixel min [ 1.0, 1.0, 1.0 ];
}

CONTROL

forall (ppixel in IMAGE)
  blur (ppixel);
```

Program 6.4: Gaussian blur.

padded loading of the pixel data from the hard disk into memory. The parallel execution here is done by working on complete pixels rather than single components for 4 different pixels in parallel.

All processor cores with caches show speedups of up to 2.8. If the cache optimization is turned off, the affected targets perform slightly worse than their scalar counterpart, underlining the importance of cache optimizations for neighboring operations. The characteristics of the Gaussian blur example are:

- ☐ Dominated by gathering operations.
- ☐ Few and simple floating-point arithmetics.
- ☐ Padded stream data.
- ☐ Optimizations applied: loop sectioning, removal of unnecessary boundary checks.

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
128 × 128	24	7	27	13	31	8	32	8
256 × 256	89	26	102	57	120	44	123	33
512 × 512	284	100	384	230	450	173	353	136
1024 × 1024	1105	484	1542	901	1784	689	1206	589
2048 × 2048	4455	1990	6175	3576	7185	2718	5145	2413
∅ speedup		2.857		1.784		2.837		2.829

Table 6.10.: Results of Gaussian blur test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
128 × 128	14	9	62	31	645	1252
256 × 256	53	34	258	142	1493	2801
512 × 512	220	133	1030	700	4511	8913
1024 × 1024	902	534	4124	2892	15878	33478
2048 × 2048	3713	2332	27603	12505	55002	110404
∅ speedup		1.597		1.774	0.252	0.127

Table 6.11.: Results of Gaussian blur test for AltiVec and Cell BE.

SSE

We can see in Table 6.10, that for all targets tested with gcc the SSE versions run 2.8 times faster than their respective scalar counterpart. Tests compiled with the icc show a performance gain of only 1.784 as visible in the third column. This results partly from the better scalar code and partly from the suboptimal SSE code generated here. For the largest image size of 2048^2 pixels scalar execution needs 6175ms for icc and 7185 for gcc on the E6300.

The three SSE test systems all feature cache hierarchies. To inspect the importance of the loop sectioning optimization we have run the tests with loop sectioning disabled. The measurements showed approximately equal times for scalar and SSE execution. Thus in a program that is dominated by memory accesses, CGiS implementations can only perform well with loop sectioning enabled.

AltiVec

The AltiVec version of the Gaussian blur runs little faster than scalar execution, namely 1.597 times faster. The measurement data for the G4 can be seen in Table 6.11. In most examples examined here AltiVec can profit from having a larger vector register set and so having less spill code. In the Gaussian blur algorithm though, each gathered value is only used once. This favors the SSE instruction set where one operand can be directly read from the memory. With AltiVec, on the other hand, it has to be loaded into a

register first.

Cell BE

The Gaussian blur algorithm shows the same performance gain for the AltiVec implementation on the PPE as on the G4. In the PPE tab in Table 6.11 the runtimes for the different image sizes can be seen. The average speedup is 1.774.

The timing measurements look very different on the SPEs, though. The slow DMAs undo the speedup of the parallel operations as there are not many arithmetic operations available to hide the latency of the DMAs. Also remember that the SPEs do not exhibit caches which is rendering loop sectioning redundant. This results in a performance loss compared to scalar execution. Even the smallest image of 64^2 pixels takes 1252ms for blurring with dynamic load balancing on the SPE versus only 62ms for the scalar version.

Summed up, this example demonstrates where the CGiS approach on the Cell BE fails. Load/store intensive algorithms can only be ported efficiently to the hardware if there are enough arithmetic operations to occupy the SPEs while the data is being waited for.

6.2.5. Black-Scholes

The Black-Scholes model is an option pricing model to compute the values of call and put options over a certain amount of time. An *option* is a contract between two parties giving one party the right but not the obligation to do something, usually the purchase or the sale of some asset. Because these rights to do something without being obligated to are valuable in themselves, they have to be bought by the option holders. There are two different kinds of options, namely *call options* and *put options*. Call options include the right to buy something, while put options entitle the holder to sell something. The buying and selling of options is a large part of the dealing on the Stock Exchange. It alleviates the risk of losing too much money as the option holder can decide not to buy or sell his asset. In their article "The Pricing of Options and Corporate Liabilities"[2], Fischer Black and Myron Scholes present the Black-Scholes model. With this model the pricing of call and put options can be determined.

In the Black-Scholes model the computation of the call option price consists of two parts. The term $S \cdot N(d_1)$ computes the benefit to be expected from buying a stock right now. The second part $K \cdot e^{-rt} \cdot N(d_2)$ gives the present value of paying the exercise price on the expiration day. The fair market value of the call option C and the value of the put option P are then computed by

$$
\begin{aligned}
C = &\quad S \cdot N(d_1) - K \cdot e^{-rt} \cdot N(d_2), \text{with} \\
P = &\quad K \cdot e^{-rt} - S + C \\
&\quad \text{with} \\
d_1 = &\quad \frac{log(S/K) + (r + \frac{s^2}{2})t}{\sigma\sqrt{t}} \text{and} \\
d_2 = &\quad d_1 - \sigma\sqrt{t}
\end{aligned}
$$

S hereby denotes the current stock price, t the time until option expiration, K the option striking price, r the risk-free interest rate and N the standard normal distribution. e is the exponential term, log the logarithm and σ the standard derivation. There are a few assumptions for the Black-Scholes model:

- ☐ During the option's life the stock pays no dividends.
- ☐ European exercise terms are used, meaning the option can only be exercised at its expiration date.
- ☐ Efficient markets. This means that the share prices follow a Markov process in continuous time.
- ☐ No charges for commissions.
- ☐ Constant interest rates.
- ☐ Returns are log-normally distributed.

Characteristics and Performance

This example is the one with most computations inside a static loop of which the maximum iteration count is passed by the main application and represents the number of simulation steps taken. The model has many different parameters to be explored. In our implementation we consider different tuples of (S, K, r, σ, t) and compute the call and put option price after the expiration time t. Exponential as well as logarithmic functions are used, the first one inside the main loop of the computation kernel. For the scalar implementation the libmath is used to compute the exponential function, for the SIMD targets a modified version the libsimdmath. The errors in precision can be seen after the fifth decimal place.

The results of the tests for SSE hardware are presented in Table 6.12 and for AltiVec hardware and Cell BE in Table 6.13. The number of simulation steps, the granularity of the approximation, is 10000; the number of sample call/put options computed is held in the leftmost column of the tables.

The characteristics of this example are

- ☐ Numerous calls to exponential and logarithmic functions.

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
512	1018	6	60	27	1645	7	1158	8
1024	2036	12	115	56	3264	14	2271	16
1536	3032	17	171	79	4890	21	3355	22
2048	4052	22	231	103	6524	29	4504	32
2560	5062	27	282	131	8151	36	5605	41
∅ speedup	175.508		2.163		224.340		140.009	

Table 6.12.: Results of Black-Scholes test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
512	1369	179	4419	332	193	755
1024	2741	358	8845	666	474	754
1536	4103	539	13284	1000	755	755
2048	5477	745	17691	1332	942	1503
2560	6848	895	22113	1666	1223	1504
∅ speedup	7.576		13.276		19.334	12.368

Table 6.13.: Results of Black-Scholes test for AltiVec and Cell BE.

- ☐ Static loop with many iterations.
- ☐ Complex arithmetics.
- ☐ Optimizations applied: inlining, control flow conversion, kernel flattening, scalarization.

SSE

