A PTX Code Generator for LLVM

Helge Rhodin

Saarland University
Saarbrücken, Germany

Bachelor's Thesis
Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

Betreuender Hochschullehrer / Supervisor:
Jun.-Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany

Gutachter / Reviewers:
Jun.-Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes,
Saarbrücken, Germany

Dekan / Dean:
Prof. Dr.-Ing. Holger Hermanns, Universität des Saarlandes,
Saarbrücken, Germany

Eingereicht am / Thesis submitted:
Okt 29, 2010

Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Im Stadtwald - Building E 1 1
66123 Saarbrücken

Abstract

Today's GPGPU architectures and corresponding high level programming languages like CUDA replace the traditionally restricted GPU pipelines. Proprietary compilers allow to translate these languages into native GPU assembly. Unfortunately, these compilers are non-customizable and restricted to static compilation. High performant application currently require particular manual optimizations.

To overcome these cumbersome manual optimizations, this thesis develops an open source PTX code generator—PTX is assembly code for NVIDIA GPUs. The code generator is based on the existing open source LLVM compiler. In conjunction, both systems compose a customizable compiler for current GPU architectures.

Detailed resource analyzes and PTX shader run-time measurements demonstrate the capacity and quality of generated kernels. At this stage the PTX code generator achieves similar performance to the nvcc compiler.

The developed compiler forms a sound basis for a variety of applications and further research topics. Additional feature support, novel optimization techniques, and applications from various fields are conceivable.

Contents

Chapter 1
Introduction

1.1 Motivation

Multicore and vector programming is one of the fastest growing fields of computer science. Vector processors and parallel processing systems were already introduced in the field of *super computing* in the 1970s and 1980s [Wil94] and are represented in the low priced *personal computer* (PC) market since the 2000s [Asa06]. Single core architectures evolved to multicore and multicore to many core systems. The trend towards parallel computing is immense and currently moves towards *graphics processing unit* (GPU) architectures. GPU designs have always been parallel, but programmable restricted. Currently, highly restricted fixed pipeline GPU architectures are succeeded by GPUs with programmable shading capabilities with increasing parallelism and complexity. It became possible to execute general-purpose programs on GPUs. This technique is known as *general-purpose computing on graphics processing units* (GPGPU) [Owe07].

The current trend is to write GPGPU programs in GPU-specific high-level programming languages. CUDA and OpenCL are the most popular choices. GPU vendors provide compilers which compile these high-level programs into GPU assembly code. Oftentimes, the resulting assembly code is afterwards optimized manually. The manual optimization is cumbersome, but required for high performance applications. Different GPU applications require distinct optimizations, but the current available compilers are not able to fulfill all of them. Researchers complain about limited compiler functionalities and the impractical extensibility of current closed source compilers [Pop07, Fra10].

To overcome these drawbacks, this thesis develops an open source GPU code generator. It is based on the existing open source *low level virtual machine* (LLVM) compiler. In conjunction, both systems compose a customizable compiler for current GPU architectures. *Parallel thread execution* (PTX) [NVI10c] code—assembly code for NVIDIA GPUs—turned out to be the best fitting target language. The code generator of this thesis, hereinafter referred to as the *PTX code generator*, offers many interesting new options:

1. The previously described problem of closed source compilers is solved. The LLVM compiler allows to develop and use custom GPU and application specific optimization passes. LLVM's design is modular, the whole compilation pipeline is customizable. Existing passes can be combined arbitrarily and the implementation of new passes is easy.

2. Programs written in any source language which is supported by LLVM (e.g. C++, OpenCL) can be compiled to multicore GPU code. Even combinations of different source languages are possible. For example the combination of target independent high-level code with GPU-specific code (e.g. RenderMan [PIX98] shading language and OpenCL).

3. The NVIDIA *driver API*[1] can load and execute PTX code dynamically. However, the nvcc compiler is restricted to static compilation from CUDA files to PTX code. The PTX code generator allows to use the LLVM *just in time* (JIT) compiler to specialize code during runtime and utilize the driver API's JIT capabilities.

The PTX code generator and GPU programs in general have a broad field of application. The efficiency of GPU programs compared to CPU solutions highly depends on the problem to solve. Some problems are well suited for parallel execution; speed-ups of two orders of magnitude are possible, while others can only be computed sequentially. For example, the performance of a parallel image processing system [Yan08] improves by a factor of 200 and a profiled application suite gains speed-ups from 1.16X to 431X on the GeForce 8800 GTX processor [Ryo08] in comparison to CPU implementations. Applications from different fields profit from the (at least part-wise) execution of code on GPUs. Examples are video decoding, ray tracing, physical and chemical simulations, game physic engines, data compression, and image processing.

[1]The NVIDIA driver API is an interface for PTX kernel compilation and execution.

1.2 Overview

The thesis consists of two main parts: the code generator and the integration of PTX shaders into the OGRE [Gre06] renderer. It focuses on the correct development of the code generator while the renderer integration is limited to key features and mainly serves for evaluation purposes. This thesis covers many different Computer Science fields. A good understanding of compiler construction, programming languages and GPU architectures is necessary in order to understand, design, and implement the code generator. Computer graphics knowledge is essential for the proper renderer integration.

The remainder of this thesis is organized as follows. Section 1.3 continues the motivation for the PTX code generator development. Its application in the AnySL [Kar10] shading system is outlined. The introductory chapter is followed by Chapter 2 which embraces background knowledge required for understanding this thesis and covers related work. Section 2.2 gives information on the LLVM compiler, shows its advantages and its extensibility. An overview of current GPU designs and main differences to CPU architectures is given in Section 2.4. Sections 2.5 and 2.6 elaborate the chosen target language PTX and its benefits with respect to alternative languages. The code generator design, its challenges, features, and limitations are explained in Section 3. Chapter 3.5 defines a language extension for GPU-specific features of LLVM's intermediate representation. The evaluation of proper code generation and code quality is accomplished in Chapter 4. This chapter also includes an explanation of the integration into a deferred shading[2] OGRE setup (Section 4.2). Finally, Section 5 gives an outlook to additional applications and possible improvements of the PTX code generator and ways to bypass current GPU restrictions. The appendix 6 contains the complete source code of the shown shaders and test cases.

[2]Deferred shading is a rasterization renderer setup which delays all shading computations to the end of the graphics pipeline.

1.3 Application

The PTX code generator will be used in the AnySL system [Kar10]. AnySL is a new shader integration system which combines the advantages of high-level, target independent shading languages, and the speed of low level, target specific code. It integrates easily and with high performance into all kinds of renderers.

The present AnySL system implementation uses the LLVM compiler for optimizing high-level RenderMan [PIX98] surface shaders as well as C++ shaders for x86 hardware with SIMD instructions. It currently supports the CPU ray tracer renderers RTfact [Geo08], Manta [Big06], and PBRT [Pha10].

The LLVM PTX code generator enables the AnySL system to generate optimized shader code for GPUs. The code generator operates on any LLVM program represented in the intermediate *bitcode* representation. It is independent of high-level source languages which automatically allows the AnySL system to generate GPU code for all supported source shading languages. Figure 1.1 shows an exemplary RenderMan phong surface shader.

```
surface phong(){
  normal Nf = faceforward(N, I);
  vector reflView = reflect(I, Nf);
  color tmp = color(0.1, 0.06, 0.0);

  illuminance(P, Nf, PI/2) {
    color c = color (0, 0, 0);
    float dotLRefl = L.reflView;
    if (dotLRefl > 0.01) c = pow(dotLRefl, 50);
    float f = Nf.L;
    tmp += color(1, 0.2, 0) * Cl * f + c;
  }
  Ci = tmp;
}
```

Figure 1.1: Phong surface shader written in the RenderMan shading language.

The AnySL system transforms the RenderMan shader into LLVM bitcode and combines it with renderer specific glue code which implements shading features like the illuminance statement. A simplified version of the resulting bitcode shader is shown in Section 2.3 (Figure 2.1). Finally, the PTX code generator transforms the combined bitcode shader into a PTX kernel. GPU ray-tracers (e.g. Optix [Par10]) as well as rasterizers (e.g. OGRE) are possible targets.

Chapter 2
Field of Application

2.1 Related Work

At the beginning of the PTX code generator development no other open source PTX backend was announced to the public, neither for LLVM nor for any other compiler. The closed source CUDA compiler was the only available compiler which targets PTX code. During a talk about the PLang frontend [Gro09] NVIDIA mentioned that they are also developing an LLVM PTX backend. But no additional information is published up to now. Other GPU languages like the *compute abstraction layer* (CAL) are the target of closed source compilers only (see Section 2.6 for alternative GPU languages).

The closed source CUDA and OpenCL compilers are suitable for performance comparisons. No information about their design and contained optimization passes is publicly available. Conceptually, a PTX code generator can only be based on the currently available non-GPU code generators. Either the target independent code generator environment or other target specific backends of LLVM can be used. The target independent code generator provides a framework for backends targeting arbitrary assembly languages. Amongst others it includes algorithms for register allocation, scheduling, and stack frame representation. On the contrary, a custom LLVM backend which does not use the target independent code generator needs to incorporate all of these algorithms. Section 3.3 provides a comparison of both concepts. The custom backend approach turned out to be convenient for the PTX code generator. The concept of the LLVM CBackend is the main basis for the PTX code generator.

The development of an LLVM PTX backend based on the target independent code generator of LLVM was recently announced on the LLVM mailing list. The current preliminary status of the project is not comparable though to the PTX code generator of this thesis. Once the project reaches a more mature status, pros and cons of both concepts can be analyzed.

The 4-Centauri [Dit09] project pursues a similar target—.NET bytecode is mapped into PTX code. Potentially, every language with .NET-support can be mapped to PTX code. Like LLVM's intermediate code, the .NET bytecode instruction set is akin to PTX. But, mapping from the .NET intermediate code to PTX code requires a different concept. The .NET bytecode is stack-based and object-oriented which prevents a simple one-to-one mapping. Unfortunately, the project has not produced any statistics and results by now and development seems to be stalled.

2.2 LLVM Compiler

LLVM [Lat04] is an open source compiler that is designed modularly. Different source languages are compiled by appropriate frontends to the intermediate bitcode representation. Details about the intermediate language are given in the following Section 2.3. Middle-end optimization passes operate on this intermediate representation. Finally, target specific code generators—also called backends—transform the intermediate representation to machine code for various architectures. LLVM allows static as well as just in time compilation.

LLVM frontends for C, C++, Objective-C, and Fortran are available. Frontends for other language like Java and Scheme are under development. LLVM supports many target architectures through different backends, including X86, PowerPC, ARM, Thumb, SPARC, Alpha, and CellSPU. A C backend also exists. In this respect and also in terms of compilation time and performance of the generated code, LLVM is comparable to the gcc compiler [LLV10]. This thesis extends the list of LLVM's supported backends by its first GPU backend - the PTX code generator.

2.3 LLVM Bitcode

The intermediate LLVM representation [Lat10], often called LLVM bitcode, is a low-level assembly language. LLVM is supposed to be able to represent all currently known high-level languages in LLVM bitcode. LLVM bitcode programs are constructed hierarchically. A program consists of *modules* which contain *function* definitions and *global fields*. Functions are divided into *basic blocks*. A block consists of sequential *instructions*. Instructions operate on a *virtual-memory* and *virtual-register* model. LLVM bitcode is in *static single assignment form* (SSA)[1].

[1]In SSA form variables are only assigned once, specific PHI-instructions are required for convergent control flow. Consider the work of R. Cytron et al. [Cyt91] for additional details.

It provides type safety by a *strict type system*. New types can be defined by encapsulating existing types in C-like structs and arrays. It is also extensible by new language features (for details see next section). Figure 2.1 shows the hierarchical LLVM bitcode representation of a RenderMan shader converted to LLVM bitcode by the AnySL system.

Extensibility

The LLVM compiler design is suitable for multiple frontends which process various high-level source languages and a variety of backends which generate code for different architectures. This variety requires the intermediate LLVM bitcode representation to be general and extensible. The following three characteristics of LLVM bitcode allow to represent GPU-specific features, which are required by the PTX code generator:

1. *Intrinsic functions*. Without specific handling, unknown intrinsics are handled like external function calls during all LLVM passes. Intrinsics are well suited for target dependent instructions like texture look-ups or mathematical functions which are implemented in hardware on some targets.

2. Arbitrary *meta-data*. Meta-data can be attached to every LLVM instruction. This is a smart way to provide additional information like source-level debug information to optimization passes and backends.

3. *Address spaces*. Each global variable and each pointer contains an address space number attribute. The default address space is 0, memory allocated by alloca instructions resides automatically in address space 0. There is no convention for non-zero values thus they can be used for target specific memory locations.

```
; ModuleID = 'rsl_phong.bc'

%0 = type{i16, i16, i16, i16}
%1 = type{i8, i8, i8, i8}
%Matrix4f = type{float, float, (...), float}
%Vec3f = type{float, float, float}
%struct.anon = type{%Vec3f, float, %Matrix4f, float, %Vec3f, \\
                    i32, i32, %0*, %0*, %1*, i32}
@data_dev = addrspace(3) global %struct.anon zeroinitializer, align 4
[...]

define void @inner_shade() noinline {
entry:
  %s1 = tail call zeroext i16 @_Z16__ptx_sreg_tid_yv()
  %u1 = zext i16 %s1 to i32
  %u2 = load i32* getelementptr inbounds \\
           (%struct.anon addrspace(3)* @data_dev, i32 0, i32 5)
  %u3 = mul i32 %u2, %u1
  [...]

  %f1 = tail call float @sqrtf(float %x1)
  %div.i218 = fdiv float 1.000000e+00, %call2.i.i217
  %mul.i.i219 = fmul float %call53, %div.i218
  [...]

for.cond:
  %numLights = load i32* getelementptr inbounds \\
                   (%struct.anon addrspace(3)* @data_dev, i32 0, i32 10)
  %lightValid = icmp slt i32 %currentLight, %numLights
  br i1 %lightValid, label %for.body, label %shade.exit
shade.exit:
  %.16.1.2.i = phi i32 [1031127695, %entry], [%x3, %for.cond]
  %.16.2.2.i = phi i32 [0, %entry], [%x4, %for.cond]
  [...]

  %u8 = load %1** getelementptr inbounds
             (%struct.anon addrspace(3)* @data_dev, i32 0, i32 9)
  %u9 = load i32* getelementptr inbounds
        (%struct.anon addrspace(3)* @data_dev, i32 0, i32 5)
  %u10 = mul i32 %u9, %x5
  %u11 = add i32 %u10, %x6
  %u12 = getelementptr inbounds %1* %u8, \\
                              i32 %u11
  %u13 = bitcast %1* %u12 to i32 addrspace(2)*
  store i32 %x7, i32 addrspace(2)* %u13
  ret void
}
```

Figure 2.1: Extracts from a phong surface shader in bitcode representation. The shader is generated from the RenderMan shader in Figure 1.1 with the AnySL system.

2.4 GPU Design

Current GPU Architectures are dominated by many-core designs consisting of many multiprocessors. Each processor contains a number of scalar cores, different memory spaces and fast *arithmetic logic units* (ALU). They are tuned for high memory bandwidth, heavy computational capacity, and fine-grained parallel execution [Khr09]. In comparison to CPUs, which are specialized on data caching and flow control, modern GPUs try to hide memory access latencies by executing computations in parallel. *Data-level* as well as *thread-level parallelism* is utilized. Cutting-edge GPUs like the NVIDIA *Fermi* architecture also provide caching.

