LLVM to PTX Backend

Program Analysis and Transformation seminar

June, 2011

University of Applied Sciences Rapperswil (HSR), Switzerland

Marco Keller

ABSTRACT

The low-level virtual machine (LLVM) compiler infrastructure is a mature and stable framework to implement optimization and compiler passes. H. Rhodin presented an LLVM backend to generate Parallel Thread Execution (PTX) instructions from LLVM bitcode. PTX is used as intermediate representation for parallel programming.

This paper discusses Rhodin's PTX generator. Due to the similarity between LLVM bitcode and PTX instructions, a straightforward transformation can be done. The requirements of parallel computing show the GPU-depending disparities between LLVM bitcode and PTX instructions. The design of the generator, the disparities and an approach to solve them is showed.

INTRODUCTION

The low-level virtual machine (LLVM) is a compiler infrastructure that bases on a virtual machine (VM). The virtual Processor handles code in an intermediate representation (IR) of its instruction set architecture (ISA). This allows language and hardware independence and improved optimizations.

The Parallel Thread Execution (PTX) is a low-level VM. It provides an ISA for general purpose parallel programming. The representation in its intermediate language is used to abstract from a GPU device.

GPUs are many-core processors that are used to parallel calculate graphics data. The recent push of parallel computing, caused by the performance gain slowdown of CPUs, has led the manufacturers to extend the GPU architecture to general purpose calculations. This is also called General Purpose Computation on Graphics Processors (GPGPU). The GPU supports the CPU as data-parallel co-processor.

The NVIDIA Compiler Collection (NVCC) uses two steps to translate high-level code to GPU assembly code. The first step translates high-level code to PTX code. In the second step, PTX code is translated to GPU assembly code**.** The compiler collection can be freely used, but it is source closed. Custom optimizations and extensions cannot be done because no information about design or optimization passes is available. An open source replacement for the NVCC is asked by the community. The open source LLVM yields a mature compiler infrastructure for

custom optimization passes and custom extensions. The LLVM bitcode is similar to the targeted PTX code, which simplifies the implementation of a LLVM to PTX generator.

H. Rhodin developed an open source LLVM-to-PTX code generator, documented in his bachelor thesis [Rho10]. The code generator is a LLVM compiler backend. With this backend a transformation from LLVM bitcode to PTX code is done. The generator implements all important GPU-specific features. It is designed to be easily extended with remaining features. A comparison between NVCC generated code and the code generated by the PTX code generator brought similar results. Optimization passes are not done by the PTX generator.

This paper gives an overview of H. Rhodin's PTX code generator. The design of the generator, the disparities between LLVM and PTX representations and a way to clear the disparities is described. Finally further work is discussed.

LLVM COMPILER

LLVM has been developed to advance dynamic compilation and optimizations during all stages of software lifetime. [Lat04] gives an introduction of the *Compilation Framework of Lifelong Program Analysis & Transformation.* LLVM provides custom optimization passes, a customizable compilation pipeline and a structure to extend an implementation with new passes.

The LLVM compiler consists of a frontend and a backend. The frontend transfers source code to the LLVM IR (LLVM bitcode). A backend is used to further transfer the LLVM bitcode to another programming language, to assembler code or to machine code. The discussed code generator is built as an LLVM backend to transfer LLVM bitcode to PTX code.

LLVM BITCODE

The LLVM bitcode is a low-level instruction set. Its advantage is that it yields high-level information for analysis of types, control- and data flow. It is comparable to an assembler language, but yields more information. It needs to be general and extensible to connect many front- and backends. It has a hierarchically composited structure. The structure can be described as follow:

- A program consists of modules and global fields.
- A module contains different functions.
- A function consists of basic blocks.
- Basic blocks contain single instructions. Basic blocks are e.g. *if-than-else* blocks or *loop-blocks*.

The advantage to this hierarchical composition is its portability and its understandable structure.

The use of the bitcode allows building frontend and backend independently. Optimizations can be done on the bitcode representation. This can be done in a more general way because library code can be considered too.

A special property of the LLVM bitcode representation is the static single assignment (SSA) form [Zad09]. Using the SSA form, a variable is only assigned once. This is done to simplify optimization passes. As a consequence of SSA, Phi-functions need to be introduced. Phi-functions

are inserted at the merging point of branches. They determine the value of variables if they depend on a previous branch.

PTX

NVIDIA describes PTX in different versions. The version 2.2 is described in [PTX22]. PTX was originally designed for NVIDIA GPUs but is now used as standard format for parallel computing. Since the NVIDIA GPU is widely used, PTX has a change to become a common intermediate language.