On all targets using gcc and the Streaming SIMD extension the CGiS version of the Black-Scholes examples performs outstandingly. Taking a look at Table 6.12, the top performance gain is visible for the E6300 with 224.34. The main reason for this is the different precision for the floating-point computations that is used in the scalar implementation. This holds for the arithmetics as well as the exponential and logarithmic functions.

The third column shows the measurements for icc generated code. The scalar execution for a 512 calculated option prices only takes 60ms, while it takes 1645ms on the same machine when using the gcc. As discussed in the examination of the particle example already (Section 6.2.1) this is due to the use of scalar SSE instructions and a faster implementation of the exponential function for icc. Still, our implementation shows a speedup by more than a factor of 2.

AltiVec

The AltiVec version shows an average speedup by a factor of 7.576 taken from the G4 tab in Table 6.13. The benefits are not as high as on the SSE hardware because the PowerPC core features scalar 32-bit single-precision floating-point arithmetics. Apart from parallel execution the increased performance stems from the built-in square-root and reciprocal instructions in the AltiVec instruction set compared to standard scalar execution.

Cell BE

The third column in Table 6.13 holds the measurements for the Cell Broadband Engine for the four different execution modes. While the speedup on the PPE reaches an average of 13.276, the static distribution of the computations to the SPEs is ahead with a speedup of 19.334. Its dynamic complement shows a speedup of 12.368. Let us take a closer look at the results of the dynamic distribution in the rightmost column. For the first three input sizes the runtime measurements are almost identical (755ms). The same holds for the last two (1503ms). This is the result of the choice for the number of iterations done in one package: the default number of iterations assigned to one work list item is 256. This is half the size of the first input set. What happens is that the dynamic distribution issues only two SPE to do 256 iterations each and thus, only occupies two of six. The static distribution divides the iterations equally between all six SPEs. At the input size of 2048, the dynamic distribution generates 8 work list items, which means that 2 of the SPEs are issued twice and hence double the runtime compared to the smaller input sizes.

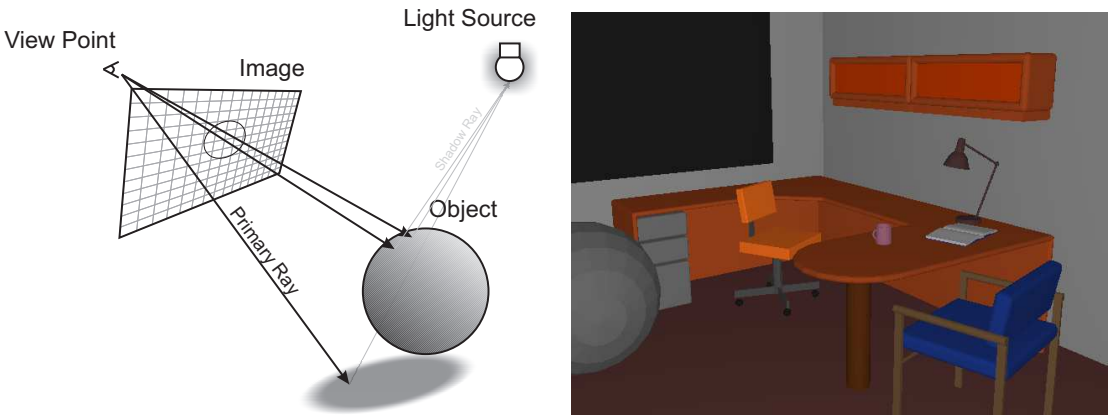
Here, the strengths of an implementation using SPEs become obvious: many complex operations done on a relatively small data set. The more self-sustaining SPEs can compute, the higher the performance gain.

6.2.6. Intersection

The last example presented here is part of a ray tracer. Ray tracing[16] is a method to generate realistic two-dimensional images from a three-dimensional scene description. The generated image can be seen as a window inside the scene and for each pixel of the image a light ray is cast into the scene, broken and reflected depending on the surfaces it hits and the light sources. A simple overview of how ray tracing works can be seen in Figure 6.3(a).

Ray tracing starts by casting the primary rays or view rays into the scene and checks which objects are hit by them. From the hit points, rays are shot in direction of the light sources to determine the illumination of the point, so called shadow rays. Depending on the kind of surface material, reflection or refraction rays may also be cast. When all influences are considered the final color of the point can be determined taking the properties of the hit surface into account.

Usually three-dimensional objects in such scene descriptions are broken down into



(a) Ray tracing overview (b) Sample scene

Figure 6.3.: Ray tracing

triangles. Whether one of the rays cast into the scene hits an object can be computed by algorithms that intersect rays with triangles such as the Möller-Trumbore intersection algorithm[38].

A ray is given by $R(t) = O + tD$ with O being the origin and D the direction vector. A point $T(u, v)$ on a triangle T is determined by $T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$. u and v are barycentric coordinates. For $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ and $T = O - V_0$ after a series of transformations the intersection point is computed by

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

with $P = (D \times E_2)$ and $Q = (T \times E_1)$. If the intersection point is on the triangle $0 \leq u \leq 1$, $0 \leq v \leq 1$ and $u + v \leq 1$ hold.

Characteristics and Performance

This example stems from the computer graphics domain and contains vector computations and transformations, such as cross products and dot products.

For this example we implemented a brute force algorithm that checks the list of primary rays against all triangles in the scene. Our scene is an office space as shown in Figure 6.3(b). It contains 33952 triangles and the number of primary rays varies with the test cases from 64^2 pixels to 256^2 pixels. The relatively small numbers were chosen to still have acceptable execution times on the slower machines like the G4.

The triangle data – the coordinates of the corner points of the triangles – have not been padded for this example. The re-arrangement of data is done completely by local reordering and data are processed by flattened kernels. In summary, the characteristics of the ray-triangle intersection are

- ☐ Vector arithmetics, like dot and cross product.

size	E8400		E6300 (icc)		E6300 (gcc)		Core i7	
	scalar	SSE	scalar	SSE	scalar	SSE	scalar	SSE
64 × 64	3485	2000	4981	2228	6654	3598	3571	2614
128 × 64	7053	3983	9969	4422	13479	7125	7191	5372
128 × 128	13925	7948	19909	8907	26582	14230	14223	10518
256 × 128	28139	15972	39842	17727	53825	28433	28802	21196
256 × 256	56367	31794	79707	35600	106275	57240	56187	42909
∅ speedup		1.760		2.242		1.872		1.345

Table 6.14.: Results of intersect test for SSE hardware.

size	G4		Cell BE			
	scalar	AltiVec	scalar	PPE	static	dynamic
64 × 64	42659	14024	17423	11282	6586	7527
128 × 64	85140	28144	35250	22606	13165	15048
128 × 128	170735	56416	69824	45214	26638	27584
256 × 128	342133	112653	141236	90491	53270	55160
256 × 256	547687	225131	301913	180653	106847	107814
∅ speedup		2.913		1.576	2.640	2.466

Table 6.15.: Results of intersect test for AltiVec and Cell BE.

☐ Looped lookups of triangle data.

☐ Brute force approach.

☐ Optimizations applied: inlining, control flow conversion, kernel flattening.

SSE

Table 6.14 shows the results of the execution using SSE hardware. The average speedup on targets with gcc created binaries varies between 1.345 for the Core i7 and 1.872 for the E6300. For icc it is even 2.242. The overall performance gain is decent but not as high as expected. The CGIS implementation iterates over streams of primary rays and each ray is intersected against each triangle. One problem reducing the performance is the inner loop iterating over the triangles. Its counter cannot be kept in a scalar register, as its value is passed to the outside as the id of the triangle hit. Therefore, the loop counter must be kept in a SIMD register which results in slightly slower checks via the intrinsic `_mm_movemask_ps`.

A further improvement of the scalarization optimization could introduce a second counter kept in a scalar register. The scalar counter could be used for the branch check and the SIMD one for the passing of the id. Also, some of the select operations could be eliminated as they are not needed for scalar loop counters. In a handwritten adaptation

of the SIMD code a performance improvement of roughly 7% could be observed leaving this adaptation for future work on the compiler.

AltiVec

Results for the G4 can be seen in Table 6.15. The performance gain is 2.913 which is slightly better than the SSE versions. Here as well, an amplification of the scalarization would improve the performance gain.

Cell BE

The intersection example on the PPE shows an average speedup by a factor of 1.576 as observable in the rightmost column of Table 6.15 under the PPE tab. Compared to the G4 scalar execution, it is much faster, which is surprising, as floating-point operations in general seem slower on the PPE than on the MPC7450. Here the newer gcc produces faster code for the scalar execution on the PPE, impairing the performance gain of the parallel execution.