The idea is to have thousands of threads which execute in parallel on many simple but powerful multicore processors. While one thread is issuing a slow memory access a different thread can take over execution and maintain a high occupancy of each underlying processor. In contrast, CPUs try to predict execution paths and cache data and instructions in an intelligent way. This GPU design needs to allow fast context switching between concurrent threads and requires dedicated thread scheduling. In order to reduce the complexity, the current GPU processor designs are kept simple, no branch prediction or speculative execution is performed. Such details are covered more elaborately for NVIDIA GPUs in the following section.

Both designs, GPU and CPU, have advantages and disadvantages: They are well suited for different, mostly orthogonal problems. GPUs are optimal for compute-intensive problems which can be solved in parallel while the CPU is well suited for sequential and control-flow-intensive problems. Shifting compute-intensive program parts to the GPU allows huge performance improvements.

NVIDIA GPUs

The generated PTX code can be executed on any CUDA capable NVIDIA Graphics card. The current NVIDIA product line includes different GPU series, ranging from the GeForce desktop and notebook series for gaming and the powerful Quadro series to the Tesla high performance GPGPU series. Each series has different performance properties designated for its specific application field. The number of processors, memory sizes, and throughput, the number of registers, and clock rates differ. The general design of all current CUDA enabled products is similar.

The GPU consists of a number of stream-multiprocessors, which itself include a few scalar processor cores, on-chip *shared memory*, *constant* and *texture caches*, register banks, and a single *instruction unit* (see Figure 2.2). Each scalar processor of one multiprocessor shares these resources. Threads which are executed on cores of one multiprocessor share the pool of registers. It is strictly partitioned among the active threads. The shared memory serves as a communication channel between different cores of a single multiprocessor. Because of its low latency, shared memory is also often deployed as a custom cache.

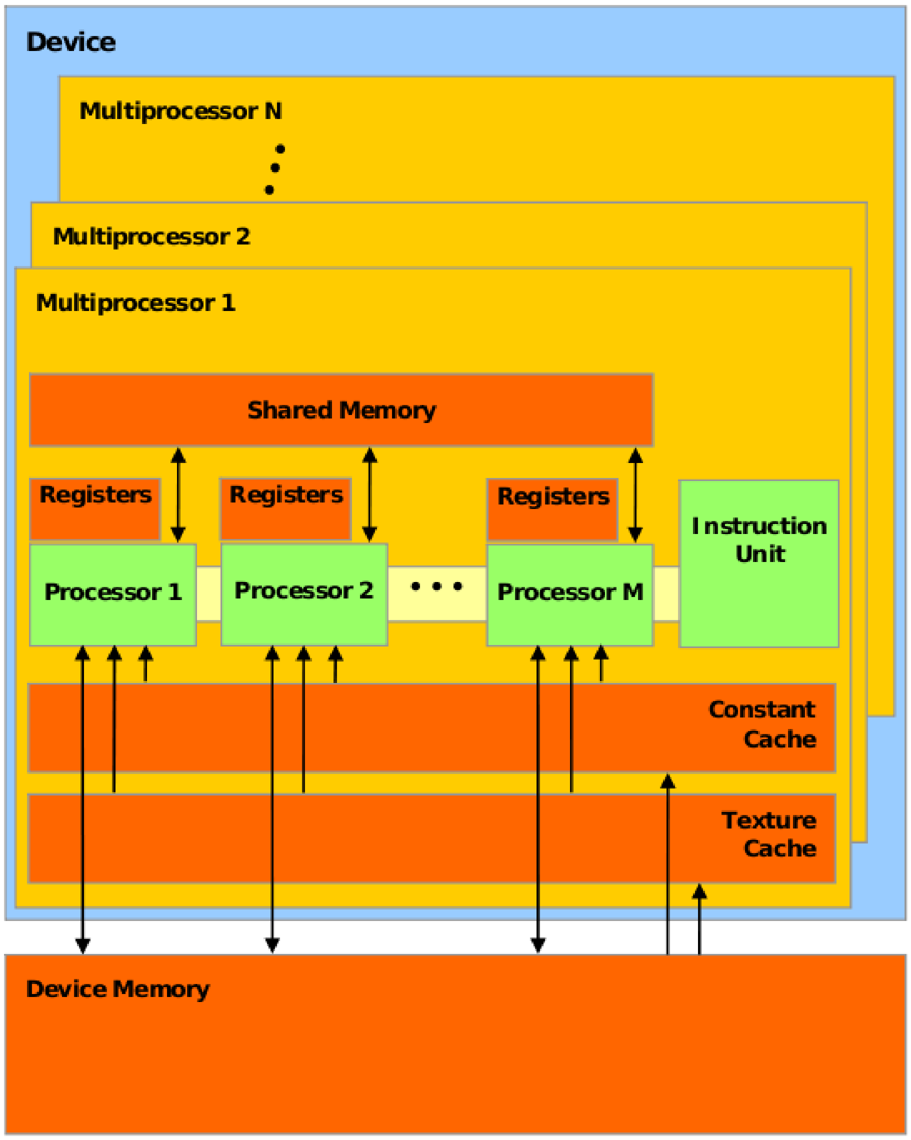


Figure 2.2: Abstract representation of NVIDIA GPUs and their resources. The image is extracted from the NVIDIA CUDA Programming Guide [NVI10a].

During a single clock cycle all scalar processors execute the same instruction on their own set of registers in parallel. This *single instruction multiple data* (SIMD) design is further extended by the so called *single instruction multiple* *thread* (SIMT) design. Each thread maintains its current state in the instruction sequence. One can think of threads having their own *program counter* (PC), the actual implementation involves stacks of masks. Still only a single instruction can be executed per clock cycle on cores of one multiprocessor. Cores with a different PC are excluded from execution, similar to predication[2]. Every distinct instruction which a PC of the currently executing threads points to are executed sequentially, sacrificing performance. If the PCs of threads currently executing on the cores of one multiprocessor point to n distinct instructions, execution time generally scales by a factor of O(n). The advantage of SIMT is that threads can branch independently which allows flexible programming. The border between thread-level parallel and data-parallel execution is blurred. It is possible to write data-parallel code by eliminating divergent control flow or thread-level-parallel code by neglecting control flow drawbacks and sacrificing performance.

Each multiprocessor has one *SIMT unit*. It schedules threads to the scalar cores of the corresponding multiprocessor. Threads are divided into groups called *warps*. The current warp size is 32, different sizes are possible in the future. The SIMT unit iteratively selects idle warps, issues their next instructions and executes them. Instructions of divergent threads of one warp are executed in sequence as described before. Threads of different warps are unrelated concerning divergent control flow. The SIMT unit is able to hide the latency of slow instructions, for example loads and stores. While one warp is performing a high latency instruction, the SIMT unit can pick a different idle warp for execution. Context switching between different warps causes minimal overhead. Because of the strict register partition among active threads it only requires to load the PC of the new thread. This permits fine grained parallelism including thousands of threads.

NVIDIA GPUs have an involved memory model. Beside the on-chip shared memory, NVIDIA GPUs are equipped with a *global memory* bank which is accessed by all multiprocessors concurrently. Parts of this global memory can be declared as *constant-* or *texture-memory*. Such memory regions are cached by the constant-cache and the *texture unit*, respectively. They can be accessed by special cached fetch instructions. The texture unit features interpolation of nearby data elements as well as caches which are optimized for two and three dimensional texture access patterns. Memory declared as constant can still be accessed by the typical load and store instructions. However, every store access potentially results in unpredictable behavior because consecutive cached reads can either read the value from the cache or from the newly written global memory.

The latency of load and store instructions depends on the locality of the ad-

[2]Consider the work of J. Park and M. Schlansker [Par91] for an introduction on predicated execution and example applications.

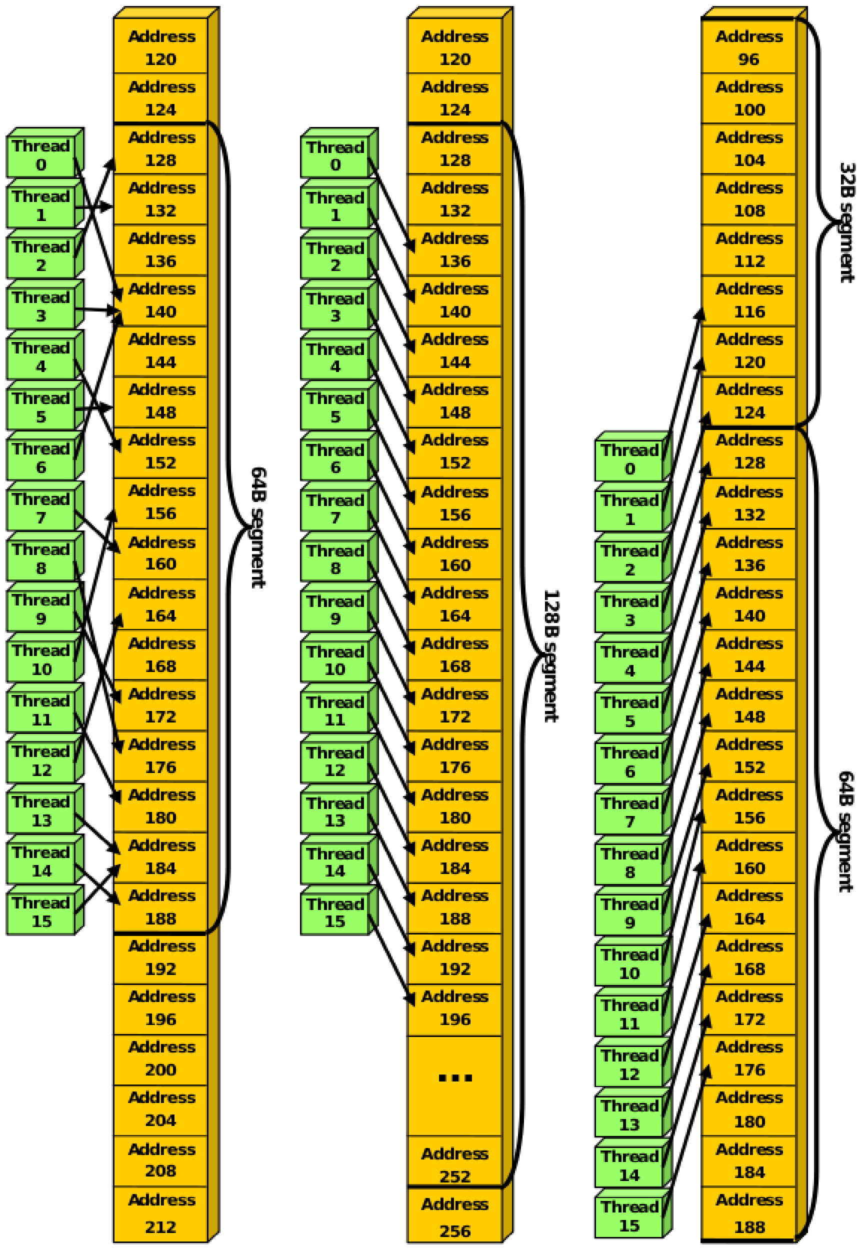


Figure 2.3: Different memory access patterns, their segment size and the required number of transaction. Left: 64b segment, one transaction. Center: unaligned, 128 bit segment, one transaction. Right: two segments, two transactions. The image is extracted from the NVIDIA CUDA Programming Guide [NVI10a].

dressed memory region. If all accessed 4-bit or 8-bit words of one half warp[3] are located in the same segment of 128 bytes the memory access is *coalesced*, and only one memory transaction is issued. Each additional segment results in an additional transaction. This results in a latency of O(n) with n being the number of accessed memory segments per half warp. The same scheme applies for 2-byte words in 64-bit segments and for 1-byte words in 32-bit segments. The accessed locations of coalesced loads and stores, which point to a single segment, can be in any order, they can also point to the same address. Figure 2.3 shows coalesced access patterns.

The processor occupancy of GPU programs decreases with the number of required registers per thread because less threads are executed in parallel. A maximum of R/n threads can be executed in parallel per stream processor, where R is the number of physical registers and n is the number of registers per thread. Also the overall amount of shared and local memory cannot exceed the available physical resources. This can lead to performance loss. The relation between occupancy and performance is not self-contained. Also the ratio of arithmetic instructions and memory accesses, and the kernel size is involved.

All details about NVIDIA GPUs are extracted from the official NVIDIA reference manuals [NVI09, NVI10a]. The knowledge about NVIDIA GPUs is required for a good understanding of the design and behavior of PTX programs.

2.5 Target Language: PTX

Parallel Thread Execution (PTX) [NVI10c] is a human-readable assembly-like low-level programming language for NVIDIA GPUs. PTX belongs to the *reduced instruction set computing* (RISC) type languages. But PTX is not machine code, it is a typed intermediate language which can be compiled to specific machine code of different multicore architectures. PTX is, to some extend, target dependent: It contains a lot of GPU-specific instructions as its main purpose is to represent efficient GPU code. PTX code provides thread synchronization instructions as well as access to the GPU texture unit, to the different memory spaces, and to mathematical functions which are implemented in hardware. Figure 2.4 shows a simplified PTX code example which corresponds to the previously shown RenderMan phong shader.

PTX is similar to the previously described LLVM bitcode except for the SSA property and the support of GPU-specific features. PTX exposes the fine grained parallelism of the underlying hardware with an hierarchical paradigm. Threads are grouped to warps of size 32. Warps are combined to *cooperative thread ar-*

[3]A half warp is the first or second half of threads from a warp.

```
.const.align 4 .b8 data_dev[120]  = {0, 0, (...) , 0};

.entry inner_shade
{
  .reg .u32 u1;
  .reg .u16 s1;
  .reg .f32 f1;
  .reg .pred p1;
  [...]

entry:
  mov.u16 s1, %tid.y;
  cvt.u32.u16 u1, s1;
  ld.const.u32 u2, [data_dev + 96];
  mul.lo.u32 u3, u2, u1;
  [...]

  sqrt.approx.f32 f1, x1;
  div.full.f32 f2, 1.000000, f1;
  mul.f32 f3, x2, f2;
  [...]

  ld.const.u32 numLights, [data_dev + 116];
  setp.lt.s32 lightValid, currentLight, numLights;
 @lightValid  mov.u32 u4__PHI_TEMP, x3;
 @lightValid  mov.u32 u5__PHI_TEMP, x4;
  [...]
 @lightValid  bra for_body;
  mov.u32 u6__PHI_TEMP, x3;
  mov.u32 u7__PHI_TEMP, x4;
  bra shade_exit;
shade_exit:
  mov.u32 u6, u6__PHI_TEMP;
  mov.u32 u7, u7__PHI_TEMP;
  [...]

  ld.const.u32 u8, [data_dev + 112];
  ld.const.u32 u9, [data_dev + 96];
  mul.lo.u32 u10, u9, x5;
  add.u32 u11, u10, x6;
  shl.b32 u12, u11, 2;
  add.u32 u13, u8, u12;
  st.global.u32 [u13], x7;
  exit;
}
```

Figure 2.4: Extracts from a phong surface shader in PTX code. The PTX code is generated by the PTX code generator from the bitcode shader of Figure 2.1.

rays (CTAs[4]), which are parts of *grids*. CTAs and grids are one-, two-, or three-dimensional, their size in each dimension must be specified. Picture 2.5 shows this hierarchical structure.