The instruction set of PTX is extended by *GPU specific instructions,* used for efficient parallel computing. These are instructions like the *thread synchronization instructions*, the support of *different memory spaces*, or specific *hardware implemented functions*.

There are similarities between the PTX intermediate language and the LLVM bitcode. Both are low-level VMs with their own instruction sets. This alleviates the transformation from LLVM to PTX in many cases. This similarity is an argument for choosing LLVM as source for the PTX generator.

The two languages differ in their form. LLVM uses the SSA form which is not used in PTX. PTX uses *GPU specific instructions* which cannot be directly mapped to LLVM instructions. The disparities between the two languages are discussed in more detail in the chapter PTX Code generation.

Programming Model

The PTX programming model has an *explicit parallel form*. Using this form, a program has to describe the execution of a thread. PTX uses a hierarchical structure to abstract from the GPU hardware. This hierarchical structure explicitly describes the parallel execution on the hardware. This structure is described in more detail in the chapter GPU Architecture.

CUDA / OpenCL

PTX is used by NVIDIA's Compute Unified Device Architecture (CUDA). CUDA is a proprietary framework to develop GPGPU software. Details about the CUDA implementation are described in the NVIDIA CUDA C Programming Guide [Cud11]. CUDA C is a language that is similar to C, but with GPU-specific extensions.

An open standard for GPGPU-software is the Open Computing Language (OpenCL) [Ops08]. Some manufacturers support the open standard, but still provide source closed compilers. PTX is used by CUDA or OpenCL compilers as common assembly language.

The Compilation Pipeline

The compilation pipeline in Figure 1 shows how the NVCC is used to compile CUDA and OpenCL code. These high-level source codes are translated to PTX and NVIDIA GPU machine code. The image is extracted from [Rho10].

The AMD Stream SDK implementation [Amd09] bases on OpenCL. It uses the ATI OpenCL compiler to translate OpenCL code to the ATI Compute Abstraction Layer (CAL) Intermediate Language (IL). CAL is a device-driver library on a lower level than an OpenCL implementation,

but still includes high-level control-flow structures. The CAL runtime is used to translate CAL code to ATI GPU machine code.

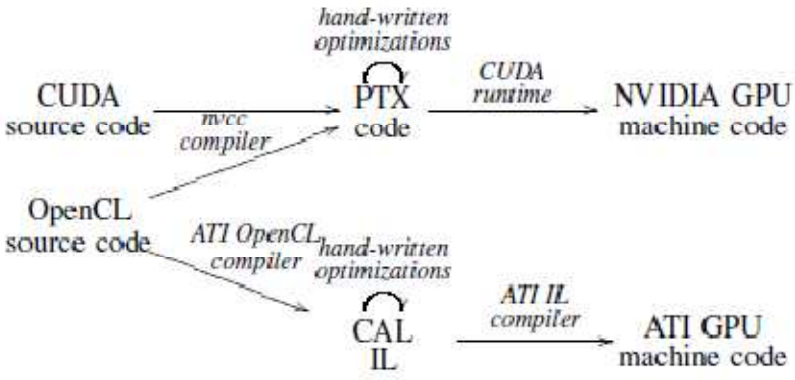

Figure 1: Compilation pipelines from CUDA/OpenCL code to NVIDIA/ATI GPU, extracted from [Rho10]

A comparison between PTX, OpenCL and CUDA is showed in Table 1.

PTX	OpenCL/CAL	NVIDIA CUDA
Low-level	High-level, based on extended C++	High-level, based on extended C++
Limited to NVIDIA GPUs	Target independent. Can be translated to NVIDIA, ATI or other GPUs.	Limited to NVIDIA GPUs
The NVCC is used to map to PTX and NVIDIA GPU machine code.	The ATI OpenCL compiler is used to map to CAL IL and to ATI GPU machine code. The NVCC is used to map to PTX.	The NVCC is used to map to PTX and NVIDIA GPU machine code.
Optimizations on low-level to low-level transformation	Optimizations on high-level to low-level transformation.	Optimizations on high-level to low-level transformation.
Fast code generation from LLVM to PTX due to their similarity. Both are low-level representations.	Complicated code generation from LLVM to OpenCL code or CAL IR due to a low-level to high-level transformation.	Complicated code generation from LLVM to CUDA C code due to a low-level to high-level transformation.