For the SPEs the static distribution is slightly faster than the dynamic with 2.640 to 2.466. This is to be expected, as the number of iterations in the inner loop is constant. Also with every initialization of an SPE, the complete triangle data need to be copied into the local storage for the lookup operation. These two properties of the CGiS implementation on SPEs make the static work load distribution a better choice for this intersection algorithm.

6.3. Interpretation of Performance Results

The examples investigated here are a representative set of data-parallel algorithms and allow to point out the strengths and the weaknesses of the SIMD backend of the CGiS compiler.

An overview presenting the average speedup of targets featuring the Streaming SIMD Extensions over the scalar implementation is given in Figure 6.4. Note that the scale of the Y-axis is logarithmic. The results for targets with AltiVec support, namely the G4 and the PPE, and for the two different load balancing methods on the SPEs can be found in Figure 6.5[3]. Again, the scaling of the Y-axis is logarithmic. The single core targets, i. e. the SSE hardware, AltiVec on the G4 and AltiVec on the PPE, exhibit 4-way SIMD execution. However, for some examples the performance gain was even higher than the expected factor of 4. This is possible due to poor scalar floating-point performance or special SIMD operations that allow combined computations of a sequence of scalar operations. The toy particle system shows the best average speedups on all targets ranging from a factor of 6.8 up to a factor of 157.7. The speedups of the computation of the

[3]For the Gaussian test the "speedup" on the SPEs is below 0.5, which why they are not visible in this chart.

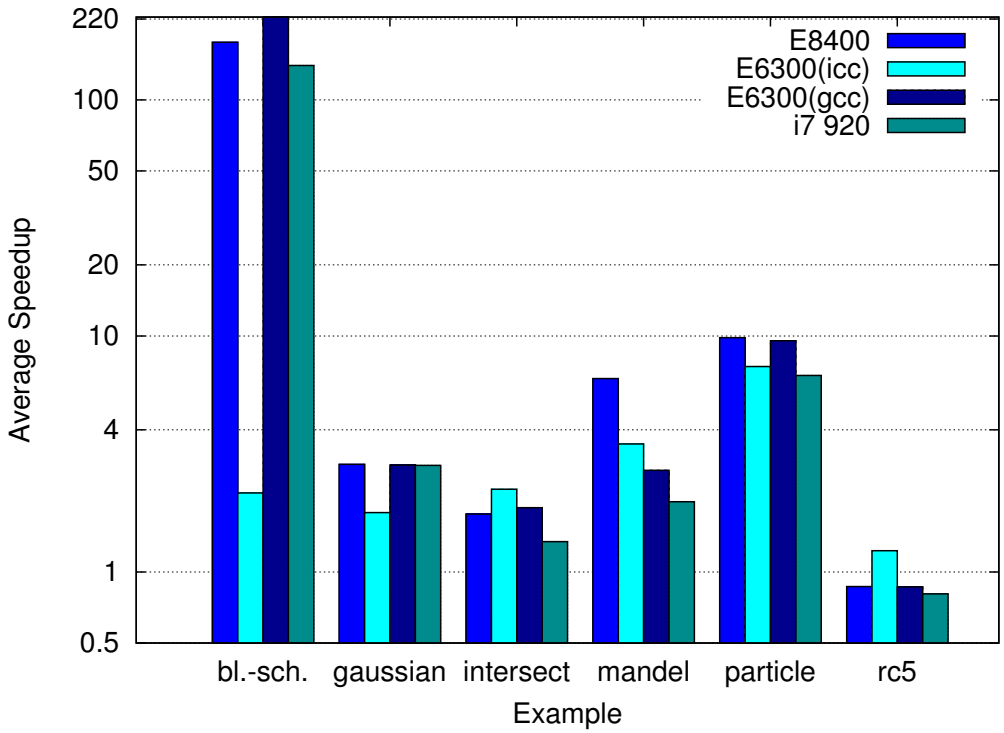


Figure 6.4.: Speedups of SIMD backend over sequential implementations on SSE hardware.

Mandelbrot set and the ray-triangle-intersection are distributed around a factor of 3 which meets the expectations. The rc5 block cipher encryption performs well on AltiVec machines due to the built-in rotation operation. The best average speedup is on the PPE with 7.5. The Gaussian blur test is an example for a memory dominated algorithm. Although not performing generally well it shows good performance gains on SSE hardware with a speedup factor of 2.8. The Black-Scholes option pricing model exhibits by far the best speedup with a factor of 224 on the E6300. This exceptional performance gain though can only be reached in special circumstances: very high arithmetic density and data locality together with slow scalar floating-point operations and a fast SIMD mathematical library are needed. All in all, the results show good speedups making an implementation in CGiS worth the effort.

Aside from the performance of the CGiS implementations, it is worth mentioning that overall icc generates more efficient code than gcc for scalar and SIMD execution, making it a better choice when compiling for SSE hardware.

Unfortunately, the SPE implementations of most examples is not as efficient as expected. With six SPEs available and SIMD execution similar to AltiVec, a theoretical maximum performance gain of 24 is possible leaving combined operations aside. Except for the particle system simulation, the speedup remains behind these expectations. There are various reasons for this:

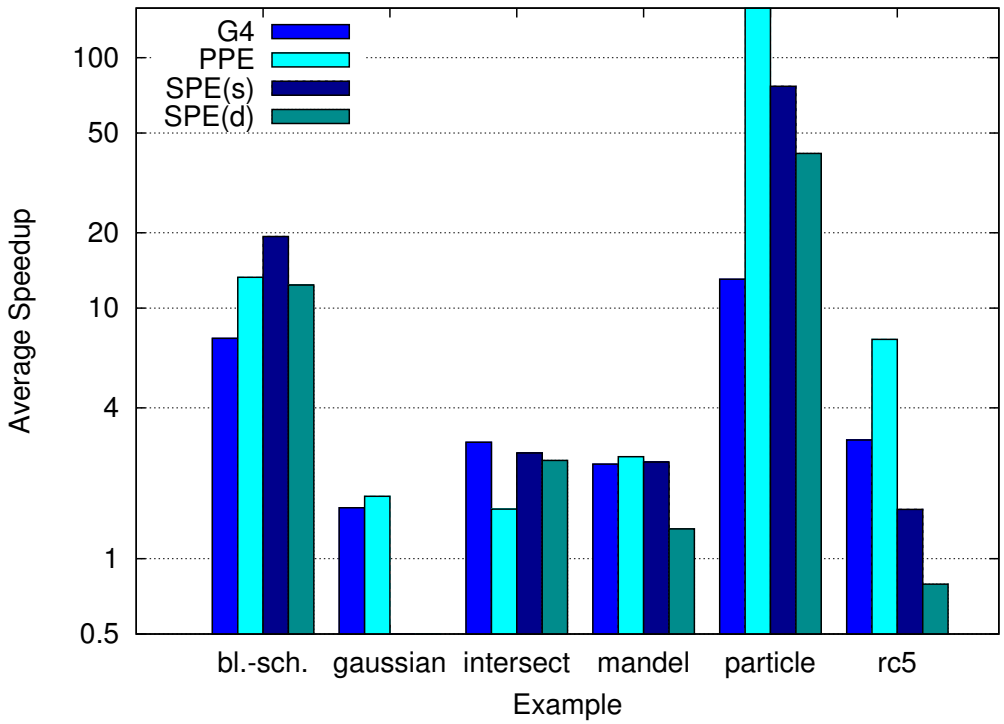


Figure 6.5.: Speedups of SIMD backend over sequential implementations on the G4 and the Cell BE.

- ☐ The SIMD backend for the Cell BE is in a prototypical state. In our approach, communication between PPE and SPE works via threads. Threaded execution seems suitable but especially for the examples where dynamic distribution of the computations is profitable, the usage of alternate communication methods, such as mailboxing, might increase the performance.

- ☐ The SPEs are very sensitive to the code layout and all of this is left to the supplied gcc of the Yellow Dog Linux distribution at the moment, leaving out many optimization opportunities.

Still, the SPE implementations show good overall performance gains underlining the viability of the approach using CGiS for multi-core architectures.

The most prominent weakness of the SIMD backend, or rather the SIMD applications on the examined targets, are the neighboring operations. By exploiting caches, even here a speedup can be achieved, but in general algorithms with gather operations do not perform well on the SIMD hardware. For the SPE implementations, gathering is slow, as possibly lots of data have to be uploaded to the SPE. Many of the examples presented for the GPU backend in [32] contain gather operations. They result in fast and efficient GPU applications, as the neighboring can be done in hardware, but in poor SIMD applications. Another interesting feature of CGiS, the possibility to directly visualize the results of the computations, is limited to the GPU backend.

6.3.1. Comparison to GPUs

When taking a closer look at the examples and their characteristics, we can see that with more memory accesses the speedup decreases. For certain algorithms the loop sectioning optimization and elimination of boundary checks for neighboring operations ease the high memory latency but do not really make those applications worth rewriting them in CGiS for SIMD execution. The preferred targets for memory dominated algorithms would be GPUs where the very fast texture memory is designed for such operations.

Let us examine the differences in performance between SIMD CPUs and GPUs in more depth. Current graphics hardware exhibits much more parallel computation power than any CPU with SIMD units, yet there are applications that profit less from GPUs, especially the ones with smaller data amounts to be processed. This is mainly because of the comparatively high setup costs for starting GPU execution. For older generations this has to be done through the OpenGL driver which can be seen as black box that starts pushing data whenever it has buffered enough. In any case, data have to be copied from the main memory of the host into the texture memory of the graphics board and back, which is expensive.

The computation of the Mandelbrot set in Section 6.2.3 was done with the same configuration as in [32]. For the two smallest input sizes, the SSE version on the E6300 is faster than running on the same host with a GTX8800. Though the computations run much faster combined with the setup and writeback time, the pure SIMD version is ahead. The same holds for the particle system. Whenever integer support is needed, as in the rc5 encryption, at least G80 GPUs must be used as on older graphics hardware integer computation is not included. Here, for example, the PPE beats every G80 implementation due to AltiVec. Even the old G4 is faster for data set sizes smaller than 2^{22} words.

6.4. Future Work

The SIMD backend of the CGiS compiler contains transformations to enable SIMDization and many optimizations to speed up SIMD code. But its full potential has not yet been reached. The examples show that scalarization is not fully exploited yet. Running a SIMD counter and a scalar one in parallel increased the average performance of the intersection algorithm by another 7%, also enabling the discarding of superfluous select and mask combining operations.

While the Cell BE backend shows promising results, there are still a couple of optimization opportunities left out. For example, the communication between SPEs and PPE needs to be investigated further. Also, the distribution of instructions to the two different pipelines of the SPEs should be improved. As the output of the CGiS compiler is restricted to intrinsics at the moment, the direct assignment of instructions to certain addresses (and thus pipelines) cannot be approached. We believe that going a step ahead and generating assembly output could further improve the code quality and the performance of the applications generated with `cgisc`.

As an example to explain the deficiency of the generated SPE code, consider the following excerpt from the particle example:

```
153c:       34 00 80 83     lqd     $3,32($1) # odd
1540:       34 00 80 82     lqd     $2,32($1) # even
1544:       37 00 01 02     frest   $2,$2
1548:       7a 80 81 82     fi      $2,$3,$2
```

This snippet is part of the loop computing the velocities of the particles. The instruction `frest` estimates the reciprocal of its operand and `fi` is an interpolation instruction used to improve the result of the estimation. However, both instructions can only be executed by the even pipeline while both operand loads can only be completed by the odd pipeline. In SPEs, instructions are always fetched in pairs. Instructions at even addresses, the last two bits of the address are zero, are favored by the even pipeline; inversely for the odd pipeline. If an instruction pair matches the pipeline requirements it can be *dual-issued*. Executing instructions at even addresses in the odd pipeline induces a small runtime penalty for switching. In the schedule above the second quad-word load (`ldq`) is set on an even address although it must be processed by the odd pipeline and vice versa for `frest`. Without violating data-dependencies, the instructions can be rearranged to

```
153c:       34 00 80 82     lqd     $2,32($1) # odd
1540:       37 00 01 02     frest   $2,$2     # even
1544:       34 00 80 83     lqd     $3,32($1)
1548:       7a 80 81 82     fi      $2,$3,$2
```

removing all penalties. More of these patterns can be found in gcc generated code for all examples running on the SPEs.

Another problem with the provided gcc for SPEs is the missing inlining. Though specified, gcc produces calls to all runtime library functions, which are inlined on other targets. This also results in a performance loss as is induces additional control flow.

While generating intrinsics shows excellent performance gains on SSE architectures and AltiVec architectures, it is not sufficient to fully access the potential of the SPEs – at least not with the examined version of the Gnu C Compiler. Moving away from intrinsics and towards assembly code generation would be one possibility to address these problems in CGiS. By this, the CGiS compiler would have full control over the code layout, profiting the Cell BE backend very much.

CHAPTER 7

Conclusion

The focus of this thesis is efficient code generation for SIMD hardware and multi-core architectures with the CGiS compiler framework. Compilers for general purpose programming languages often fail at automatic SIMDization. Uncovering parallelism in sequential programming languages is a demanding task and many vectorization techniques have been employed to exploit this parallelism. Yet the success of these approaches remains marginal. Data dependencies can avoid vectorization and in connection with the restrictions imposed by SIMD hardware in modern CPUs such as memory alignment, SIMDization can become almost impossible for standard compilers. Direct programming of SIMD hardware, on the other hand, requires the developer to have detailed knowledge and has other disadvantages like bad portability and bad maintainability.

Our approach bypasses some obstacles of vectorization by exerting a data-parallel language with explicit parallelism, CGiS. We designed CGiS after carefully examining the characteristics of data-parallel algorithms. The language CGiS provides a high abstraction level hiding all hardware intricacies and still assists experienced programmers to further improve code quality. The programs are clearly structured and the kernel language is close to C. With `cgisc` we presented an optimizing compiler for CGiS programs. We have implemented backends for various GPUs, the most relevant SIMD instruction sets, namely Intel's Streaming SIMD extensions and Freescale's AltiVec, and the Cell Broadband Engine.

CGiS exhibits two kinds of parallelism: parallel loop iterations and small vector data types. Mapping these types of parallelism to SIMD hardware requires kernel transformations to enable exploitation of superword level parallelism. This is done with kernel flattening by breaking down vectorial variables and operations to scalar ones. This program transformation can be applied to any language featuring small vector data types. Many data parallel algorithms contain neighboring or gathering operations which access additional stream data according to a certain pattern. To speed up data accesses, SIMD CPUs exhibit multi-level caches. Exploiting caches in CGiS is accomplished by a form of loop sectioning. It modifies the iteration sequence over streams according to the access

patterns of the gathering operations. In the context of SIMDization and especially with kernel flattening, full control flow conversion is indispensable. If- and loop-conversion enables the CGiS compiler to parallelize the execution of flattened kernels while preserving the program semantics. With SIMDization it can still be useful to keep certain variables in scalar registers, such as loop counters. This optimization is also integrated in `cgisc` and is called scalarization. All of these transformations and optimizations require knowledge about the program properties that can be acquired by data-flow analyses. To ease the generation of program analyses for CGiS programs, an interface to PAG has been implemented. All program analyses, we identified indispensable for efficient SIMD code generation, have been generated by PAG using that interface. These are the live variable analysis, the reaching definitions analysis and the invariant analysis. The latter determines variables suitable for scalarization.

To further explore the usability of CGiS we implemented a prototypical backend for IBM's Cell Broadband Engine. As a multi-core architecture, it required additional efforts to enable generation of efficient applications. The SPEs can be controlled independently by threads. The distribution of computations to the SPEs can be static or dynamic, i. e. by use of a worklist.

The SIMD backend has been tested on a set of representative target machines. The sample applications used for our experiments are specific examples of the field of data-parallel algorithms. The execution times of the generated applications have been compared to similar hand-written C++ implementations and show excellent speedups. For the usage of CGiS programs we demand high arithmetic density, which all the investigated examples exhibit. The best speedup is gained for the Black-Scholes option pricing example on the Core2Duo E6300 with a factor of 224. This extreme performance gain can only be achieved under certain conditions: very high arithmetic density and slow floating-point operations for scalar execution together with the excessive usage of mathematical library functions. On the other hand, the Gaussian blur example showed where the CGiS approach is not successful on the Cell BE. This memory dominated algorithm runs 4 times faster in standard execution than on the SPEs in parallel. On average a speedup by a factor of 3 is observable. Considering these performance gains, this underlines the viability of this approach. The programming language CGiS is fit for implementing data-parallel algorithms on CPUs with SIMD units and multi-core architectures.

The CGiS project ends with this work. Its original purpose was to make unused parallel computation power in common PCs accessible to the average programmer. It is safe to say that together with [32] the goal has been achieved: GPUs, SIMD CPUs and multi-core architectures have been successfully explored. To the best of our knowledge, this is the first compiler to support this variety of parallel hardware.

APPENDIX A

Selected SIMD Instructions and Intrinsics

A.1. Selected Streaming SIMD Extension instructions

The instructions presented here are a subset of the available SSE, SSE2, SSE3, SSSE3 and SSE4 instructions. Most of them have been referenced in the thesis and are grouped into five different categories.

- ☐ Data transfer instructions
- ☐ Arithmetic instructions
- ☐ Logical instructions and comparison instructions
- ☐ Shuffle and unpack instructions
- ☐ Conversion instructions

Most instructions operate on two operands (two-address instructions) except for the data transfer instructions and shuffles. Each instruction also exists in a scalar mode which is denoted by the suffix "ss" instead of "ps". For floating-point computations, these instructions can be used to speed-up scalar computations as they do not require alignment and directly operate in IEEE single-precision floating point mode.

The intrinsics of the SSE instructions operating on integer values are suffixed with "epi32". The logical bitwise operations are denoted by a "si128" suffix. Alike AltiVec there is no 32-bit integer multiplication. It can be simulated by converting the operands to floating-point values, multiplying those and then converting them back to integer.

A.1.1. SSE Data Transfer Instructions

All the memory operations presented here required the memory location to be 16-byte aligned. With the Core i7 architecture Intel has removed this restriction and the locations can be an arbitrary memory address. Unaligned loads were already possible with the SSE2 instruction `MOVDQU` but it was significantly slower than its aligned counterpart.

Mnemonic	Intrinsic	Meaning
MMVAPS	_mm_[load|store]_ps	Move 4 aligned packed single-precision floating-point values between XMM registers or between XMM registers and memory
MOVUPS	_mm_[load|store]_ps	Move 4 unaligned packed single-precision floating-point values between XMM registers or between XMM registers and memory
MOVHPS	_mm_[load|store]h_ps	Move 2 packed single-precision floating-point values to an from the high quadword of an XMM register and memory
MOVHLPS	_mm_movehl_ps	Move 2 packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of an XMM register
MOVLPS	_mm_[load|store]l_ps	Move 2 packed single-precision floating-point values to an from the low quadword of an XMM register and memory
MOVLHPS	_mm_movelh_ps	Move 2 packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of an XMM register
MOVMSKPS	_mm_movemask_ps	Extract sign mask of 4 single-precision floating-point values
MOVDQA	_mm_[load|store]_si128	Move an aligned double quadword between XMM registers or between XMM registers and memory
MOVDQU	_mm_[load|store]u_si128	Move an unaligned double quadword between XMM registers or between XMM registers and memory

A.1.2. SSE Packed Arithmetic Instructions

Mnemonic	Intrinsic	Meaning
ADDPS	_mm_add_ps	Add packed single-precision floating-point values
SUBPS	_mm_sub_ps	Subtract packed single-precision floating-point values
MULPS	_mm_mul_ps	Multiply packed single-precision floating-point values
DIVPS	_mm_div_ps	Divide packed single-precision floating-point values
RCCPS	_mm_rcc_ps	Compute reciprocals of packed single-precision floating-point values
SQRTPS	_mm_sqrt_ps	Compute square roots of packed single-precision floating-point values
RSQRTPS	_mm_rsqrt_ps	Compute reciprocals of square roots of packed single-precision floating-point values
MAXPS	_mm_max_ps	Return maximum packed single-precision floating-point values
MINPS	_mm_min_ps	Return minimum packed single-precision floating-point values
PADDD	_mm_add_epi32	Add packed 32-bit integer values

A.1.3. SSE Comparison and Logical Instructions

Mnemonic	Intrinsic	Meaning
CMPPS	_mm_cmpx_ps[1]	Compare packed single-precision floating-point values
ANDPS	_mm_and_ps	Perform bitwise logical AND of packed single-precision floating-point values
ORPS	_mm_or_ps	Perform bitwise logical OR of packed single-precision floating-point values
XORPS	_mm_xor_ps	Perform bitwise logical XOR of packed single-precision floating-point values
ANDNPS	_mm_andnot_ps	Perform bitwise logical AND NOT of packed single-precision floating-point values

In contrast to AltiVec there is no special select instruction for SSE. However, the select needed for if-conversion can be simulated as as series of locical operations:

```
inline __m128 _mm_sel_ps(__m128 a, __m128 b, __m128 mask)
{
  return _mm_or_ps(_mm_andnot_ps(mask,a), _mm_and_ps(b,mask));
}
```

A.1.4. SSE Shuffle and Unpack Instructions

Mnemonic	Intrinsic	Meaning
SHUFPS	_mm_shuffle_ps	Shuffles values in packed single-precision floating-point operands
UNPCKHPS	_mm_unpackhi_ps	Unpacks and interleaves the two high-order values from two single-precision floating-point operands
UNPCKLPS	_mm_unpacklo_ps	Unpacks and interleaves the two low-order values from two single-precision floating-point operands

A.1.5. SSE Conversion Instructions

Mnemonic	Intrinsic	Meaning
CVTPS2PI	_mm_cvtps_epi32	Convert packed single-precision floating-point values to packed doubleword integers
CVTTPS2PI	_mm_cvttps_epi32	Convert with truncation packed single-precision floating-point values to packed doubleword integers
CVTSS2SI	_mm_cvtss_si32	Convert a scalar single-precision floating-point value to a doubleword integer

A.2. Selected AltiVec Instructions

The selection of AltiVec instructions listed here can be categorized into the following sets:

- ☐ Data transfer instructions
- ☐ Arithmetic instructions
- ☐ Logical and comparison instructions
- ☐ Permute and unpack instructions
- ☐ Conversion instructions

The complete listing of AltiVec intrinsics can be found in [12]. Unlike SSE most of the AltiVec intrinsics are not limited to a certain data type, e. g. `vec_add` can be used with integers of various widths and floating-point as well. The compiler then maps them to different instructions. AltiVec supports the follwing data types for most operations:

- ☐ 8-bit integer (signed and unsigned)

- ☐ 16-bit integer (signed and unsigned)
- ☐ 32-bit integer (signed and unsigned)
- ☐ 32-bit floating-point

All data types are kept in the same vector registers, i. e. there are no special floating-point registers.

A.2.1. AltiVec Data Transfer Instructions

Mnemonic	Intrinsic	Meaning
lvx	vec_ld	Perform a 16-byte load at a 16-byte aligned address
lvxl	vec_ldl	Perform a 16-byte load at a 16-byte aligned address and mark the cache line as least-recently-used
stvx	vec_st	Perform a 16-byte store at a 16-byte aligned address
stvxl	vec_stl	Perform a 16-byte store at a 16-byte aligned address and mark the cache line as least-recently-used

A.2.2. AltiVec Arithmetic Instructions

Mnemonic	Intrinsic	Meaning
vadduwm	vec_add	Add four 32-bit integer elements
vaddfp	vec_add	Add four 32-bit floating-point values
vmaddfp	vec_madd	Multiply-add four floating-point elements (floating-point only)
vmulesh	vec_mule	Even multiply of four 16-bit integer alements, results are 32-bit wide (integer only)
vrefp	vec_re	Reciprocal estimate of four floating-point elements (floating-point only)
vrsqrtefp	vec_rsqrt	Reciprocal square root estimate of four floating-Point elements
vrlw	vec_rl	Left rotate of four 32-bit integer elements
vrfin	vec_round	Round to nearest of four floating-point integer elements

A.2.3. AltiVec Logical and Comparison Instructions

Mnemonic	Intrinsic	Meaning
vcmpgtfp	vec_cmpgt	Compare greater-than of four floating-point elements
vcmpgtsw	vec_cmpgt	Compare greater-than of four signed, 32-bit integer elements
vand	vec_and	Logical bitwise AND
vor	vec_or	Logical bitwise OR
vxor	vec_xor	Logical bitwise XOR
vor	vec_nor	Logical bitwise NOR
vsel	vec_sel	Bit-wise conditional select of vector contents

A.2.4. AltiVec Permute and Unpack Instructions

Mnemonic	Intrinsic	Meaning
vperm	vec_perm	Permute sixteen byte elements
vpkuwum	vec_pack	Pack eight unsigned, 32-bit integer elements to eight unsigned, 16-bit integer elements

A.2.5. AltiVec Conversion Instructions

Mnemonic	Intrinsic	Meaning
vcfsx	vec_ctf	Convert four signed, 32-bit integer elements to four floating-point elements
vctsxs	vec_cts	Convert four floating-point elements to four saturated signed, 32-bit integer elements
vctuxs	vec_ctu	Convert four floating-point elements to four saturated unsigned, 32-bit integer elements

APPENDIX B

SSE Intrinsics Generated for the Alpha Blending Example