This hierarchical fragmentation of threads directly maps to the previously described GPU hardware. Each CTA is assigned to one multiprocessor. The SIMT unit of each multiprocessor selects suitable warps out of all CTAs which are assigned to this processor. Finally, each thread of the currently executing warp is executed by one scalar core. This mapping determines the memory access scope and restricts the communication possibilities between threads. Threads of different CTAs can only communicate through the global memory space. PTX provides *atomic operations* as well as *barrier* and *memory-barrier synchronization* instructions. Threads of a single CTA additionally have access and can communicate through the shared memory space of each multiprocessor. Shared memory provides significantly faster access in comparison to global memory. At the warp level PTX also provides *vote instructions* which perform reduction of predicates across a single warp. For each thread PTX reserves a memory region of the global memory bank. This *local memory* region is restricted to access from its corresponding thread. The memory regions are arranged in a *struct of arrays* type manner such that loads and stores of a single warp are always coalesced[5].General properties and disparities of PTX and LLVM bitcode are given in Section 3.4. Specific details can be found in the official PTX documentation [NVI10c].

PTX is specifically designed for NVIDIA GPUs. However, its abstraction of threads, synchronization mechanics, and virtual registers can easily be mapped to various multicore architectures. For example the Ocelot system [Ker09] allows to execute PTX programs on x86-CPUs.

PTX is used by NVIDIA's CUDA and OpenCL compilers as an intermediate representation of GPU-specific code. Direct programming in PTX is not practical due to the low-level assembly-like construction. Although, critical code parts can be fine tuned manually. Chart 2.6 visualizes the *compilation pipeline*. PTX code is standardized and forward compatible to future targets. PTX is therefore convenient as a standardized exchange format for parallel computing programs and because of its simplicity suitable as a target for external compilers. The next section compares PTX to alternative target languages and reveals its advantages.

[4]Sometimes also called Blocks.

[5]Internally a struct which is stored in local memory is split up in its elements. The nth element of each thread from one warp lie in one segment in sequence. One coalesced memory access is sufficient for all threads of a single warp to access its nth element.

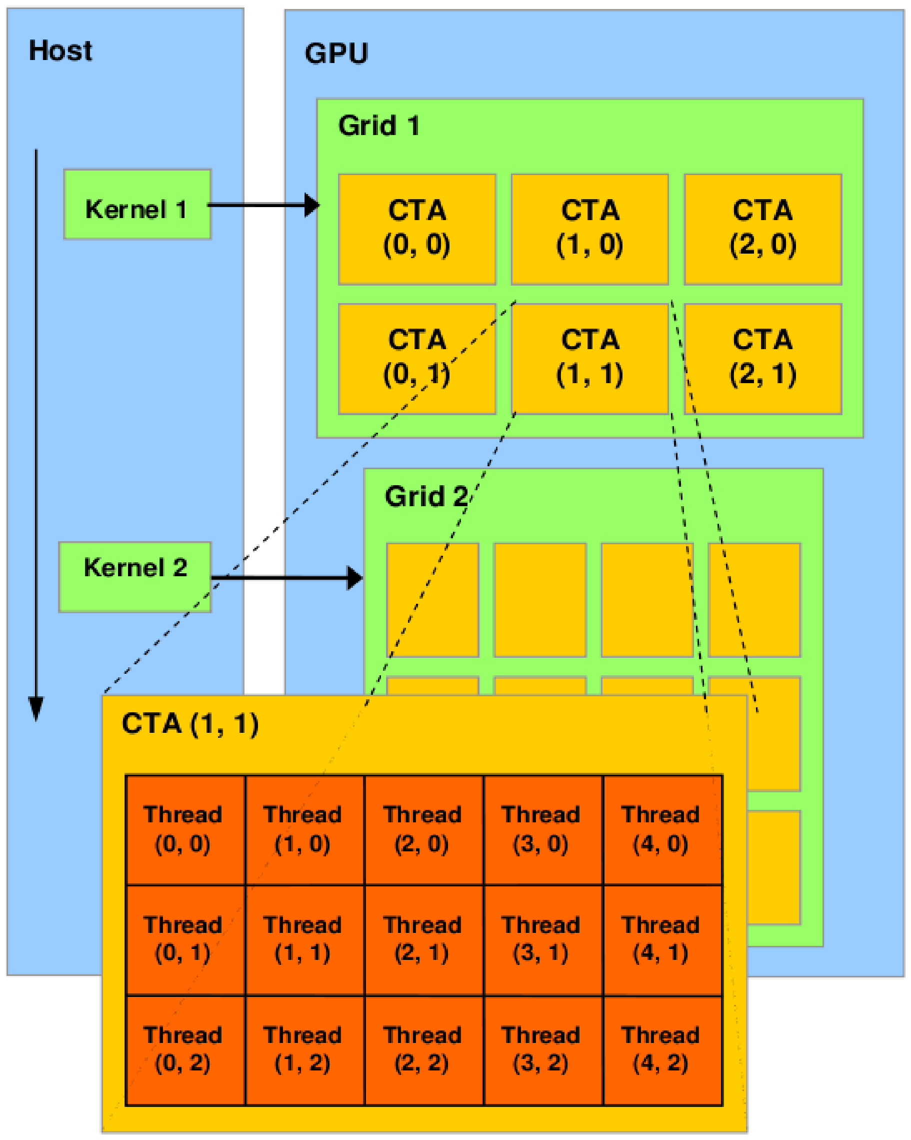


Figure 2.5: Multidimensional hierarchical structure of PTX threads (Threads $\subset$ CTAs $\subset$ Grids $\subset$ GPU). The image is extracted from the NVIDIA PTX ISA 1.4 [NVI10b].

2.6 Alternative Target Languages

The aim of this thesis is to extend the LLVM compiler with a GPU code generator. This section discusses advantages of PTX as the target language compared to alternative languages. OpenCL [Khr09] and NVIDIA CUDA [NVI10a] are currently the most popular general-purpose, GPU-specific programming languages besides PTX. CUDA has the disadvantage of being limited to NVIDIA GPUs and it lacks JIT support. OpenCL is supported by GPUs of both vendors. The current AMD Stream SDK implementation [AMD09] is based on and incorporates all advantages and disadvantages of OpenCL. Both languages are high-level languages based on C and extended with GPU-specific language constructs. The performance is almost the same [Pet03].

Figure 2.6 depicts the compilation pipeline of all three languages. Each high-level language requires a dedicated compiler which breaks down high-level language constructs to low-level instructions integrating GPU-specific as well as miscellaneous optimizations. In the case of NVIDIA GPU targets, the compilation process is split into two stages. First, high-level CUDA and OpenCL programs are compiled into the intermediate PTX code representation. During this step common optimizations like *inlining*, *global value numbering*, *constant propagation*, *common subexpression elimination* and other data-flow optimizations are performed. The second step transforms PTX code into machine code, register allocation, instruction scheduling, dead-code elimination, and numerous other late-optimizations are performed. ATI handles OpenCL code in a similar way. First, the ATI compiler compiles OpenCL to the CAL and from this *intermediate language* (IL) to ATI GPU machine code. NVIDIA as well as ATI have not published any details about the performed optimizations and their properties. The previously listed optimizations are extracted from the NVIDIA Optix paper [Par10] which deals with their GPU ray-tracer system.

Out of the high level GPU languages OpenCL is the preferred choice because of its target independency. The next paragraph therefore focuses on comparing PTX and OpenCL.

Both languages provide a different trade-off between power and responsibility of the user and the language compiler. Due to the OpenCL abstraction level many low-level features are hidden, such that the programming process is simplified. Code written in OpenCL is optimized for the desired target architecture by the OpenCL compiler allowing the programmer to write abstract and clear code. Low-level code optimizations are not necessary and only possible to a certain extent, because OpenCL's internal optimizations may revert previously applied optimizations. The OpenCL compiler may optimize the code of one application nicely but may fail in a different case. The power and responsibility of choosing the right optimization is propagated from the user to the compiler.

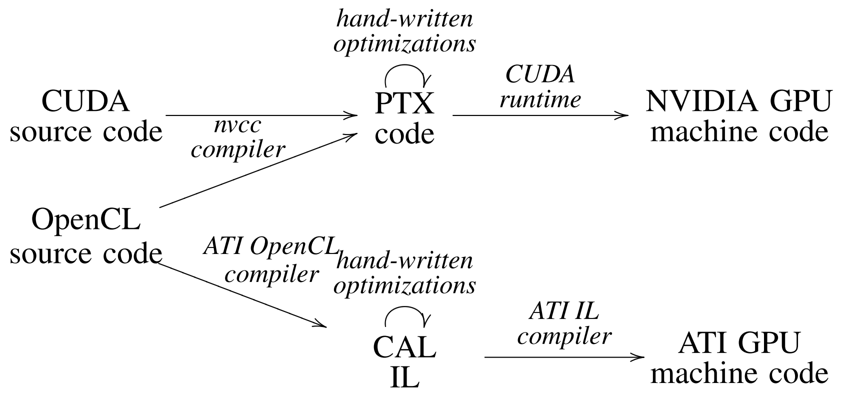


Figure 2.6: Compilation pipelines of current GPGPU languages for NVIDIA and ATI GPUs.

The low-level PTX code needs to be optimized beforehand, the NVIDIA PTX compiler only applies late-optimizations. All other, more sophisticated optimizations need to be applied manually or by preceding compiler passes. On the one hand PTX propagates more work to the user, on the other hand this allows for more flexibility and power. The user has the power and is responsible to choose and execute the right optimization before running the NVIDIA PTX compiler.

Beside the greater flexibility, PTXs simplicity and similarity to LLVM bitcode allows an easy and fast code generation. The conversion from LLVM bitcode to OpenCL code would be more complicated. The previously shown phong shaders evince the code-complexity of PTX and OpenCL code. The RenderMan source code is shown in Figure 1.1. Figure 2.7 shows its C++ version. The C++ code is extended with GPU-specific properties akin to OpenCL. Figure 2.1 shows its bitcode representation and Figure 2.4 depicts the PTX counterpart. The similarity of LLVM bitcode and PTX code is noticeable. A transformation from one language to the other is straightforward. The bitcode and C++ code representations differ strongly; no immediate transformation is possible. It is easy to transform high-level languages into low-level representations. However, the opposite direction, which would be required for bitcode to OpenCL conversions, is complicated. OpenCL lacks many low-level constructs of LLVM bitcode like branch, select and GEP instructions. Information about loops and other high-level control flow structures are not included in the low-level bitcode representation. Sophisticated analyzes are necessary to recover the required high-level information. Especially the conversion into goto-free code is complicated and in the case of irreducible control flow impossible without code duplications.

Beside the previously analyzed ones other potential target languages are the

```
#include shader.h

ShadingData CONSTANT data;
PointLight CONSTANT lights[6];

void inner_shade(){
  //read from texture
  int x = __ptx_ntid_x()*__ptx_ctaid_x()*THREAD_WIDTH_X
    +__ptx_tid_x()*THREAD_WIDTH_X;
  [...]
  COLOR_IN * inter = getPixel(data.tex1, x, y, data.w, data.h);
  float hitDistance = __half2float(*((short GLOBAL*)&inter->d));
  [...]

  for(int l=0; l<data.lights_n; l++){ // illuminance loop
    Point P_light = lights[l].position;
    Vector L_dir_norm = P_light - P;
    [...]

    //diffuse component
    float cosLight = Dot(L_dir_norm, N);
    if (cosLight >= 0.0)
      C_diffuse += Cl*cosLight;

    Vector h = (IN + L_dir_norm);
    Normalize(h);
    float dotLightRefl = Dot(N, h);
    if(dotLightRefl> 0)
      C_specular += pow(dotLightRefl,32);
  }

  Color result = Ci * (Ka + Kd  * C_diffuse + Ks * C_specular);
  int rgb = rgbaToInt(result.x,result.y,result.z,1.f);
  COLOR_OUT *out  = getPixel(data.texOut, x, y, data.w, data.h);
  *((unsigned int GLOBAL*)out) = rgb;
}
```

Figure 2.7: Extracts from a C++ phong shader, extended with GPU-specific properties. The semantic is equivalent to the RenderMan shader of Figure 1.1. The complete source code is shown in Section 6.2.

sparsely used GPU aware languages Brooke and Ct as well as the GPU shading languages HLSL, GLSL, and Cg. All of them have the same disadvantages as OpenCL. The complex compilation from high-level representations to GPU assembly code eliminates most of the previously applied optimizations. Meanwhile, AMD offered the *close to the metal* (CTM) low-level programming interface for AMD GPUs. CTM allowed to access the native instructions set of the GPU. But

CTM was short lived and is now deprecated. The CAL IL is now used as the intermediate representation. CAL handles control flow with if-then-else construct, switch expressions, and explicit loops. Therefore, it is as cumbersome to generate CAL code from LLVM bitcode as to generate OpenCL code.

The idea of this thesis is to develop a code generator for a powerful customizable compiler. Only the low-level design, the degree of architecture independence, and forward compatibility as well as optimization possibilities of PTX can apply the compiler's features to GPUs in a satisfactory way. The target dependency of PTX to NVIDIA GPUs of course is a huge disadvantage. However, NVIDIA ships the majority of GPUs on the GPGPU market [Kow08], which reduces the fraction of PTX incompatible GPUs and improves chances for PTX to become a common intermediate GPU language and

Chapter 3
Code Generator Implementation

3.1 Compilation Pipeline

Every LLVM frontend can be used to generate LLVM bitcode which is later transformed to PTX code by the PTX code generator. The Clang frontend is the most advanced frontend. It compiles C-family languages to LLVM bitcode and supports the three language extensions of Section 2.3. In combination with the LLVM compiler it enables the following compilation pipeline: GPU-specific C++ code is compiled to extended LLVM bitcode by Clang. Common LLVM passes as well as custom GPU-specific passes perform desired optimizations on the intermediate bitcode representation. Finally, the PTX code generator transforms the optimized bitcode into PTX GPU code. This pipeline is visualized in Figure 3.1.

$$\begin{tikzcd}
\text{extended C++ code} \arrow[r, "\textit{Clang}"] & \text{LLVM bitcode} \arrow[loop above, "\textit{optimization passes}"] \arrow[r, "\textit{PTX code gen.}"] & \text{PTX code} \arrow[r, "\textit{CUDA runtime}"] & \text{NVIDIA GPU machine code}
\end{tikzcd}$$

Figure 3.1: Compilation pipeline of the PTX code generator from high level C++ code to PTX assembly code

The previously shown phong surface shader exposes this compilation pipeline. Figure 2.7 shows C++ source code which is semantically equivalent to the RenderMan phong shader of picture 1.1. Figure 2.1 shows the intermediate bitcode representation. The final PTX code is shown in Figure 2.4. Chapter 3.5 provides a language extension which defines a fixed exchange format between all compilation pipeline components. The language extension convention is obeyed by the Clang frontend, the optimization passes, and the PTX code generator. All handwritten shaders and test-functions of this thesis are written in C++ and are compiled to PTX code by the previously described pipeline.