Table 1: GPGPU language comparison

Both GPGPU high-level languages (CUDA, OpenCL) are based on C++ with extensions for GPU specific features .The PTX language is kept simpler, which makes it a desired target for external compiler. The translation to CUDA, OpenCL or CAL would be more complex. Information about high-level structures is not included in the low-level bitcode. For a translation to these languages a reconstruction of the high-level constructs would be needed. The translation from LLVM to PTX is simpler and is therefore done by Rhodin. But PTX is limited to the NVIDIA GPU. The wide spread of the NVIDIA GPUs is given to alleviate this disadvantage.

Related Work

Ocelot, PLANG and the OpenCL implementation [Opi09] use a LLVM frontend to translate PTX code to LLVM.

Ocelot [Dia10] is a compilation framework that uses LLVM as internal representation. Its aim is to translate PTX code to different many-core processors. This makes PTX portable. The NVCC only translates PTX to NVIDIA GPU assembly code.

WHY IS A TRANSLATION FROM PTX TO LLVM CHOSEN?

LLVM is chosen due to its *similarity* to the PTX ISA, the existing *x86 backend generators* and the *optimization possibilities*. LLVM supports Ocelot in creating a framework that can be used to create compilers for *PTX to an arbitrary target representation*. PTX code can be translated to GPU assembly code of other manufacturers than NVIDIA. An emulator is further integrated into Ocelot to emulate PTX code at runtime with its just-in-time compiler. The emulation of PTX code on CPUs helps debugging during parallel programming. With LLVM, the code can be *optimized at runtime*.

PLANG [Pla09] is another LLVM frontend similar to Ocelot. It also translates PTX code to LLVM. NVIDIA developed it to execute PTX kernels on multicore CPUs. Its purpose is to debug PTX code one the CPU.

The OpenCL implementation [Opi09] uses LLVM as intermediate representation. Clang is used to translate from an OpenCL-extended C++ implementation to LLVM bitcode. The LLVM optimization opportunities, the features of the bitcode libraries and the LLVM inliner are arguments for LLVM. The inliner is important, because some target devices do not support function calls.

GPU ARCHITECTURE

HARDWARE

A GPU device consists of a set of streaming multiprocessors (SM) and dynamic random access memory (DRAM). Figure 2 shows the hierarchical structure of the GPU. The image is extracted from the NVIDIA PTX ISA 2.2 [PTX22].

A SM contains several scalar processor (SP) cores, an instruction unit, shared memory and registers. Because registers and shared memory are on-chip, they benefit from low-latency.

DATA PARALLEL APPROACH

All SP cores are executed in parallel. The instruction unit is used to schedule threads to the SP cores. A program counter (PC) per core determines the control-flow path. All cores execute at the same position in the code, but every core calculates different data.

This machine organization is called single instruction multiple data (SIMD). [FLY72] describes it as *an organization that applies an instruction over a vector of operands*. The *array processor* is described as a SIMD type, characterized by a control unit and processing elements. As in GPU architecture, the processing elements are independent by using their own registers but receive instructions from the control unit. [FLY72] describes several problems caused by the SIMD organization. One of these is called: *Degradation due to branching*. It describes that due to a branch, the processing elements may be in different states, but the control unit can only control one state.

Due to this problem, the NVIDIA GPU architecture is expanded to the single instruction multiple thread (SIMT) architecture. The difference of this organization is that each thread can be independent branched. Thus, it is possible to have threads at different control-flow paths. This makes GPU programming more flexible but introduces the disadvantage of the serial execution of all threads on different control-flows. This situation is called *divergent*. Due to such serial executions, performance is decreased.