```
static void blend_f_SSE (__m128 *ipixel_x_17, __m128 *ipixel_y_18,
                         __m128 *ipixel_z_19, __m128 *bg_x_20,
                         __m128 *bg_y_21,    __m128 *bg_z_22,
                         __m128 *alpha_x_23)
{
  // local declarations
  __m128 ipixel_x_17_local = *ipixel_x_17;
  __m128 ipixel_y_18_local = *ipixel_y_18;
  __m128 ipixel_z_19_local = *ipixel_z_19;
  __m128 bg_x_20_local = *bg_x_20;
  __m128 bg_y_21_local = *bg_y_21;
  __m128 bg_z_22_local = *bg_z_22;
  __m128 alpha_x_23_local = *alpha_x_23;

  // data reordering:
  __m128 ipixel_x_17_local0;
  ipixel_x_17_local0 = _mm_shuffle_ps(ipixel_y_18_local,
                                      ipixel_z_19_local,
                                      _CGIS_SHUFFLE(2,0,1,0));
  ipixel_x_17_local0 = _mm_shuffle_ps(ipixel_x_17_local,
                                      ipixel_x_17_local0,
                                      _CGIS_SHUFFLE(0,3,0,2));

  __m128 ipixel_y_18_local1;
  ipixel_y_18_local1 = _mm_shuffle_ps(ipixel_x_17_local,
                                      ipixel_z_19_local,
                                      _CGIS_SHUFFLE(1,0,2,0));
  ipixel_y_18_local1 = _mm_shuffle_ps(ipixel_y_18_local1,
                                      ipixel_y_18_local,
                                      _CGIS_SHUFFLE(0,2,0,3));
  ipixel_y_18_local1 = _mm_shuffle_ps(ipixel_y_18_local1,
                                      ipixel_y_18_local1,
                                      _CGIS_SHUFFLE(0,2,3,1));
```

```
__m128 ipixel_z_19_local2;
ipixel_z_19_local2 = _mm_shuffle_ps(ipixel_x_17_local,
                                    ipixel_y_18_local,
                                    _CGIS_SHUFFLE(2,0,1,0));
ipixel_z_19_local2 = _mm_shuffle_ps(ipixel_z_19_local2,
                                    ipixel_z_19_local,
                                    _CGIS_SHUFFLE(0,2,0,3));

//combining intermediate results of reordered data
ipixel_x_17_local = ipixel_x_17_local0;
ipixel_y_18_local = ipixel_y_18_local1;
ipixel_z_19_local = ipixel_z_19_local2;

__m128 bg_x_20_local0;
bg_x_20_local0 = _mm_shuffle_ps(bg_y_21_local, bg_z_22_local,
                                _CGIS_SHUFFLE(2,0,1,0));
bg_x_20_local0 = _mm_shuffle_ps(bg_x_20_local, bg_x_20_local0,
                                _CGIS_SHUFFLE(0,3,0,2));

__m128 bg_y_21_local1;
bg_y_21_local1 = _mm_shuffle_ps(bg_x_20_local, bg_z_22_local,
                                _CGIS_SHUFFLE(1,0,2,0));
bg_y_21_local1 = _mm_shuffle_ps(bg_y_21_local1, bg_y_21_local,
                                _CGIS_SHUFFLE(0,2,0,3));
bg_y_21_local1 = _mm_shuffle_ps(bg_y_21_local1, bg_y_21_local1,
                                _CGIS_SHUFFLE(0,2,3,1));

__m128 bg_z_22_local2;
bg_z_22_local2 = _mm_shuffle_ps(bg_x_20_local, bg_y_21_local,
                 _CGIS_SHUFFLE(2,0,1,0));
bg_z_22_local2 = _mm_shuffle_ps(bg_z_22_local2, bg_z_22_local,
                 _CGIS_SHUFFLE(0,2,0,3));