3.2 Objective

The purpose of the code generator is to transform LLVM bitcode into PTX code. The code generator should be able to transform arbitrary LLVM bitcode programs into valid PTX code. This enables the PTX code generator to transform every program to PTX code that was written in a language for which an LLVM frontend exists. The semantic of the program must be preserved during code generation. The machine code of deterministic LLVM bitcode programs must compute exactly the same results on all target platforms. The code generator is required to convert every LLVM bitcode instruction into appropriate PTX instructions. However, due to the limited functionality of current GPUs not every LLVM bitcode program can be mapped directly. Function calls need to be non-recursive and direct, indirect calls are not supported on current GPUs. For bypassing these restriction by continuations see Section 5. The generated code will be verified in Section 4.1 by comparing the results of test-functions generated by the existing x86-LLVM backend and the PTX code generator, executing it on the CPU and GPU, respectively.

It should not only be possible to generate code from general LLVM bitcode programs, but also to access the whole functionality of GPUs. Every GPU feature which is supported by PTX should be accessible through LLVM and the PTX code generator. The source language as well as the intermediate LLVM bitcode representation must be capable of describing all GPU-specific features. GPUs provide a lot of special features which are not implemented on traditional CPUs (Section 2.4). LLVM bitcode is designed to represent programs targeted for CPUs and akin architectures and therefore lacks representations for GPU-specific features. There are possibilities to extend LLVM bitcode by intrinsic functions and address space annotations (see Section 2.3). Section 3.5 defines a language extension convention for GPU features. This convention serves as an interface between frontends, optimization passes and the PTX code generator.

This thesis focuses on the support of commonly used PTX language constructs. Instructions that offer additional performance like fused arithmetic operations are a target of future work. Table 3.1, Table 3.3, and Table 3.2 separate supported and unsupported instructions. The code generator provides a framework which allows to implement unsupported features analogously.

Beside soundness and feature support, code performance is the third objective of the PTX code generator. Architectures of GPUs and CPUs differ strongly.

Some programs run faster on a CPU and others are more suited for GPUs. Thus, it only makes sense to compare the performance of the generated code to other GPU programs, for example CUDA applications. Ideally, the performance of the generated PTX code is as good as the PTX code generated by the nvcc compiler from an analogous CUDA program.

As described in Section 2.6 the nvcc compiler compiles CUDA programs in a two step process: from CUDA to PTX and from PTX to GPU assembly code. The LLVM compiler and the PTX code generator replace the first step which is the conversion from a high-level language to PTX. Thus, the LLVM compiler needs to perform all optimizations which are usually performed by the nvcc compiler in order to be competitive. The nvcc compiler runs common optimizations and additional GPU-specific optimizations. They utilize the great processing power of GPUs, the SIMT execution model and the different memory space characteristics. NVIDIA has not published any details about performed optimizations. However, the PTX code analysis in Section 4.3 reveals some of them. Such optimizations are part of the compiler middle-end and they are separated from the code generator. The implementation of GPU-specific optimization passes and the adaption of exiting passes is not part of this thesis. But the PTX code generator must allow to add such optimizations retrospectively. This is ensured by the exhaustive language extension convention which covers all PTX features.

The second compilation step—PTX to assembly code—performs late-optimizations like register allocation, dead code elimination, and instruction scheduling (see Section 2.6). This step is also applied in the PTX code generator pipeline. The PTX code generator can expect that all of these optimizations run accurately. It does not need to care about the definition of unneeded registers and instructions.

3.3 Design

There are two ways to implement an LLVM backend. Either the target independent code generator is used or it is implemented as a general LLVM Pass. The target independent code generator provides code generation utilities and various late optimizations which can be adapted for the individual target. Amongst others, register allocation, scheduling and instruction selection are included. It requires to define abstract *target description* and *instruction description* classes. They define important properties and aspects of the target architecture and its instructions in a general way. Particular aspects which can not be represented in this general fashion must be specified by *custom passes*. The final code emittance depends on the *instruction* and *assembly printer* classes. The *TableGen* tool can be used to automate a lot of the instructions description process. However, the programming effort for the target independent code generator is immense. The existing X86, PowerPC, and ARM backends require more than $45,000$, $25,000$, and $40,000$ lines of code, respectively. In comparison the CBackend requires only around $3,500$ lines, including some dispensable duplicated code parts.

On the contrary, the custom pass variant requires to implement many features by hand that are provided by the target independent code generator. However, the conversion to PTX code is special in this regard. As previously described, the conversion from LLVM bitcode to PTX code is only a transition from one intermediate representation into another. PTX code is the intermediate representation of the nvcc compiler. Late optimizations are performed during PTX to GPU assembly code generation. Assuming that these passes are accurate and efficient, the PTX code generator does not need to perform them beforehand. Thus, most of the features provided by the target independent code generator are useless. The few remaining features do not justify the cumbersome specification and adaption process of the target independent code generator. The implementation of the PTX code generator as a custom LLVM pass is simpler. Thus, this approach is followed here.

Generally, LLVM bitcode can be transformed to PTX code instruction-wise. The PTX code generator can iterate over each LLVM bitcode instruction and transform it into equivalent PTX instructions. However, disparities between both languages need a special handling. The following section works out all disparities. Some of them are handled during code generation, others are eliminated in pre-code-generation passes. The code generator is implemented as a sequence of LLVM passes. Each pre-code-generation pass takes LLVM bitcode as input and transforms it into a more convenient format. Section 3.6 and 3.7 describe these passes in more detail. Only the last executed pass actually generates code. It iterates over the LLVM bitcode and outputs corresponding PTX code. The final PTX code is stored in a string stream which can either be written to a file or directly stored in memory, allowing static as well as just in time compilation.

3.4 LLVM Bitcode and PTX Comparison

The disparities of LLVM and PTX code can be divided into two classes A and B. Class A consists of disparities which can be handled by pre-code-generation passes, which transform LLVM bitcode into a more suitable format. For example, LLVM Instructions and intrinsics which are not directly supported by PTX are replaced by simpler supported instructions. Disparities of class B are handled directly during code generation. Either it is not possible to perform the transformation on LLVM bitcode because LLVM bitcode is too restrictive. Or it is more suitable to handle the disparity during code generation. Most of the disparities of class B also require a bitcode language extension convention (see next section).

Class A disparities

- The **address calculation** of struct and array elements is handled differently. LLVM provides *get element pointer* (GEP) instructions, which expect two types of arguments. The base address of pointer type and integer element indices. The underlying address calculation is hidden. In PTX structs and arrays are not typed; they are defined as arrays of bit types. There is also no GEP equivalent and no pointer type. Address calculations are hard-coded by standard integer arithmetic. The original sub-types define sizes and offsets of the fields.

- The base address of **global variables** is known at code generation time. LLVM therefore treats pointers to global variable base addresses as constants. This is not the case in PTX. Global variable addresses need to be moved to registers first and constant offsets need to be added by normal integer arithmetic instructions.

- High-level **utility functions** like exp and pow are represented in LLVM by intrinsics. PTX and the underlying GPU hardware also provide some mathematical utility instructions like ex2 and log2, but others are unsupported. These unsupported instructions need to be approximated by simpler instructions. For example the exp instruction is approximated by a multiplication and an ex2 call.

- Both LLVM and PTX support **vector types**, however PTX is more restrictive. LLVM supports arbitrary vector widths and almost every instruction can operate on vector operands. In PTX only loads, stores, texture fetches, and extract element instructions can operate on vectors. Every other instruction is restricted to scalar values. Also the vector width is restricted to 2 and 4. Vectors with 3 elements can be handled by four element vectors.

- PTX supports conditional execution of instructions. Each instructions can be marked with a **predicate** flag. Instructions with a false-predicate are excluded from execution. LLVM bitcode contains no predicate representation. Conditional execution must be implemented by conditional branches.

The handling of class A disparities by pre-code-generation passes is captured in Section 3.6.

Class B disparities

- LLVM bitcode fulfills the **SSA property**, each variable is only assigned once. So called PHI-instructions are used to determine the right value at convergent control flow nodes. PTX does not restrict the definition of virtual registers and contains no PHI-instruction equivalent.

- PTX distinguishes between **signed and unsigned integers**, sign dependent instructions behave differently depending on the argument type. The source and operand types must be specified explicitly. For example a division of two signed integers performs a signed division and unsigned arguments indicate an unsigned division.

  LLVM does not distinguish between signed and unsigned integer types in general. Instructions which differ, depending on the signed or unsigned argument type, are provided in two versions. For example there is a div instruction for signed division and an udiv instruction for unsigned division. In both cases the arguments are integers.

- LLVM allows to specify primitive types with **arbitrary bit sizes**. PTX primitive types are generally restricted to 16, 32 and 64 bit. Load and store instructions also allow 8 bit types.

- PTX provides access to the different **memory spaces** of NVIDIA GPUs: global constant, shared, local and texture memory. LLVM also provides an address space qualifier for pointers and global fields. But their semantic is not fixed. Each code generator can handle the address space qualifier differently. The adherence of the convention of Section 3.5 is necessary for the PTX code generator.

- PTX distinguishes between **kernel and device functions**. Kernel functions define entry points. They are invoked by the host system and are restricted to the void return type. Device functions are not visible to the host system, they are only accessible from kernel functions and other device functions. This is a GPU-specific feature and is not supported by LLVM explicitly. Again a convention must be defined.

- **GPU-specific instructions** which are implemented in hardware are not natively supported by LLVM. Examples are texture look-ups, synchronization instructions, and fused arithmetic instructions like muladd.

Beside these disparities both languages are similar. In particular the memory model is the same. Both languages operate on virtual registers. Register allocation is performed during PTX to GPU assembly compilation and is therefore not required during PTX code generation. The similarities of both languages allow an elementary mapping from LLVM bitcode constructs to PTX instructions. Optimizations can be performed on the bitcode representation. The mapping does not incorporate complex transformations which potentially eliminate previous optimizations.

3.5 GPU Language Extension Convention

The previously listed disparities do not allow to represent the whole functionality of PTX in LLVM without language extensions. This section defines an exhaustive convention, it includes language extensions for the class B disparities. The convention only extends the bitcode semantically, no syntactical extension is required. The convention serves as a fixed interface between frontends which generate GPU-specific LLVM bitcode, optimization passes, and the PTX code generator. This convention can also be used by newly developed GPU code generators, e.g. an OpenCL code generator.

The PTX code generator needs to decide on the function type. It uses the following mapping from arbitrary LLVM bitcode functions to the PTX**-kernel** and PTX**-device function** type. The mapping is based on the number of calls and does not require any additional information. Functions defined in a LLVM module which are called at least once are mapped to device functions, because PTX does not allow to call kernel functions out of a different function. All other functions, with no calls, are potential entry points for the host system, they are mapped to kernel functions. bitcode functions with non-void return type and no calls can not be called from the host CPU nor from device functions. Thus, they are never used. They are handled as device functions for the sake of completeness. It is also possible to ignore them completely during code generation. But it is sometimes beneficial to allow the inspection of PTX code generated from unused test functions.

The address space attribute of LLVM bitcode pointer types is used for the five different **memory spaces** of PTX. In general memory allocated by alloca instructions is part of the function stack frame. PTX does not have an explicit stack. Each thread has a local memory space region. It can be regarded as a stack of depth 1. A dynamic stack depth is not possible because the local memory region size must be defined at compilation time. A stack depth of 1 is sufficient since PTX functions are non-recursive. Memory allocated by alloca instructions automatically resides in the default address space 0. Thus, memory space 0 is mapped to local PTX memory. The remaining address spaces can be mapped arbitrarily. Address space 1 is mapped to shared, address space 2 to global, address space 3 to constant, and address space 4 to texture memory.

As previously described the type-systems of LLVM bitcode and PTX differ. But no language extension is required for them. The LLVM **integer type** is mapped to the unsigned integer type of PTX. This way every PTX integer instruction operates on unsigned integers. Only the specific LLVM signed instructions like sdiv are mapped to PTX instructions marked with the signed type of appropriate size. The non-strict type system of PTX allows such implicit casts.

The PTX code generator promotes **unsupported bit-sizes** to the smallest supported bit-size which is greater than the requested bit-size. This increases the computation range and potentially changes the program semantic. Programs which depend on overflows and related side effects must be adapted by hand. Conditional expressions which simulate overflows can be inserted automatically. However, such adaptions significantly reduce performance. It is more vulnerable to rewrite such programs without overflow dependencies.

All other **GPU-specific features** which are not natively supported by LLVM like special registers, texture look-ups, and synchronization methods are represented by intrinsic functions. Basically, the name of intrinsic functions is equivalent to the corresponding PTX instruction name. This direct mapping is not possible for synchronization instructions, special register accesses, and texture fetches, which depend on vector types. Tables 3.1, 3.3, and 3.2 define the complete mapping of PTX instructions to intrinsic functions. The mapping includes every instruction form the PTX ISA 1.4 [NVI10b] which has no counterpart in LLVM bitcode. Intrinsics which are currently not utilized by the PTX code generator are shaded in gray.

According to the previously described compilation pipeline (Section 3.1), C++ code can be converted to LLVM bitcode fulfilling the defined conventions with the LLVM Clang frontend. The address space annotation is set through the C attribute *__address_space(*$\{0,1,2,3,4\}$*)*. All other features are represented by intrinsic functions which are defined in a header file (see Code Appendix 6.1). This header file also provides definitions for the local, global, constant, shared, and texture memory space which resolves the explicit use of the *__address_space()* annotation. Using this header the source C++ code is syntactically similar to CUDA code. This similarity eases ports from original CUDA applications to extended C++ programs.

3.6 Pre Code Generation Passes

The PTX code generator incorporates three custom passes. Problems which arise from disparity of class A (marked in **bold**) are solved by them. First, the *SetGPUAddressSpace* pass handles the different available PTX **address spaces**. The

PTX Instruction	LLVM Intrinsic	Semantic
.sreg .u32 %tid.{x,y,z}	i16 __ptx_tid_{x,y,z}()	thread identifier within a CTA
.sreg .u32 %ntid.{x,y,z}	i16 __ptx_ntid_{x,y,z}()	number of thread IDs per CTA
.sreg .u32 %laneid	i16 __ptx_laneid()	lane within a warp
.sreg .u32 %warpid	i16 __ptx_warpid()	warp within a CTA
.sreg .u32 %nwarpid	i16 __ptx_nwarpid()	number of warp identifiers
.sreg .u32 %ctaid.{x,y,z}	i16 __ptx_ctaid_{x,y,z}()	CTA identifier within a grid
.sreg .u32 %nctaid.{x,y,z}	i16 __ptx_nctaid_{x,y,z}()	number of CTA ids per grid
.sreg .u32 %smid	i16 __ptx_smid()	processor identifier
.sreg .u32 %nsmid	i16 __ptx_nsmid()	number of processor identifiers.
.sreg .u32 %gidid	i16 __ptx_gridid()	grid identifier
.sreg .u32 %clock	i32 __ptx_clock()	32 bit Cycle counter
.sreg .u32 %pm{0,1,2,3}	i16 __ptx_pm_{0,1,2,3}()	performance monitoring counters

Table 3.1: Mapping from special PTX registers to LLVM intrinsics.

lowerInstruction pass simulates LLVM **utility functions** which are not supported by PTX, by adequate instructions. It also replaces **GEP instructions** by arithmetic instructions. Finally, the *PolishBeforeCodegen* pass cleans up useless constructs which were introduced during previously applied passes.