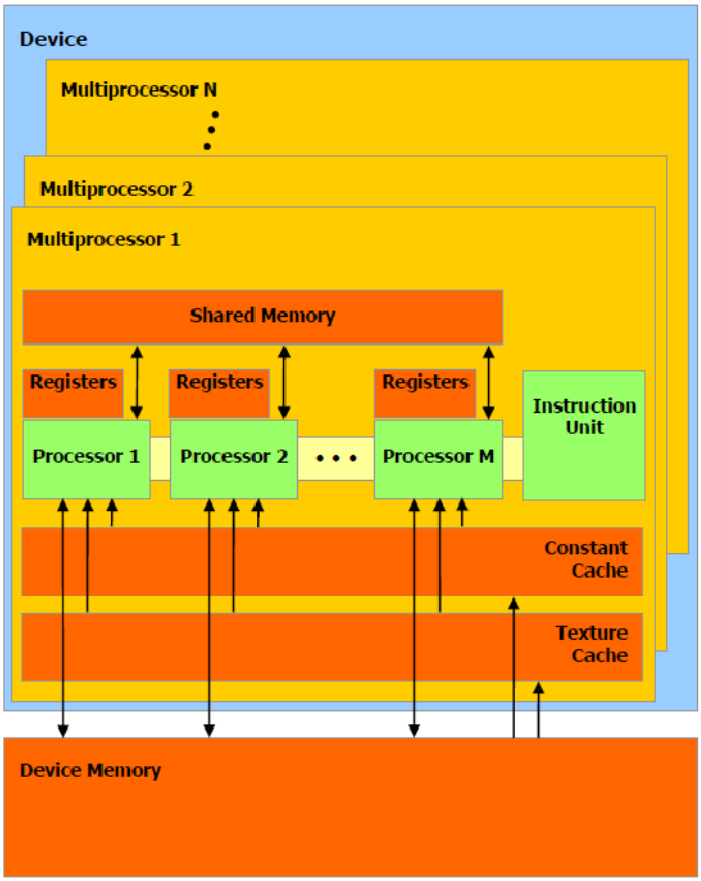

Figure 2: The hierarchical structure a GPU, extracted from the NVIDIA PTX ISA 2.2

PTX Implementation

PTX abstracts the hardware in a grid of threads. These threads are grouped to several cooperative thread arrays (CTA). PTX uses CTAs as an abstraction of the SP cores. CUDA implements CTAs as thread blocks.

The threads in a CTA are grouped into warps. A warp consists of 32 threads. At runtime, a SM gets several CTAs for execution. The instruction unit is used to select a warp out of all CTAs to be executed. The threads of this executing warp are mapped to the SP cores.

Memory Spaces

An automatic variable is allocated and freed if the program flow enters and leaves its scope. On the GPU these variables are allocated on the registers of the SP cores. If the variables do not fit into the registers or arrays with variable indexes are used, the variables are allocated in the local memory space which is in the DRAM. This memory space is used as some kind of stack with the constraint that it does not support recursive calls. Other memory spaces on the DRAM are the global memory, the constant memory and the texture memory. The global memory is used for

the data transfer to the CPU and to share data among threads of all CTAs. Constant and texture memories are used for read-only parameters.

All threads within a CTA have access to the same shared memory scope. Thread communication and caching of global memory data with the help of shared memory is limited to a CTA. Access to shared memory scopes of other CTAs is not possible.

THE PTX CODE GENERATOR

The purpose of the PTX code generator is to provide a LLVM backend. This backend translates LLVM bitcode to PTX code. The PTX code generator is part of a compilation pipeline that can be established to create GPU assembly code. Figure 3 shows this compilation pipeline. The image is extracted from [Rho10].

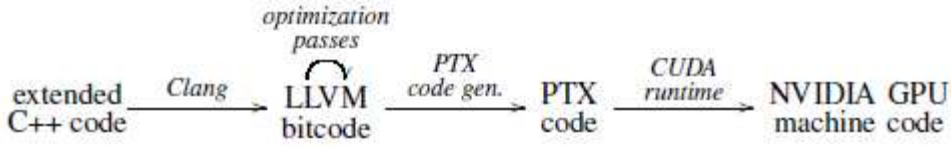

Figure 3: The compilation pipeline from extended C++ code to NVIDIA GPU machine code, extracted from [Rho10]

The NVCC compiler translates a CUDA program in two steps to GPU assembly code. The LLVM frontend Clang [Dev07] and the PTX code generator replace the first step of the NVCC.

Clang is used to translate extended C++ code to LLVM bitcode. Other frontend compilers may be used as long as they support the extensions which are needed for the GPU. All extensions of the frontend need to be handled by the LLVM bitcode to support GPU-functionality. LLVM needs to be extended by intrinsic functions and address space annotations to support these features. LLVM itself has no representations that support these features. A convention is used to map frontend extensions to LLVM extensions.

Optimizations that are done by the NVCC during the first step need to be replaced by LLVM optimization passes. These are common and GPU-specific optimizations. NVIDIA does not publish information about these optimization passes, thus, custom optimizations cannot be applied. The replacement of the first step of the NVCC by the LLVM compiler and the PTX generator offers the possibility for custom optimization passes. Such optimizations are separated from the PTX code generator. Rhodin leaves the implementation of optimization passes to further work.