//combining intermediate results of reordered data
bg_x_20_local =  bg_x_20_local0;
bg_y_21_local =  bg_y_21_local1;
bg_z_22_local =  bg_z_22_local2;

// code for blend_f
__m128 _temp_0_x_24;
__m128 _temp_0_y_25;
__m128 _temp_0_z_26;
_temp_0_x_24 = _mm_mul_ps(alpha_x_23_local, ipixel_x_17_local);
_temp_0_y_25 = _mm_mul_ps(alpha_x_23_local, ipixel_y_18_local);
_temp_0_z_26 = _mm_mul_ps(alpha_x_23_local, ipixel_z_19_local);
__m128 _temp_1_x_27;
_temp_1_x_27 = _mm_sub_ps(1, alpha_x_23_local);
__m128 _temp_2_x_28;
__m128 _temp_2_y_29;
__m128 _temp_2_z_30;
_temp_2_x_28 = _mm_mul_ps(_temp_1_x_27, bg_x_20_local);
```

```
  _temp_2_y_29 = _mm_mul_ps(_temp_1_x_27, bg_y_21_local);
  _temp_2_z_30 = _mm_mul_ps(_temp_1_x_27, bg_z_22_local);
  bg_x_20_local = _mm_add_ps(_temp_0_x_24, _temp_2_x_28);
  bg_y_21_local = _mm_add_ps(_temp_0_y_25, _temp_2_y_29);
  bg_z_22_local = _mm_add_ps(_temp_0_z_26, _temp_2_z_30);

  // storing inout/out parameters
  // returning bg_x_20 to its original ordering
  // returning bg_y_21 to its original ordering
  // returning bg_z_22 to its original ordering
  // ...

  //combining intermediate results of reordered data
  bg_x_20_local =  bg_x_20_local0;
  bg_y_21_local =  bg_y_21_local1;
  bg_z_22_local =  bg_z_22_local2;

  *bg_x_20 = bg_x_20_local;
  *bg_y_21 = bg_y_21_local;
  *bg_z_22 = bg_z_22_local;
}
```

APPENDIX C

Code Generated by icc and gcc for the Particle Example

The particle example discussed in Section 6.2.1 showed remarkable performance differences for the SSE code generated from the instrinsics of the flattened kernel. Here we take a closer look at these differences. All the assembly listings here are in AT&T syntax. In table C.1 a part of the assembly code for the computations before the inner loop is shown. The left column shows gcc generated code while the right one shows the code generated by icc. Intel's C compiler was able to move the division above the loop body as it is invariant. The shorter number of instructions also shows a much better register allocation. Table C.2 holds the inner loop of the procedure `sim` as in Program 6.1. Again the code in the left column produced by gcc contains more stack operations and the divisions were not moved out of the loop decreasing the performance.

```
movaps  _ZL13simd_temp3_58, %xmm2
movaps  %xmm7, 400(%esp)
movaps  _ZL13simd_temp4_59, %xmm1
movaps  %xmm4, 384(%esp)
movaps  _ZL13simd_temp5_60, %xmm0
movaps  %xmm2, 688(%esp)
movaps  %xmm1, 672(%esp)
movaps  %xmm0, 656(%esp)
jle     .L19
movaps  464(%esp), %xmm7
xorl    %eax, %eax
movaps  448(%esp), %xmm6
movaps  %xmm7, 304(%esp)
movaps  400(%esp), %xmm5
movaps  %xmm6, 288(%esp)
movaps  384(%esp), %xmm4
movaps  %xmm5, 352(%esp)
movaps  688(%esp), %xmm7
movaps  %xmm4, 336(%esp)
movaps  672(%esp), %xmm6
movaps  800(%esp), %xmm4
movaps  656(%esp), %xmm5
mulps   %xmm4, %xmm7
movaps  496(%esp), %xmm0
mulps   %xmm4, %xmm6
movaps  432(%esp), %xmm3
mulps   %xmm4, %xmm5
movaps  %xmm0, 320(%esp)
movaps  %xmm3, 368(%esp)
```

pre-loop code (gcc)

```
movaps    simd_temp3_58.0, %xmm7
mulps     %xmm6, %xmm7
divps     (%ebx), %xmm7
movaps    %xmm7, 80(%esp)
movaps    96(%esp), %xmm7
shufps    $33, %xmm3, %xmm0
movaps    simd_temp5_60.0, %xmm3
mulps     %xmm6, %xmm3
divps     (%ebx), %xmm3
movaps    %xmm3, 32(%esp)
shufps    $200, %xmm2, %xmm0
movaps    simd_temp4_59.0, %xmm2
mulps     %xmm6, %xmm2
divps     (%ebx), %xmm2
movaps    (%esp), %xmm6
movl      24(%esp), %ebx
movaps    %xmm2, 48(%esp)
shufps    $120, %xmm0, %xmm0
```

pre-loop code (icc)

Table C.1.: Assembly listing of the code before the loop in particle.

```
.L16:
        movaps  320(%esp), %xmm3
        addl    $1, %eax
        movaps  %xmm7, 592(%esp)
        cmpl    %eax, %edx
        movaps  %xmm3, %xmm2
        movaps  %xmm6, 576(%esp)
        mulps   %xmm4, %xmm2
        movaps  %xmm5, 560(%esp)
        movaps  %xmm2, 640(%esp)
        movaps  304(%esp), %xmm2
        movaps  %xmm2, %xmm1
        mulps   %xmm4, %xmm1
        movaps  %xmm1, 624(%esp)
        movaps  288(%esp), %xmm1
        movaps  %xmm1, %xmm0
        mulps   %xmm4, %xmm0
        movaps  %xmm0, 608(%esp)
        movaps  368(%esp), %xmm0
        addps   640(%esp), %xmm0
        movaps  %xmm0, 368(%esp)
        movaps  352(%esp), %xmm0
        addps   624(%esp), %xmm0
        movaps  %xmm0, 352(%esp)
        movaps  336(%esp), %xmm0
        addps   608(%esp), %xmm0
        movaps  %xmm0, 336(%esp)
        movaps  592(%esp), %xmm0
        divps   816(%esp), %xmm0
        movaps  %xmm0, 544(%esp)
        movaps  576(%esp), %xmm0
        divps   816(%esp), %xmm0
        addps   544(%esp), %xmm3
        movaps  %xmm3, 320(%esp)
        movaps  %xmm0, 528(%esp)
        movaps  560(%esp), %xmm0
        divps   816(%esp), %xmm0
        addps   528(%esp), %xmm2
        movaps  %xmm2, 304(%esp)
        movaps  %xmm0, 512(%esp)
        addps   512(%esp), %xmm1
        movaps  %xmm1, 288(%esp)
        jg      .L16
.L15:
        movaps  304(%esp), %xmm1
        ...
```

inner loop (gcc)