The utilization of **predicated** PTX instructions requires a pre-code generation-pass which determines program parts which should be predicated. Conditional branches to small basic blocks are potential candidates. This pass cannot operate on the bitcode representation exclusively, because LLVM bitcode has no representation for predicated execution. Instead, an additional data structure, which contains predication information, must be created and passed to the code generator. Such a pass is not necessarily required since conditional branches can be used instead. It is disputable if predication really increases performance since the execution of conditional branches of threads inside one warp comes close to predicated execution (see Section 2.4). At least the additional branch instruction can be avoided by predication. The support of predicated execution by the PTX code generator and an analysis of the performance gain is left for future work.

Vector types are currently not handled specifically. They are supported as far as PTX supports vector types. It is easy to split up unsupported LLVM vector operations into a stream of sequential operations of the same type. An LLVM *FunctionPass* which is executed before code generation can perform this transformation.

SetGPUAddressSpace Pass

In theory the LLVM address space attribute allows for a simple and clean handling of the different PTX address spaces. The compilation pipeline, depicted in Figure 3.1, uses the Clang LLVM frontend. It is capable of generating arbitrary

PTX Instruction	LLVM Intrinsic	Semantic
bar.sync d	void __bar(const i32 d)	barrier wait at d
membar.{gl, cta}	void __membar_{gl, cta}()	memory barrier
trap	void __trap()	perform trap operation
brkpt	void __brkpt()	suspends execution
pmevent c	void __pmevent(i32 c)	performance monitor event (c is a constant index)
atom{.global, .shared}.op.type	typeC __atom_{op}(void* a, typeC b)	atomic operation
red{.global, .shared}.op.type	void __red_{op}(void* a, typeC b)	reduction operations on global and shared memory
vote.{all, any, uni}.pred d, a	bool __vote_{all, any, uni}(bool a)	vote across thread group
op = {and, or, xor, cas, exch, add, inc, dec, min, max}		
type = {b32, b64, u32, u64, s32, f32}, typeC = {i32, i64, float}		

Table 3.2: Mapping from PTX synchronization and atomic instructions to LLVM intrinsic functions. Instructions which are currently not implemented are shaded in gray.

PTX Instruction	LLVM Intrinsic	Semantic
tex.{1d,2d,3d} d, [a, c]	<4xfloat> __tex1D(float* a, float cx), <4xfloat> __tex2D(float* a, float cx, cy), <4xfloat> __tex3D(float* a, float cx, cy, cz)	one-, two-, or three-dimensional texture fetch
sqrt.{approx.f32,rn.f64} a	ftypeC __sqrt(ftypeC a)	$\sqrt{a}$
sin.f32 a	float sinf(float a)	$\sin(a)$, a in radians
cos.f32 a	float cosf(float a)	$\cos(a)$
lg2.f32 a	float __lg2(float a)	$\log_2(a)$
ex2.f32 a	float __ex2(float a)	$\exp_2(a)$
sad.itype d, a, b, c	itypeC __sad(itypeC a, b, c)	$\|a-b\|+c$
mad.rn.{itype, f32} d, a, b, c	{itypeC, float} __mad({itypeC, float} a)	$a*b+c$
fma.rn.f64 d, a, b, c	double __mad(double a)	$a*b+c$ (64 bit)
abs.{itype, ftype} a	typeC __abs(typeC a)	$d=\|a\|$
min.{itype, ftype} a, b	typeC __min(typeC a, b)	minimum of a and b
max.{itype, ftype} a, b	typeC __max(typeC a, b)	maximum of a and b
neg.{itype, ftype} a	typeC __neg(typeC a)	$-a$
rcp.{approx.f32,rn.f64} a	ftypeC __rcp(ftypeC a)	$1/a$
rsqrt.{approx.f32,rn.f64} a	ftypeC __rsqrt(ftypeC a)	$1/\sqrt{a}$
not.{b16, b32, b64} d, a	itypeC __not(itypeC a)	bit-wise negation
cnot.{b16, b32, b64} d, a	itypeC __cnot(itypeC a)	logical negation
ftype = {f32, f64}, typeC = {float, double}		
itype = {u16, u32, u64, s16, s32, s64}, itypeC = {i16, i32, i64}		
type = itype ∪ ftype, typeC = itypeC ∪ ftypeC		

Table 3.3: Mapping of arithmetic PTX instructions for which no LLVM instruction counterpart exists. Instructions which are currently not implemented are shaded in gray.

address space attributes for pointers and global variables. Every existing LLVM pass is aware of this distinction. Finally, the code generator can easily generate memory access instructions with the corresponding address space qualifier. Currently, in some cases the Clang frontend does not generate correct memory attributes. In particular, Clang has problems with C++ classes, copy constructors, and function overloading. It either generates incorrect code or fails completely during compilation.

The *SetGPUAddressSpace* pass circumvents this restriction by a naming convention for variables and pointers. If an LLVM variable resides in or a pointer points to the default memory space zero its name is inspected. Names starting with the key words *__ptx_global*, *__ptx_local*, *__ptx_constant*, and *__ptx_texture* are mapped to the corresponding address space value. This pass changes the address space type of such variables, pointers, and all occurring uses to the requested address space value. The pass is implemented as an LLVM *ModulePass*.

This naming convention is only a workaround and is not included in the previously defined language extension convention. Once the Clang frontend handles address space attributes correctly this pass can be removed.

However, related transformations might still be beneficial. Most of the time global variables reside in the global memory space. The global address space is the preferred default in this case. On the contrary memory regions allocated with alloca instructions most of the time are constrained to access from the allocating thread. The local or shared memory space is the most convenient choice in this situation. Currently, the default memory space 0 is mapped to the local PTX memory space. Clearly, the programing effort is reduced by a pass which moves global variables with unspecified address space attributes to the global address space. Ideally, the compiler should make the decisions on the address space. The development of such optimization passes is a research topic on its own.

InsertSpecialInstruction Pass

The *InsertSpecialInstruction* pass replaces LLVM features, which are not supported by PTX, with simple adequate instructions. Mathematical functions like exp, log, and tan as well as GEP instructions are handled. The transformation is performed on LLVM bitcode as an LLVM *BasicBlock* pass. PTX assembly insertions are not necessary. The mathematical functions are simplified according to the equations seen in Figure 3.2. The basic trigonometric functions sin, cos, $\log_2$, and $\exp_2$ are supported by PTX.

Each LLVM GEP instruction is simplified iteratively on its nested structs and arrays. The iterative algorithm starts at the up-most nesting level and descends one level per iteration. The iteration is completed as soon as a primitive type is reached. At each level, the algorithm calculates the offset between the accessed

$$\begin{aligned}
\mathrm{pow}(a,x) &= a^x = (2^{\log_2(a)})^x = 2^{\log_2(a)*x} = \exp_2(\log_2(a)*x) \\
\exp_e(x) &= e^x = (2^{\log_2(e)})^x = 2^{\log_2(e)*x} = \exp_2(\log_2(e)*x) \\
\log_e(x) &= \frac{\log_2(x)}{\log_2(e)} \\
\tan(x) &= \frac{\sin(x)}{\cos(x)} \\
\cot(x) &= \frac{\cos(x)}{\sin(x)}
\end{aligned} \tag{3.1}$$

Figure 3.2: Computation of trigonometric functions by instructions which are implemented in hardware on current GPUs.

element and the parent structure based on the structure layout and the element indices. The GEP instruction provides an element index for every nesting level. This index can be constant or dynamic. Computations which involve dynamic indices must be evaluated at runtime, constant indices are addressed during compilation time.

At the beginning, this pass inserts a pointer to integer conversion instruction, which converts the base address of the accessed structure. A pointer *offset_dyn* to the last dynamic computation is maintained during each iteration step. It is initialized with the inserted pointer to integer instruction. An offset variable (*offset_const*) is also defined and initialized to zero. This variable records the constant offset to the base address of the GEP instruction.

During each iteration step the minor offset between the accessed element and the first element of that level is calculated. The minor offset depends on the element size, a potential padding and the index of the accessed element according to the equation seen in Algorithm 1. Constant element indices and the resulting minor offsets are known at compilation time. In this case the minor offset is simply added to the *offset_const* variable. Minor offsets of variable indices must be determined at runtime. Arithmetic instructions are inserted according to Algorithm 2. The final result is added to the last dynamic calculation, it is referenced by *offset_dyn*. Afterwards the reference is set to the newly inserted add instruction.

Finally, the value of *offset_const* is added to *offset_dyn* by an add instruction. The final result is converted back to pointer type. The inserted conversion instructions will be mapped to *no-operation* (NOP) instructions during code generation. However, they cannot be omitted because LLVM bitcode is type safe and requires such explicit type conversions.

Algorithm 1 Offset calculation for structures

```
for $i = 1$ to $elemIndex - 1$ do
    $padding = (align - (elemSize(i) \bmod align)) \bmod align$
    $offset \mathrel{+}= elemSize(i) + padding$
end for
```

Algorithm 2 Offset calculation for arrays

```
$padding = (align - (elemSize(0) \bmod align)) \bmod align$
$offset = (elemSize(0) + padding) * elemIndex$
```

The handling of GEP instructions before code generation allows to run general as well as custom optimization passes afterwards on the offset calculation. For example consecutive mul and add instructions can be combined to fused muladd instructions. Currently, only the available commutative simplification, constant propagation, and dead code elimination passes are executed.

The constant propagation pass reveals the problem of global variable base addresses—which are constant in LLVM—as described in Section 3.4. The constant propagation pass combines the constant arithmetic operations with the global base address into constant expressions which must be evaluated at compilation time. This is not possible during PTX code generation, because the explicit base address is not known. Also it is not possible to express the constant expression in PTX because global base addresses are not considered as constants by PTX.

To avoid this problem the *InsertSpecialInstruction* pass inserts wrapper function calls around each global base address occurrence. The wrapper function is defined as external; it takes one argument of pointer type and also has pointer return type. Each global variable access is encapsulated by a call to the wrapper function with that global variable as an argument. The LLVM compiler has no information about the wrapper function, arbitrary computations could be performed inside this function. Thus, the compiler cannot treat the result of the wrapper function call as constant anymore. Hence, the encapsulated base address cannot be included in constant expressions. After the execution of arbitrary optimization passes the wrapper function will be replaced by the previously encapsulated global base address. The *PolishBeforeCodegen* pass performs this simple transformation.

It is also possible to handle GEP instructions during code generation. This however would forbid to run LLVM optimization passes on the generated calculations. The GEP code generation method would need to perform impractical individual optimizations in order to emit performant code.

PolishBeforeCodegen Pass

This LLVM ModulePass removes the previously inserted wrapper functions. Masking is no longer required during code generation. No additional optimization passes are executed in between this pass and the code generation pass. Thus, the base address handling of global variables cannot be affected anymore.

3.7 Code Generation Pass

This pass performs the actual conversion from LLVM bitcode to PTX code. It is implemented as a standard LLVM ModulePass, however it does not alter the bitcode representation but transforms it into corresponding PTX code. The PTX code is emitted as a string stream, which allows static as well as just in time compilation.

LLVM bitcode is represented in a hierarchical manner. The top level element is the so called module, which consists of global variables and function definitions. Functions are defined by their signature and body. The body is separated in sequential segments called blocks. Blocks finally contain a sequence of instructions, they start with a number of PHI-instructions and end with terminator instructions.

The final code generation pass processes each of these hierarchical elements by corresponding handlers. The visitor pattern[1] is used to handle the instruction level, its nodes form the leaves of this hierarchical representation.

First, the module handler *doInitialization()* is called for every module. It prints out global variables and function declarations. Afterwards the handler *runOnFunction()* is called for every defined function of that module. The function handler prints the function signature and the first part of the function body, consisting of the declaration of required registers. Thereafter the block handler *printBasicBlock()* is called for every block. The block handler starts by printing the name of the currently processed block as a branch mark for jump instructions. Afterwards the visitor method is called sequentially for every visitor method that is included in the current block.

This design induces a high-level of abstraction. Every hierarchical element just needs to handle its own local properties and additional tasks are propagated to the children. Most of the code generation is done on the instruction level. However, the visitor pattern distributes tasks to the visitor methods of each instruction type. Each visitor method only needs to care about local, instruction specific

[1]The visitor pattern is an object oriented design pattern which assigns algorithms to already existing objects. Consider *Design Patterns: Elements of Reusable Object-Oriented Software* [Gam94] for a detailed description.

properties which simplifies the code generation complexity.

The following passage describes all handlers and visitor methods in more detail. All problems arising from disparities of class B (Section 3.4) are solved. Each disparity type occurrence is typed bold during the following passage to provide a better overview.

doInitialization()

Global variable declarations are generally located at the beginning of PTX files, before function definitions. Thus, the *doInitialisation* method, which is called before every other method of that pass, is ideal for global variable declarations. This method iterates over each global variable of that bitcode module, determines the PTX address space by the mapping described in Section 3.5 and prints initializers wherever required.

Structs are not supported by PTX and need a special handling. The smallest primitive type is 8 bit wide and the bit-size of every other primitive is a multiple of 8 bit. Thus, the size of an arbitrary struct is divisible by 8. This observation allows to allocate the right memory amount for each struct by defining an array of 8 bit primitives with the appropriate size. Also the initialization of such structs must be done by an array of 8 bit values. Bit representations of primitives with a bit-size larger than 8 bit are split up in 8 bit chunks. Each chunk is passed to the initialization array as a single value. Nested structs and arrays are handled in the same way.

runOnFunction()

The *runOnFunction()* method is called for every function defined in the bitcode module. It starts by printing the function signature which includes the function type qualifier. The function type, either **kernel or entry function**, is determined by the convention described in Section 3.5. The body of each function starts with register definitions. The *runOnFunction()* method simply iterates over every instruction of the current function and prints sufficient registers. LLVM bitcode is in **SSA form**, each instruction requires one destination register. The PTX backend handles SSA-related PHI-instructions according to the naive method described in [Sre99]. The described mapping from SSA to assembly code requires an additional virtual register for every primitive PHI-instruction. For every PHI-instruction, a copy instruction is inserted before the corresponding branch instruction in the antecedent basic blocks. The PTX backend proceeds on the assumption that late optimizations, which optimize the register pressure, are performed by the NVIDIA driver. Thus, no sophisticated transformation out of SSA form like the advanced method from [Sre99] is required.

Arguments of entry functions reside in the shared memory space. Load instructions are inserted for them after the register definitions. Finally, the *printBasicBlock* handler is invoked for every basic block of the processed function.

printBasicBlock()

The *printBasicBlock()* method handles one basic block at a time. First, a branch mark is printed which can be targeted by branch instructions. Afterwards the appropriate visitor method is called sequentially for every instruction that is included in the current block.