```
..B3.2:
        movl    $1, %edx
        xorl    %ecx, %ecx
        cmpl    272(%esp), %ebx
        movaps  %xmm6, 160(%esp)
        movaps  %xmm0, 176(%esp)
        movaps  %xmm1, 192(%esp)
        movaps  %xmm4, 144(%esp)
        movaps  %xmm5, 128(%esp)
        movaps  %xmm7, %xmm2
        cmovl   %edx, %ecx
        testl   %ecx, %ecx
        je      ..B3.5

        movaps  %xmm7, 96(%esp)
        movaps  %xmm4, (%esp)
        movaps  %xmm2, 112(%esp)
        movaps  64(%esp), %xmm2
        movaps  %xmm5, %xmm3
        mulps   %xmm2, %xmm4
        mulps   %xmm2, %xmm7
        mulps   %xmm2, %xmm3
        movaps  112(%esp), %xmm2
        addps   80(%esp), %xmm5
        addps   %xmm3, %xmm1
        addps   %xmm4, %xmm0
        movaps  (%esp), %xmm4
        addps   48(%esp), %xmm4
        addps   %xmm7, %xmm6
        movaps  96(%esp), %xmm7
        addps   32(%esp), %xmm7
        incl    %ebx
        testl   %ecx, %ecx
        jne     ..B3.2

        movaps  128(%esp), %xmm5
        movaps  144(%esp), %xmm4
        movaps  192(%esp), %xmm1
        movaps  176(%esp), %xmm0
        movaps  %xmm2, %xmm7
        movaps  160(%esp), %xmm6
        jmp     ..B3.2

..B3.5:
        movaps  %xmm4, %xmm2
        ...
```

inner loop (icc)

Table C.2.: Assembly listing of the loop body in particle.

Bibliography

[1] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher. In *INDOCRYPT*, pages 43–51, 2000.

[2] Fischer Black and Myron S Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[4] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of PLDI*, pages 279–290, 1995.

[5] Thinking Machines Corp. *C* Programmig Guide Version 6.0*, November 1990.

[6] Burroughs Corporation. *ILLIAC IV - System Characteristics and Programming Manual*. Burroughs Corporation, 1972. 66000C,IL4-PM1.

[7] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) User Guide*, November 1992. Document Part Number: 9302-0101 Revision: A3.

[8] Patrick Cousot. Progress on Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics 10 Years Back, 10 Years Ahead, Volume 2000 of Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[9] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[10] William J. Dally, Francois Labonte, Abhishek Das, Pat Hanrahan, Jung Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with Streams. In *SC*, page 35, 2003.

[11] Randall J. Fisher and Henry G. Dietz. The Scc Compiler: SWARing at MMX 3DNow! In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 399–414, London, UK, 2000. Springer-Verlag.

[12] Freescale. *AltiVec Technology Programming Interface Manual*, June 1999. ALTIVECPIM/D 06/1999 Rev. 0.

[13] Freescale. *AltiVec Technology Programming Environments Manual*, April 2006. ALTIVECPEM/D 04/2006 Rev. 3.

[14] Nicolas Fritz, Philipp Lucas, and Reinhard Wilhelm. Exploiting SIMD Parallelism with the CGiS Compiler Framework. In Vikram Adve, María Jesús Garzarán, and Paul Petersen, editors, *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, volume 5234 of *LNCS*, pages 246–260. Springer-Verlag, October 2008.

[15] Gernot Gebhard and Philipp Lucas. OORS: An Object-Oriented Rewrite System. *Computer Science and Information Systems (ComSIS)*, 4(2):1–26, December 2007.

[16] Andrew S. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press Ltd., London, UK, UK, 1989.

[17] AbsInt GmbH. aiSee Graph Layout Software. http://www.aisee.com.

[18] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.0*, December 2008.

[19] M. Hassaballah, Saleh Omran, and Youssef B. Mahdy. A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications. *Comput. J.*, 51(6):630–649, 2008.

[20] R. M. Howe. Simulation of Linear Systems Using Modified Euler Integration Methods. *Trans. Soc. Comput. Simul. Int.*, 5(2):125–152, 1988.

[21] Robert A. Hummel, B. Kimia, and Steven W. Zucker. Deblurring Gaussian Blur. *Comput. Vision Graph. Image Process.*, 38(1):66–80, 1987.

[22] IBM. Access and Management Library for Synergistic Processor Elements, libspe2. `http://www.ibm.com/developerworks/library/pa-libspe2/`.

[23] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2007.

[24] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, November 2008.

[25] Intel. *Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor*, November 2008. Data Sheet.

[26] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] Ken Kennedy, Charles Koelbel, and Hans P. Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *HOPL*, pages 1–22, 2007.

[28] Shorin Kyo and Ichiro Kuroda. An Extended C Language and a SIMD Compiler for Efficient Implementation of Image Filters on Media Extended Micro-Processors. In *Proceedings of Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pages 234–241, 2003.

[29] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *In PLDI*, 2003.

[30] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. Technical Report LCS-TM-601, MIT Laboratory for Computer Science, November 1999.

[31] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[32] Philipp Lucas. *CGiS: High-Level Data-Parallel GPU Programming*. PhD thesis, Saarland University, Saarbrücken, January 2008.

[33] Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The CGiS Compiler—A Tool Demonstration. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*, pages 105–108, 2006.

[34] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[35] F. Martin. *Generation of Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.

[36] F. Martin, R. Heckmann, and S. Thesing. *PAG – The Program Analyzer Generator User's Manual*, 2004. Version 2.0.

[37] Millind Mittal, Alex Peleg, and Uri Weiser. MMX technology architecture overview. *Intel Technology Journal*, Q3:12, 1997.

[38] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[39] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[40] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter*, pages 29–42, 1993.

[41] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[42] NVIDIA. *CUDA Programming Guide Version 0.8*, February 2007.

[43] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, 1999.

[44] Lawrence C. Paulson. *ML for the Working Programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[45] Ivan Pryanishnikov, Andreas Krall, and R. Nigel Horspool. Compiler Optimizations for Processors with SIMD Instructions. *Software—Practice & Experience*, 37(1):93–113, 2007.

[46] Gang Ren, Peng Wu, and David Padua. An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *IPDPS*, 2005.

[47] Gang Ren, Peng Wu, and David A. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *Proceedings of LCPC'03*, pages 420–435, 2003.

[48] Ronald L. Rivest. The RC5 Encryption Algorithm. In *Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.

[49] Richard M. Russell. The CRAY-1 Computer System. *Commun. ACM*, 21(1):63–72, 1978.

[50] Georg Sander. VCG – Visualization of Compiler Graphs. Technical Report Feb8-5, Saarland University, 8 1996. See also http://www.aisee.com/gdl/nutshell/.

[51] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[52] Freescale Semiconductor. *MPC7450 RISC Microprocessor Family Reference Manual*. Freescale Semiconductor, 2005. `http://www.freescale.com/files/32bit/doc/ref\_manual/MPC7450UM.pdf`.

[53] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Performance Impact of Misaligned Accesses in SIMD Extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2006)*, pages 334–342, November 2006.

[54] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Efficient Vectorization of the FIR Filter. In *In Proc. 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC*, pages 432–437, 2005.

[55] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Course Technology, February 2005.

[56] Nathan T. Slingerland and Alan Jay Smith. Multimedia Extensions for General Purpose Microprocessors: A Survey. *Microprocessors and Microsystems*, 29(5):225–246, 2005.

[57] Robert Stephens. A Survey of Stream Processing. *Acta Inf.*, 34(7):491–541, 1997.

[58] Christian Tenllado, Luis Piñuel, Manuel Prieto, and Francky Catthoor. Pack Transposition: Enhancing Superword Level Parallelism Exploitation. In *Proceedings of Parallel Computing (ParCo)*, pages 573–580, 2005.

[59] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Computational Complexity*, pages 179–196, 2002.

[60] Eric Weisstein. The Mandelbrot Set. `http://mathworld.wolfram.com/MandelbrotSet.html`, 2005.

[61] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[62] Peng Wu, Alexandre E. Eichenberer, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*, pages 169–178, 2005.

[63] Taiichi Yuasa, Toshiro Kijima, and Yutaka Konishi. The Data-Parallel C Language NCX and its Implementation Strategies. In *Theory and Practice of Parallel Programming*, pages 433–456, 1994.

[64] Hans P. Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

Index