Generally, visitor methods take an instance of the corresponding LLVM bitcode instruction as input and transform it into valid PTX instructions. The generated PTX instructions are appended to the output string stream. Due to the previously listed disparities some instructions require a non-trivial handling. The following section provides details about visitor methods which perform such non-trivial transformations.

visitBinaryOperator()

The *visitBinaryOperator()* handles various binary operations of integer, floating point, and binary type. The following operations are supported: *add*, *sub*, *mul*, *div*, *and*, *or*, *xor*, *shift*, and *rem*. The standard math operations operate on float, unsigned-, and signed-integer values. The logical operations operate on binary representations.

Some special cases must be handled. Mul operation require a specific rounding mode. Floating point divisions must be marked with the *.full* or *.approx* qualifier. Unsupported bit-sizes are promoted to larger, supported bit sizes. If no such type exists an assertion is thrown. Every visitor method performs bit-size promotion, even though it is not explicitly mentioned in the following documentation.

visitPHINode()

The handling of PHI-instructions is performed by the visitBranchInst() method to simplify matters.

visitBranchInst()

The *visitBranchInst()* method inserts move instructions for each PHI instruction in the target basic block. The move instructions copy relevant values to the temporary registers previously generated for each PHI instruction. Finally, the PTX branch instruction to the target basic block is printed.

Conditional expressions are handled by predicates. The previously described branch instructions are printed twice; First, predicated with the branch condition for the consequence. Second, predicated with the negated branch condition for the alternative. Predication is natively supported by PTX.

visitCmpInst()

The *visitCmpInst* method handles ordered and unordered signed integers, unsigned integers, and floating point comparisons.

visitCastInst()

The *visitCastInst* method handles four different cases. Bit-casts are implemented by move instructions. An unequal-to-zero *setp* expression is used to cast instructions with predicate destination operands. The *selp* instruction handles predicate source operands. Dependent on the source predicate, zero or one is selected. All other casts are handled by native PTX *cvt* instruction.

visitSelectInst()

The visitSelectInst handling depends on the operand type. Predicate operands are implemented by predicated move instructions. For every other type *selp* instructions are used.

visitCallInst()

The generation of call instructions is trivial. Only intrinsic functions, which represent **GPU-specific features**, need a special handling. The PTX code generator supports a lot of intrinsic functions, see Table 3.1, Table 3.3, and Table 3.2 for a complete listing. The *visitCallInst()* method contains hard-coded implementations for every supported intrinsic. Some of them consist of a single line, others define temporary registers and contain multiple instructions. For example the *__syncthreads()* intrinsic is simply mapped to the PTX code line *bar.sync 0*. Texture fetches belong to the class of more sophisticated intrinsics, up to four temporary register definitions are required.

visitAllocaInst()

The *visitAllocaInst()* method handles the allocation of arbitrary structs and arrays in the same way as global structs are represented. Structs are replaced by appropriately sized arrays with 8 bit elements. According to the language extension convention (Section 3.5) the alloca instruction reserves memory in the local **address space**.

visitLoadInst() and visitStoreInst()

Load and store instructions are the only PTX instructions which operate on 8 bit values. The load instruction loads values which are stored in 8 bit cells into promoted 16 bit registers such that subsequent instructions can operate on them. Analogously, values which are smaller or equal than 8 bit reside in promoted 16 bit registers. The store instruction stores the content of such registers into 8 bit memory cells.

The choice of the PTX address space is based on the address space attribute of the source and destination pointer of the load and store bitcode instruction. The address space value is mapped to the PTX address space according to the convention in Chapter 3.5. Loads and stores of vector types are marked with the appropriate vector key-word *.v2* and *.v4* (width two and four, respectively).

Chapter 4
Evaluation

The evaluation of the LLVM PTX code generator is done in three steps:

1. The basic functionality of the code generator is checked by a set of test functions. Each function is compiled to PTX GPU code and x86 CPU code for reference. The test functions are executed for various parameters and their results on both systems are compared.

2. The code generation of complex programs and its execution performance is verified with a PTX shader integration into a deferred shading setup of the OGRE renderer. OGRE is an open source rasterizer which is based on Direct3D and OpenGL. The generated code is compared to both native Cg shaders and handwritten CUDA shaders. The accuracy is analyzed by comparing the resulting images on a pixel per pixel basis.

3. For some selected shaders and test functions the performance of the generated code is compared to analogous, handwritten CUDA programs in more detail. Various kernel parameters like used registers, memory access efficiency and kernel execution runtime are profiled with the CUDA profiler.

All test functions and test shaders are written in C++. The LLVM Clang frontend is used to converted them to LLVM bitcode. Intrinsics are defined as external functions in a separate header which is inlined by all test programs. The whole functionality of PTX can be described in C++ code, mapped to LLVM bitcode using the language extensions of Section 3.5 and tested by extensive test cases.

4.1 Test Suite

The test suite is an automatic tester. It contains many separate test functions covering the majority of LLVM bitcode constructs and PTX-specific extensions with a variety of input parameters. The test functions are divided into 4 categories, each of them covers a different field of instructions:

1. **Basic Instructions:** The first category of test-functions tests arithmetic, binary, and trigonometric instructions for floating point and integer values of different bit size. Special cases verify the correct handling of signed, unsigned, and constant operands. Also binary conversion as well as cast instructions between all different types are covered. Function calls with different arguments and return values as well as struct returns are issued.

2. **Control Flow:** Nested loops containing breaks and returns are checked in this category. The correct conversion of LLVM PHI-instructions between basic blocks is included.

3. **Memory Access:** Loads and stores from and to local, global, shared, and constant memory are covered in this category. Test cases for alloca instructions are included. The correct access of nested structs, arrays, and vectors, including alignment and padding calculations is checked. Texture access is tested for one, two, and three dimensional textures.

4. **PTX-specific extensions:** This category covers PTX-specific special registers, synchronization instructions, and atomic operations. Also kernel calls issued from the CPU with different number of arguments and argument types including pointer types are covered.

The execution of the test functions on the GPU and CPU validates the generated code. The results of both instances are identical in the range specified by the *IEEE standard for floating-point arithmetic* [IEE08]. The test suite and the test functions are implemented in C++, Figure 4.1 shows two of them. Function *test_math()* covers mathematical instructions and gets executed with input values $d.f = 3.2$ and $d.i = 3$. The *test_cflow()* function tests control flow, input values are $d.i = 3$ and $d.i = 10$. Additional test cases and the data structure layouts are listed in Section 6.3, the whole test-suite covers 40 different cases.

The C++ test-functions are known to the test suite at compilation time. They are statically compiled to x86-code by the g++ compiler and to PTX code by the LLVM compiler, its C++ Clang frontend, and the equipped PTX code generator. The NVIDIA driver API is used to load, compile and execute the PTX code dynamically. The parameters and results are also copied to and from the GPU by the diver API.

The CPU has not the whole functionality of the GPU. It is not possible to execute every PTX feature on the CPU without additional effort. Different address spaces and intrinsic functions must be emulated by the CPU reference functions.

```
test_math(GLOBAL Data* d)
{
  float f  = d->f;
  float fi = d->i;
  d->fa[0]  = expf(f);
  d->fa[1]  = logf(f);
  d->fa[2]  = exp2f(f);
  d->fa[3]  = log2f(f);
  d->fa[4]  = sinf(f);
  d->fa[5]  = cosf(f);
  d->fa[6]  = sqrtf(f);
  d->fa[7]  = tanf(f);
  d->fa[8]  = floorf(f);
  d->fa[9]  = atanf(f);
  d->fa[10] = powf(f,fi);
}
```

(a) MathTest

```
test_cflow(GLOBAL Data* d)
{
  int tmp = 0;
  int is = d->i;

  for(int i=0; i<is; i++)
  {
    tmp += i;
    if(i>10)
      break;

    tmp += i;
  }

  d->f = tmp;
}
```

(b) ControlFlowTest

Figure 4.1: Two examples from the PTX test suite. (a) inspects math functionality and (b) test control flow constructs.

The test suite supplies specific intrinsic function definitions for the included test cases. For a general emulation of PTX code on the CPU consider the Ocelot project [Ker09].

The test suite was a helpful tool during the implementation of new language features and ensured the correctness of existent functionality. These code examples show the whole functionality of the PTX backend and provide a good starting point for its users.

4.2 OGRE Integration

The deferred shading demo [Gat09] of OGRE's sample pack is a deferred shading rasterizer using the rendering pipeline of OGRE for the pre-shading part. During this step, shading input parameters are determined per pixel and stored in the so called GBuffer[1]. The deferred shading demo stores the GBuffer in two 64 bit textures. The first texture contains the rgb pixel color and the specular coefficients, the second texture contains the normal and depth information. All values are stored as 16 bit floats. Additional global parameters like camera and light positions are directly passed to the shader as parameters. The shading language Cg

[1]A GBuffer is a two dimensional data-structure. Its size is equal to the output-image size. Each element contains shading information for its corresponding pixel.

is used for the deferred shading.

The PTX version replaces the deferred shading part by a PTX kernel call which performs the shading computations. The shader parameters are stored in constant memory. The kernel call and memory transactions are issued through the NVIDIA driver API. PTX shaders are loaded from text files, which are generated beforehand by the AnySL system [Kar10]. The generation of AnySL bitcode shaders is not part of this thesis but it is straightforward: Merely the definition of a short glue code file is necessary.

Additionally, a different GBuffer layout for AnySL shader is introduced, the three color components are replaced by uv-texture coordinates and a material id. OpenGL textures are not directly accessible by PTX kernels. For this additional memory copies from *frame buffers objects* (FBO) to *pixel buffers object* (PBO) are necessary. The mapping is performed via a *render buffer objects* (RBO) which ensures maximum memory throughput.

The performance of the generated code is evaluated by comparing the runtime of the original Cg shader with an equivalent handwritten C++ shader compiled to PTX by the PTX code generator. As a direct comparison some selected shaders are compared to handwritten CUDA equivalents.

Table 4.1 shows the comparison of selected shaders in frames per second. The additional overhead caused by memory copies between OpenGL and PTX-kernels is excluded in all columns marked with a star (*) for a better comparison. The light shader is a phong shader which supports a dynamic number of light sources and shadow calculation based on shadow textures. It is tested in a scene with 6 point light sources (see Picture 4.2b). All other shaders are tested in a simpler scene with a single point light (see pictures 4.2a). The shaders exclusively operate on the GBuffer and on global constants. The number of triangles is irrelevant since it does not influence the execution time. The wood and granite shaders are procedural shaders based on *perlin noise* [Per02]. Random numbers are stored in an array in constant memory space. The multi shader combines multiple shaders in a switch statement. The correct shader type is selected for each pixel, based on its material id. The multi shader incorporates different phong shaders, the procedural brick, granite, and wood shaders, a normal shader and a checker shader based on uv-coordinates.

The Cg shader outperforms the PTX shaders by a huge margin. It is executed in-pipeline like the preceding GBuffer generation process. PTX kernels and in-pipeline operations require a different GPU configuration. Context switches are required in between each invocation. In this case context switches are required before and after each PTX shader invocation. Each context switch causes additional overhead. Also the actual invocation of every thread consumes time. These two drawbacks are the main reason for the huge performance difference of Cg and PTX shaders. Already a blank shader which consists of an empty kernel has a

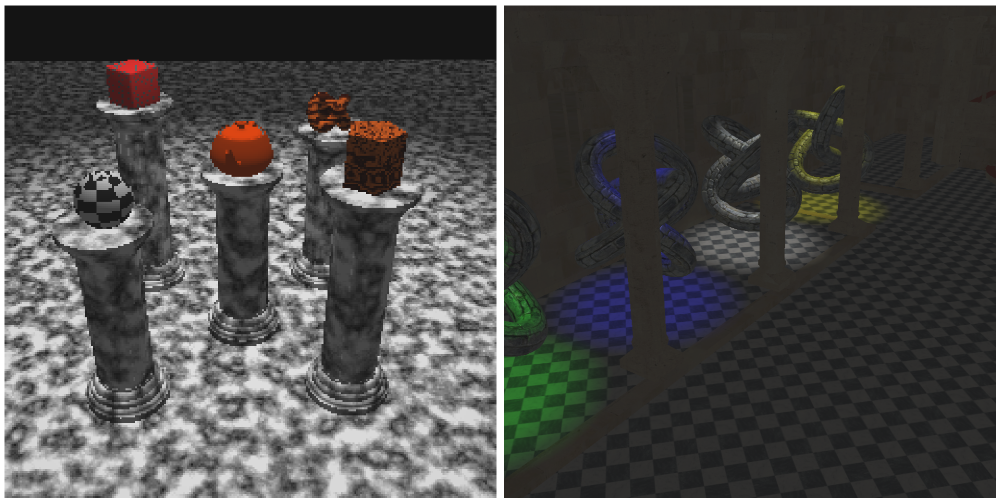
Shader	Resolution	NVCC	NVCC*	PTX	PTX*	Cg
light	800x600	70.6	146.3	76.5	175.5	260
	1024x768	63.3	124.0	71.5	156.9	260
	1600x1024	49.6	83.3	60.1	116.9	244
blank	32x32	-	-	85.8	199.4	260
	800x600	-	-	82.8	195.4	260
	1024x768	-	-	79.9	182.3	260
	1600x1024	-	-	74.2	155.4	260
wood	800x600	195	256	191	255	-
	1024x768	180	216	162	195	-
	1600x1024	126	148	115	135	-
granite	800x600	59.0	63.2	59.0	63.2	-
	1024x768	40.1	41.4	40.0	41.3	-
	1600x1024	21.5	22.1	21.5	22.1	-
multi	800x600	97.6	110.1	97.7	110.3	-
	1024x768	73.5	78.2	73.7	78.4	-
	1600x1024	44.7	47.0	44.7	47.2	-

Table 4.1: Runtime comparison of cg and PTX shaders. The PTX shaders are tested in two versions, compiled with the nvcc compiler and with the PTX code generator. All measurements are in frames per second. Only the light and blank shaders are implemented in cg. The memory copy overhead which is required for PTX and OpenGL-texture interoperability are excluded in columns which are marked with a (*)

limited frame-rate which is lower than the Cg light shader. The comparison to the nvcc compiler is more meaningful. Generally, the performance of PTX shaders generated by the PTX code generator is close to code generated by the nvcc compiler. Both versions scale well with the image resolution. For the phong shader the PTX code generator generates better code than the nvcc compiler. In every other case, both compilers are equally performant. Specific analyses of these runtime differences by the CUDA profiler are presented in Section 4.3.

The semantic of the generated code is verified by comparing the output image of an AnySL shader with a hand coded CUDA equivalent and the original Cg shader on a per pixel basis. Images of the compared test scenes are pictured in Figure 4.2. The resulting images look the same, but a pixel level comparison reveals minimal differences. The mean error between pixels of the AnySL-PTX shader and the Cg shader is 0.002 in the cathedral scene. The *root mean square* (RMS)[2] is 0.0036 and the maximum error is 0.306. Similar results are obtained in the museum scene. The most crucial differences originate from pixel artifacts in the Cg shader. The minimal differences probably originate from different rounding modes, precision bounds, instruction selections, and exponentiation implementa-

[2]The RMS is a measure of the magnitude of a varying quantity. It is defined as $err_{rms} = \sqrt{\frac{x_1^2+x_2^2+...+x_n^2}{n}}$.



(a) Museum (b) Cathedral

Figure 4.2: Pictures of the employed test scenes. The museum scene is illuminated by a single point light and is shaded with the multi shader. The cathedral scene is lit by 6 light source. Shadows are disabled in both cases.

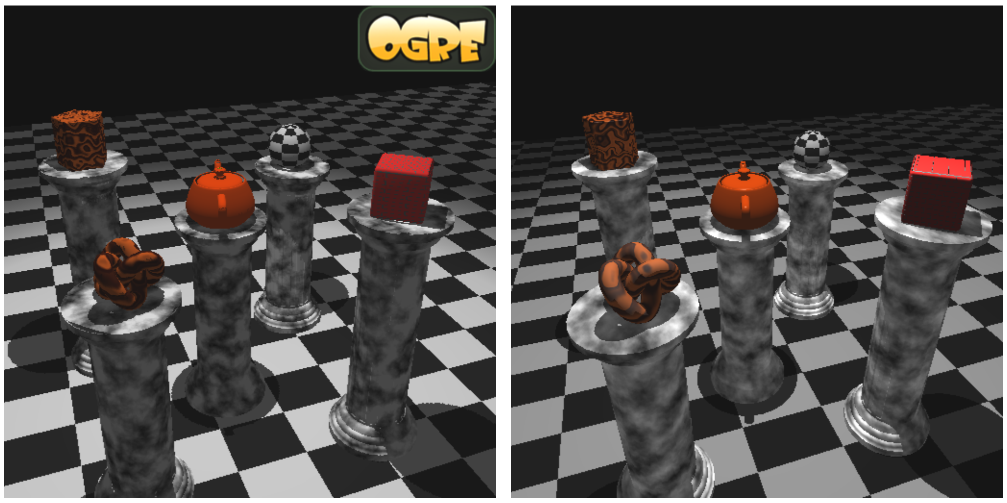
tions. The AnySL and the nvcc result images perfectly match in both scenes. The generated code of both systems is semantically identical.

The AnySL-PTX output images can also be compared to pictures generated by other AnySL capable renderers. Figure 4.3 shows a comparison to the RTfact ray-tracer. The resulting images visually look the same. However, due to the different rendering models—ray tracing and rasterization, arising precision disparities of both hardware systems, and non-uniform scene input formats, the images do not match on the pixel level.

4.3 CUDA Comparison

The correct code generation is the main aim of this thesis; no emphasis is put on the performance of the generated code. Nevertheless, a comparison of the performance of PTX kernels generated by the official nvcc compiler and kernels generated by the PTX code generator is of interest. The analysis shows the code performance, reveals drawbacks, and points out future improvements and research topics. It also delivers hints for optimizations applied by the nvcc compiler.

Table 4.2 and 4.3 show various kernel properties derived with the NVIDIA CUDA profiler. Two of the previously shown shaders, two samples from the CUDA SDK, and selected functions of the test suite are profiled. Each test is



(a) OGRE (b) RTfact

Figure 4.3: Comparison of the deferred shading OGRE integration to the RTfact ray-tracer. Shadow calculations are enabled.

implemented in CUDA and in extended C++, the semantic of both versions is the same. The CUDA implementation is compiled by the official nvcc compiler (version 2.3), the C++ code by LLVM and the PTX code generator. The CUDA profiler collects data about execution times, required shared memory, used registers, the number of executed instructions, number of divergent and non-divergent branches, coalesced loads and stores, and cache hits and misses. The generated statistic is an approximation of the overall resource consumption. Only a fraction of the executed threads is inspected. The measured values may vary depending on the thread scheduling, different control flows, and cache occupations. Every kernel is executed 10 times in a row. To prevent cold starts the first measurement is excluded. The shown tables contain the mean values of the remaining measurements.

As expected, the optimization passes of the nvcc compiler are more sophisticated and tuned for GPU architectures. The nvcc outperforms the PTX code generator in many cases. The following analysis and comparison of PTX kernels of both compilers reveals reasons for the disparity. The source code of the analyzed kernels is shown in Chapter 6.3. The CUDA SDK samples can not be shown due to license issues.

In the *phi* test, the PTX code generator implements the if-then-else construct with branch instructions. The kernel only requires 3 registers per thread. The nvcc compiler transforms the conditional statement into a select instruction. Both, the

Kernel	Runtime in *ms*	Occu-pancy	Reg. Count	Branch	Div. Branch	Instr. Count	Cache Hit	Cache Miss
nvcc_wood	242.14	0.5	25	4778	700	38717	7	8
gen_wood	239.26	0.5	22	4928	1044	43422	6	9
nvcc_light	74.21	0.375	38	2047	69	19114	3	6
gen_light	53.6	0.375	36	1399	69	16269	3	6
nvcc_granite	2064.5	0.375	36	17054	3010	114516	2	7
gen_granite	2055.7	0.5	27	20373	5629	132380	0	9
nvcc_multi	1285.7	0.312	41	17959	2863	123123	14	7
gen_multi	1287.7	0.375	35	19600	4784	133159	13	8

Table 4.2: Kernel profile of the examined shaders. Shaders which are compiled with the nvcc compiler have the prefix *nvcc*, PTX code generator shaders are prefixed with *gen*. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of active warps. Required registers are per thread.

consequence and the alternative basic blocks are executed independently from the conditional. Finally, the correct value is selected based on the if condition. This transformation is commonly known as *if-conversion* [All83]. In this case the optimization does not pay off. The register pressure is increased and execution time is not enhanced.

The *simple_phi* kernel shows that the instruction selection of the nvcc compiler is superior. It selects the negation instruction instead of a subtraction from zero operation and replaces floating point multiplications of factor two with an add instruction. The execution time is slightly increased. Also, the *phi9* kernel is optimized by a combined *muladd* instruction.

The nvcc compiler recalculates floating point operations in favor of free register slots in the *phi2* test case. The execution speed is almost unaffected. The LLVM compiler positions the evaluation of if conditions of inner loops, which are constant regarding the outer loop, before the outer loop. The number of live registers inside of both loops is increased but it avoids the recalculation of the conditional expression per iteration. In kernel *loop23* the recalculation pays off. The nvcc compiler outperforms the PTX code generator.

The *special register* test incorporates calculations with 16 bit values. The PTX code generator converts the extracted values to 32 bit types and after that performs arithmetic operations. The nvcc compiler computes in 16 bit registers. The number of occupied 32 bit registers is decreased. Two 16 bit registers fit into one 32 bit register.

In the *calculate, math, alloca,* and *signed operand* test kernels the nvcc compiler frequently reloads values from the global memory space. The register pressure is slightly reduced. However, the huge memory transaction overhead signif-

Kernel	Runtime in *ms*	Reg. Count	Branch	Div. Branch	Instr. Count	Cache Hit	Cache Miss
nvcc_matrixMul	17.024	14	274	2	3402	1	2
gen_matrixMul	43.264	14	1858	2	14164	1	2
nvcc_simpleTexture	34.496	9	1233	137	11253	0	3
gen_simpleTexture	35.584	10	1233	137	11804	0	3
nvcc_test_phi	16.35	4	0	0	448	0	0
gen_test_phi	16.256	3	130	2	646	0	0
nvcc_test_phi2	16.54	4	0	0	544	0	0
gen_test_phi2	16.448	5	130	2	749	0	0
nvcc_test_phi9	16.45	5	32	0	512	0	0
gen_test_phi9	16.384	4	226	2	745	0	0
nvcc_test_loop23	16.58	7	128	0	802	0	0
gen_test_loop23	17.888	8	322	2	1172	1	2
nvcc_test_loop13	968.26	8	99200	0	343008	2	1
gen_test_loop13	1094.91	8	147522	2	401156	2	1
nvcc_test_calculate	864	9	96	0	5922	0	0
gen_test_calculate	232.864	11	226	2	4505	0	0
nvcc_test_simple_phi	12.48	2	0	0	288	0	0
gen_test_simple_phi	12.608	2	130	2	473	1	2
nvcc_test_math	73.56	8	130	2	1615	0	0
gen_test_math	79.872	8	130	2	1746	0	0
nvcc_test_signedOp	148.96	10	160	0	3272	0	0
gen_test_signedOp	101.408	10	290	2	3490	0	0
nvcc_test_alloca	39.26	4	0	0	640	0	0
gen_test_alloca	38.464	4	130	2	806	0	0
nvcc_test_specialReg	4.48	3	1	0	17	0	0
gen_test_specialReg	4.576	4	0	0	0	0	0

Table 4.3: Selected test suite kernels from the test suite. Compiled with the nvcc compiler and the PTX code generator. The notation of Table 4.2 is applied.

icantly degrades the execution speed. In the case of shared memory, reloading values is more rewarding.

The nvcc kernel of the *matrixMul* CUDA-SDK-sample is extremely performant. It utilizes mul24 and mad instructions which are currently not supported by the PTX code generator. Additionally loop unwinding transformations reduce the number of branch instructions immensely. The loop unrolling LLVM pass is restricted to loop bodies without function calls. Modifications which allow to unroll loops including synchronization intrinsics are required in this example.

Table 4.2 includes an occupancy column. The occupancy of the nvcc kernels is low because of the higher register pressure. Although the code generator produces more cache hits divergent branches and instructions the *multi* and *granite* shaders are equally performant. The shader PTX kernels are too complicated for a manual analysis. The performance difference probably originates from the previously analyzed issues.

The influence of the .approx qualifier for floating point operations is obfuscated. In the *loop23* test case the use of approximated operation decreases the register pressure by one. In the *math* kernel the insertion of the approx qualifier increases the register pressure. The nvcc compiler uses approximated operations in both cases.

Chapter 5
Conclusion and Future Work

This thesis presented the design and implementation of the PTX code generator. Design decisions and implementation details are based on consolidated knowledge of the underlying hard- and soft-ware. The PTX code generator is not limited to graphics calculation but was developed with general purpose applications in mind.

We evaluated the performance and complexity of the PTX code generator in comparison to related work by specific test cases, the OGRE integration, and examples from the CUDA SDK. It generates correct code for a large range of LLVM programs. Only sparsely used bitcode constructs are unsupported. Also all relevant PTX instructions are utilized. The remaining unimplemented features can easily be integrated into the current code structure. Although no emphasis was put into code optimization it turned out that the generated code quality is similar to code compiled by the nvcc compiler. Some test cases revealed suboptimal code generation. The PTX code generator does not use combined instructions, generates inefficient basic block orders and occupies superfluous registers. However, it also outperforms the nvcc compiler in many test cases. The assumption that late-optimizations are applied during PTX to assembly compilation holds. The insertion of additional move instructions and the naive SSA deconstruction does not affect the performance. The custom backend approach is justified.

The unstable performance results of the nvcc compiler show the difficulty of automatically choosing the right optimization. Optimizations like if-conversion, value reloading, and the use of approximated floating point operations is only beneficial in particular situations. This emphasizes the demand for the customizable open source PTX code generator.

The following list of extensions has the potential to improve the PTX code generator regarding speed, code quality, and feature support. A pre-code-generation pass which splits up unsupported vector operands enables the PTX code generator to compile arbitrarily vectorized programs. Novel optimization passes can increase code quality in particular cases. A PTX frontend permits the optimization of arbitrary PTX programs. It would be possible to optimize PTX code generated with the available nvcc or OpenCL compilers by the LLVM compiler. An LLVM OpenCL frontend allows to replace the extended C++ code by a well specified language. A PTX frontend [Ker09] and OpenCL frontends [AMD09] already exists. It remains to connect them to the current compilation pipeline. A constraint tester which inspects the intermediate bitcode for illegal code constructs like recursion would ease debugging. Currently, every pipeline component relies on the adherence of the defined language extension convention by all predecessors. The restriction to non-recursive function calls is currently the biggest restriction of the PTX code generator, and also of other available GPU code generators. Recursive function calls are not supported by the currently available GPUs. However, it is possible to bypass this restriction with a software stack. The Optix ray-tracer implements such a stack by continuations [Par10]. An analogue implementation would allow the LLVM compiler to compile almost arbitrary programs to PTX GPU code.

Chapter 6
Code Appendix

6.1 PTX Intrinsic Header

```
#ifndef _PTX_INTRINSICS_H
#define _PTX_INTRINSICS_H

#define LOCAL    __attribute__((address_space(0)))
#define SHARED   __attribute__((address_space(1)))
#define GLOBAL   __attribute__((address_space(2)))
#define CONSTANT __attribute__((address_space(3)))
#define TEXTURE  __attribute__((address_space(4)))

extern unsigned short __ptx_sreg_tid_x();
extern unsigned short __ptx_sreg_tid_y();
extern unsigned short __ptx_sreg_tid_z();
extern unsigned short __ptx_sreg_ntid_x();
extern unsigned short __ptx_sreg_ntid_y();
extern unsigned short __ptx_sreg_ntid_z();

extern unsigned short __ptx_sreg_ctaid_x();
extern unsigned short __ptx_sreg_ctaid_y();
extern unsigned short __ptx_sreg_ctaid_z();
extern unsigned short __ptx_sreg_nctaid_x();
extern unsigned short __ptx_sreg_nctaid_y();
extern unsigned short __ptx_sreg_nctaid_z();

extern unsigned short __ptx_sreg_gridid();
extern unsigned short __ptx_sreg_clock();

extern unsigned short __ptx_laneid();
extern unsigned short __ptx_warpid();
extern unsigned short __ptx_nwarpid();
extern unsigned short __ptx_smid();
extern unsigned short __ptx_nsmid();
extern unsigned short __ptx_pm_0();
extern unsigned short __ptx_pm_1();
extern unsigned short __ptx_pm_2();
extern unsigned short __ptx_pm_3();

void __bar(const i32 d);
void __membar_gl();
void __membar_cta();
void __trap();
void __brkpt();
void __pmevent(i32 c);

extern float __half2float(short f);
extern void __syncthreads();

extern __m128 __ptx_tex1D(float* ptr, float coordinate);
extern __m128 __ptx_tex2D(float* ptr, float coordinate0,
                          float coordinate1);
extern __m128 __ptx_tex3D(float* ptr, float coordinate0,
                          float coordinate1, float coordinate3);
#endif
```

6.2 C++ Phong Shader Source Code

```
#include "shader.h"

ShadingData CONSTANT data;
PointLight CONSTANT lights[7];

extern "C" void inner_shade()
{

  //read from texture
  int x = __ptx_ntid_x()*__ptx_ctaid_x()*THREAD_WIDTH_X
    +__ptx_tid_x()*THREAD_WIDTH_X;
  int y = __ptx_ntid_y()*__ptx_ctaid_y()*THREAD_WIDTH_Y
    +__ptx_tid_y()*THREAD_WIDTH_Y;
  int x_max = __ptx_ntid_x()*__ptx_nctaid_x();
  int y_max = __ptx_ntid_y()*__ptx_nctaid_y();

  COLOR_IN * rgba  = getPixel(data.tex0, x, y, data.w, data.h);
  COLOR_IN * inter = getPixel(data.tex1, x, y, data.w, data.h);

  float hitDistance = __half2float(*((unsigned short GLOBAL*)&inter->d))
                      * data.farClipDistance;
  Normal N(__half2float(*((unsigned short GLOBAL*)&inter->a)),
          __half2float(*((unsigned short GLOBAL*)&inter->b)),
          __half2float(*((unsigned short GLOBAL*)&inter->c)));

  float x_relative = (float)x/(float)data.w;
  float y_relative = (float)y/(float)data.h;

  //wiev coordinates
  Vector IN = Vector((x_relative-0.5f)*2.f, (y_relative-0.5f)*-2.f, 1)
              * data.farCorner;

  //normalise dir
  Normalize(IN);

  Point P = hitDistance * IN;// +data.origin //world view
  Normalize(N);
  Normal Nf = FaceForward(N, IN);

  const float Kd = 0.6f;
  const float Ka = 0.2f;
  float Ks = 0.2f;

  Color C_diffuse(0.0f, 0.0f, 0.0f);
  Color C_specular(0.0f, 0.0f, 0.0f);

  // BEGIN_ILLUMINANCE_LOOP
  int n = data.lights_n;
  for(int l=0; l<n; l++)
  {
    void * P_light_ptr = &lights[l].position;
    Point P_light;
    P_light.x = (*(Point CONSTANT *)P_light_ptr).x;
    P_light.y = (*(Point CONSTANT *)P_light_ptr).y;
    P_light.z = (*(Point CONSTANT *)P_light_ptr).z;
    Vector L_dir_norm = P_light - P;
    float len_sq = Dot(L_dir_norm,L_dir_norm);
    float len = sqrtf(len_sq);
    L_dir_norm *= (1./len);

    Color Cl = Color(1,1,1);
    void * lightFalloff_ptr = &lights[l].attenuation;
    Vector lightFalloff;
    lightFalloff.x = (*(Vector CONSTANT *)lightFalloff_ptr).x;
    lightFalloff.y = (*(Vector CONSTANT *)lightFalloff_ptr).y;
    lightFalloff.z = (*(Vector CONSTANT *)lightFalloff_ptr).z;

    //spotlight falloff and attenuation
    void * dir_ptr = &lights[l].direction;
    Vector dir;
    dir.x = (*(Vector CONSTANT *)dir_ptr).x;
    dir.y = (*(Vector CONSTANT *)dir_ptr).y;
    dir.z = (*(Vector CONSTANT *)dir_ptr).z;
    float spotlightAngle = clamp2One(Dot(dir, -L_dir_norm));

    void * outer_ptr = (void *)&lights[l].spotlight_outerAngle;
    float outer = *(float CONSTANT *)outer_ptr;
    void * inner_ptr = (void *)&lights[l].spotlight_innerAngle;
    float inner = *(float CONSTANT *)inner_ptr;

    float spotFalloff = clamp2One((+spotlightAngle - inner)
                        / (outer - inner));
    float attenuation = 1.f/(1.f-spotFalloff);

    //diffuse component
    float cosLight = Dot(L_dir_norm, N); //NF
    if (cosLight >= 0.0)
      C_diffuse += Cl*cosLight/attenuation;


    Vector h = (IN + L_dir_norm);
    Normalize(h);
    float dotLightRefl = Dot(N, h);
    if(dotLightRefl> 0)
      C_specular += pow(dotLightRefl,32)/attenuation;

  }

  Color Ci(__half2float(*((unsigned short GLOBAL*)&rgba->a)),
           __half2float(*((unsigned short GLOBAL*)&rgba->b)),
           __half2float(*((unsigned short GLOBAL*)&rgba->c)));
  Color result = Ci * (Ka + Kd  * C_diffuse + Ks * C_specular);

  clamp2One(result.x);
  clamp2One(result.y);
  clamp2One(result.z);

  int out = rgbaToInt(result.x,result.y,result.z,255.f);
  COLOR_OUT *rgba_out  = getPixel(data.texOut, x, y, data.w, data.h);

  *((unsigned int GLOBAL*)rgba_out) = out;
}
```

6.3 Test Suite

```
#include "PTXIntrinsics.h"
```

```
#define ARRAY_N 64

typedef struct
{
  float f;
  char c;
  int i;
  char cc;
} DataStructInternal1;

typedef struct
{
  float f;
  DataStructInternal1 s;
  DataStructInternal1 sa[3];
  int i;
} DataStructInternal0;

typedef struct
{
  float fa[ARRAY_N];
  float f;
  int i;
  unsigned int u;
  char c;
  char ca[19];
  int ia[ARRAY_N];
  DataStructInternal0 s;
  double d;
  short half;
} DataStruct;


extern "C" void test_phi(GLOBAL DataStruct* data) {
  float a = data->fa[0];
  float b = data->fa[1];
    float x = a + b;
    float y = x * x - b;
    float z;

    if (x<y) {
        z = a+x;
    } else {
        z = a*a;
    }

    z = z+x;
    z = y-z;
    data->f = z;
}


extern "C" void test_phi2(GLOBAL DataStruct* data) {
    float x = data->fa[0] + data->fa[1];
    float y = x * x - data->fa[1];
    float z;
    float r;

    if (x<y) {
        z = data->fa[0]+x;
        r = x*x;
    } else {
        z = data->fa[0]*data->fa[0];
        r = x-data->fa[0];
    }

    z = z+x;
    z = y-z;

    data->f = z * r;
}

extern "C" void test_phi9(GLOBAL DataStruct* data) {
    float x = data->fa[0] + data->fa[1];
    float y = x * x - data->fa[1];
    float z = y;

    if (data->fa[0] < data->fa[1]) {
        z += data->fa[0];
    } else if (data->fa[1] < data->fa[0]) {
        z += data->fa[0]*data->fa[0];
    }

    z = z+x;
    z = y-z;

    data->f = z;
}

extern "C" void test_loop23(GLOBAL DataStruct* data) {
  float a = data->fa[0];
  float b = data->fa[1];
  float x = a + b;
  float y = x * x - b;
  float z = y;
  for (int i=0; i<1000; ++i) {
    z += a;
    if (z / a < x) break;
    else {
      z -= b;
      if (a > b) {
        for (int j=3; j<4500; ++j) {
          if (i == j) z /= -0.12f;
          if (z < -100.f) break;
          if (z < 0.f) {data->f = z; return;}
        }
        continue;
      }
      else {
        z *= z-y;
        if (b == a) {
          {data->f = z; return;}
        } else {
          ++z;
          break;
        }
      }
    }
  }
  z = z-y;
  data->f = z;
}

extern "C" void test_specialReg(GLOBAL DataStruct* data)
{
  int i =  __ptx_sreg_tid_x()
        + (__ptx_sreg_tid_y()*__ptx_sreg_ntid_x())
        + (__ptx_sreg_tid_z()*__ptx_sreg_ntid_x()*__ptx_sreg_ntid_y());
  if(__ptx_sreg_ctaid_x()>0)
    data->fa[i] = 0;
  else
    data->ia[i] = 1;
}


extern "C" void test_math(GLOBAL DataStruct* data)
{
  float f = data->f;
  float fi = data->i;
  data->fa[0] = expf(f);
  data->fa[1] = logf(f);
  data->fa[2] = exp2f(f);
  data->fa[3] = log2f(f);
  data->fa[4] = sinf(f);
  data->fa[5] = cosf(f);
  data->fa[6] = sqrtf(f);
  data->fa[7] = tanf(f);
  data->fa[10] = floorf(f);
  data->fa[12] = powf(f,fi);
}

extern "C" void test_loop13(GLOBAL DataStruct* data)
{
  float a = data->fa[0];
  float b = data->fa[1];
  float x = a + b;
  float y = x * x - b;
  float z = y;
  for (int i=0; i<1000; ++i) {
    z += a;
    if (z / a < x) z += a;
    else {
      z -= b;
      if (a > b) z -= b;
      else {
        z *= z-y;
        if (b == a) {
          for (int j=0; j<200; ++j) {
            if (i == j) z *= z;
            z += 13.2f;
          }
          z = a+3;
        } else
          ++z;
      }
      for (int j=0; j<100; ++j) {
        if (i < j) z += a;
        else z -= 13.2f;
      }
    }
  }
  z = z-y;
  data->f = z;
}

extern "C" void test_calculate(GLOBAL DataStruct* data)
{
  data->ia[0] = data->i + data->i;
  data->ia[1] = data->i - data->i;
  data->ia[2] = data->i * data->i;
  data->ia[3] = data->i / data->i;
  data->ia[4] = (unsigned)data->i << data->i;
  data->ia[5] = (unsigned)data->i >> data->i;

  unsigned int tmpi = data->i;
  data->ia[7] = tmpi >> tmpi;
  data->ia[8] = data->i % data->i;
  data->ia[9] = data->i & data->i;
  data->ia[10] = data->i | data->i;
  data->ia[11] = data->i ^ data->i;

  data->ia[20] = data->f + data->i;
  data->ia[21] = data->f - data->i;
  data->ia[22] = data->f * data->i;
  data->ia[23] = data->f / data->i;
  data->ia[24] = (unsigned)data->f << data->i;
  data->ia[25] = (unsigned)data->f >> data->i;


  data->fa[0] = data->f + data->f;
  data->fa[1] = data->f - data->f;
  data->fa[2] = data->f * data->f;
  data->fa[3] = data->f / data->f;

  data->fa[20] = data->f + data->i;
  data->fa[21] = data->f - data->i;
  data->fa[22] = data->f * data->i;
  data->fa[23] = data->f / data->i;
  data->fa[24] = (unsigned)data->f << data->i;
  data->fa[25] = (unsigned)data->f >> data->i;

  data->fa[26] = __half2float(data->half);
}

extern "C" void test_branch_simplePHI(GLOBAL DataStruct* data)
{
  float tmp;
  if(data->f<0)
    tmp = -data->f*2;
  else
    tmp = data->f*2;
  data->fa[0] = tmp;
}


extern "C" void test_alloca(GLOBAL DataStruct* data)
{
  DataStruct data_local;
  int i = data->i;
  data_local.s.s.f = data->f;
  data_local.s.sa[2].f = data->f*2;
  data_local.s.sa[i].f = data->f*3;

  data->fa[0] = data_local.s.s.f;
  data->fa[1] = data_local.s.sa[2].f;
  data->fa[2] = data_local.s.sa[i].f;
}

extern "C" void test_signedOp(GLOBAL DataStruct* data)
{
  int i = data->i;
  int j = i*2;
  data->ia[0] = j % i;
  data->ia[1] = j >> i;
  data->ia[2] = -7 / i;
  data->ia[3] = i - 5;
  data->ia[4] = data->f;
  data->ia[5] = (char)data->ia[3];
  data->fa[0] = data->ia[1];
}
```

Bibliography

[All83] Allen J.R. and Kennedy K. and Portfield C. and Warren J. Conversion of control dependence to data dependence. 1983.

[AMD09] AMD Staff. OpenCL and the ATI Stream SDK v2.0. 2009. `http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx`.

[Asa06] Asanovic K. and Bodik R. and Catanzaro B. and Gebis J. Husbands P. and Keutzer K. and Patterson D. and Plishker w. and Shalf J. and Williams S. and Yelick K. The Landscape of Parallel Computing Research: A View from Berkeley. 2006.

[Big06] Bigler J. and Stephens A. and Parker S. G. Design for Parallel Interactive Ray Tracing Systems. *IEEE Symposium on Interactive Ray Tracing*, 2006.

[Cyt91] Cytron R. and Ferrante J. and Rosen B. K. and Wegman M. N. and Zadek F. K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 1991.

[Dit09] Dittamo C. and Cisternino A. Filling the gap between GPGPUs and Virtual Machine Computational Models. 2009. `http://4centauri.codeplex.com/`.

[Fra10] Franke B. Analyzing CUDA's Compiler through the Visualization of Decoded GPU Binaries. *International Symposium on Code Generation and Optimization (CGO)*, 2010.

[Gam94] Gamma E. and Helm R. and Johnson R. and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. 1994.

[Gat09] Gat N. and Raman A. Ogre SDK Deferred Shading Demo. 2009. `http://www.ogre3d.org/tikiwiki/Deferred+Shading`.

[Geo08] Georgiev I. and Slusallek P. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. *In IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, 2008.

[Gre06] Gregory J. Pro OGRE 3D Programming. 2006.

[Gro09] Grover V. and Kerr A. and Lee S. PLANG: PTX Frontend for LLVM. *LLVM Developers' Meeting*, 2009.

[IEE08] IEEE. IEEE Standard for Floating-Point Arithmetic (IEEE 754). 2008. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4610935`.

[Kar10] Karrenberg R. and Rubinstein D. and Slusallek P. and Hack S. Anysl: Efficient and portable shading for ray tracing. *High Performance Graphics*, 2010.

[Ker09] Kerr A. and Diamos G. and Yalamanchili S. A Binary Translator Frame-work for PTX. 2009. `http://code.google.com/p/gpuocelot`.

[Khr09] Khronos OpenCL Working Group. The OpenCL Specification 1.0. 2009. `http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification/`.

[Kow08] Kowaliski C. GPU sales strong as AMD gains market share. 2008. `http://techreport.com/discussions.x/15778`.

[Lat04] Lattner C. and Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[Lat10] Lattner C. and Haberman J. and Housel P. S. LLVM Language Reference Manual. 2010. `http://llvm.org/docs/LangRef.html`.

[LLV10] LLVM Developers. Clang vs Other Open Source Compilers. 2010. `http://clang.llvm.org/comparison.html`.

[NVI09] NVIDIA. Cuda Reference Manual version 2.3. 2009. `http://www.nvidia.com`.

[NVI10a] NVIDIA. CudaProgramming Guide. 2010. `http://www.nvidia.com`.

[NVI10b] NVIDIA. Parallel Thread Execution, ISA Version 1.4. 2010.

[NVI10c] NVIDIA. Parallel Thread Execution, ISA Version 2.1. 2010. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[Owe07] Owens J. D. and Luebke D. and Govindaraju N. and Harris M. and KrÃijger J. and Lefohn A. E. and Purcell T. J. . A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 2007.

[Par91] Park J. and Schlansker M. On Predicated Execution. 1991.

[Par10] Parker S. and Bigler J. and Dietrich A. and Friedrich H. and Hoberock J. and Luebke D. and McAllister D. and McGuire M. and Morley K. and Robison A. and Stich M. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[Per02] Perlin K. Improving Noise. *Computer Graphics Forum*, 2002.

[Pet03] Pettersson J. and Wainwright I. Radar Signal Processing with Graphics Processors (GPUs) . *Master Thesis, Page 102*, 2003.

[Pha10] Pharr M. and Humphreys G. Physically Based Rendering: From Theory To Implementation, 2nd Edition. *Morgan Kaufmann*, 2010.

[PIX98] PIXAR. The RenderMan Interface. 1998.

[Pop07] Popov S. and Günther J. and Seidel H. and Slusallek P. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 2007.

[Ryo08] Ryoo S. and Rodrigues C. and Baghsorkhi I. and Stone S. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press*, page 73âĂŞ82, 2008.

[Sre99] Sreedhar V. C. and Ju R. D. and Gillies D. M. and Santhanam V. Translating Out of Static Single Assignment Form. *In Static Analysis Symposium, Italy*, pages 194–210, 1999.

[Wil94] Wilson G. V. The History of the Development of Parallel Computing. 1994. `http://ei.cs.vt.edu/~history/Parallel.html`.

[Yan08] Yang Z. and Zhu Y. and Pu Y. Parallel image processing based on cuda. *CSSE*, pages 198–201, 2008.