The PTX generator is used for the translation to the PTX representation. The translation must preserve the semantic of a program. All LLVM instructions need to be mapped to appropriate PTX instructions. A direct mapping of LLVM programs that contain recursive or indirect calls is not possible. NVIDIA GPUs do not support such calls. This limits the set of LLVM bitcode programs that can be handled by the generator.

The NVCC is used in the last step to translate PTX code to NVIDIA GPU assembly code. This step contains late optimizations.

DESIGN

Rhodin implemented the PTX code generator with the *LLVM custom passes approach* that contains the following steps:

1. Pre code passes performing transformations at the LLVM-level to create a form that is representable in PTX.
2. The code generation pass translates the LLVM instructions to PTX instructions.

Using this approach, the generator is a sequence of LLVM passes. The pre code generation passes prepare the generator input. The code generation pass generates the PTX code. The output of this pass is stored in a string stream. It can be further used by a file writer or another compilation entity.

PTX CODE GENERATION

A lot of instruction translations from the LLVM bitcode to the PTX code can be done in an elementary mapping. Most instruction mappings are simple. For the remaining instructions, a forming or a replacement needs to be done. These kinds of translations originate from the disparities between LLVM bitcode and PTX code. Rhodin divides these disparities in two categories, A and B, explained in the next section.

DISPARITIES A

The operations originating from the disparities A are handled in the pre-code-generation passes. Most of the operations are format transformations, which need to be done to fulfill the needs of the generator. Table 2 gives an overview of the disparities A.

Title	Description
(A1) Address calculation	**Different address calculation handling of structs and arrays.** LLVM uses the *get element pointer* (GEP) to hide address calculations. PTX has no GEP. Address calculations are hard-coded.
(A2) Global variables	**Different global variable pointer types.** A global variable pointer in LLVM is constant. PTX handles these pointers in registers.
(A3) Utility functions	**Different handling of math-functions.** LLVM uses intrinsic representations for functions like exp or pow. Some of them are not directly supported by PTX. They need to be approximated with other functions.
(A4) Vector types	**Different restrictions on vector type support.** LLVM has no restrictions. PTX only supports specific operations on vector types and the vector widths are limited to 2 or 4 elements.
(A5) Predicate	**Different predicate representation.** LLVM has no predicate representation. Predicates are implemented with conditional branches. Each PTX instruction can be marked with a predicate to exclude the instruction on execution.

Table 2: Disparities A

PRE CODE GENERATION PASSES

The transformation done in these passes only change a LLVM representation in a more suitable form. This preparation alleviates the implementation of the PTX code generation pass.

ADDRESS SPACE PASS:

This pass is a workaround. The Clang frontend sometimes has problems with the LLVM address space annotation. In such cases a naming convention is used to identify the address space dedicated to a variable or pointer. This pass searches for key words in pointer or variable names. If a key word is found on a default address space (0) member, an appropriate LLVM address space annotation is inserted to change the address space type.

SPECIAL INSTRUCTION PASS:

This pass replaces LLVM features that have no PTX counterpart. These features are replaced by LLVM functions that can be directly mapped to PTX instructions. Such LLVM features are mathematical functions **(A3)** and GEP instructions **(A1)**.

Transformation of the tan(x) function

This example shows the replacement of the tan(x) function by a sin(x) and a cos(x) function. The transformation within LLVM is required because PTX has no tan(x) function, thus a direct mapping is not possible. This pre code generation pass alleviates the implementation of the PTX code generator. The tan(x) function can be expressed as: $\tan(x) = \frac{\sin(x)}{\cos(x)}$.

Listing 1 is extracted from the *BasicBlock* pass *PTXBackendInsertSpecialInstructions*, which is part of the pre code generation pass in [PTP10]. Listing 1 shows a part of the method *replaceSpecialFunctionsWithPTXInstr*.

```
replaceSpecialFunctionsWithPTXInstr(CallInst* callI)
Function* F = callI->getCalledFunction();
if(F->getName() == "tanf"){
      CallInst* callSin = callI;
      callSin->setOperand(0, sinFun);

      CallInst* callCos = CallInst::Create(cosFun, callI->getOperand(1));
      callCos->insertAfter(callSin);

      BinaryOperator* DivInst = BinaryOperator::Create(Instruction::FDiv,
            callSin, callCos);
      DivInst->insertAfter(callCos);

      callI->replaceAllUsesWith(DivInst);
      DivInst->setOperand(0,callSin);
}
```

Listing 1: Replacement of the tan(x) function

First, the tan(x) function call is changed to a predefined sin(x) function. Then a cos(x) function call and a division operator are inserted. Finally, all users of the tan(x) function call need to replace it by the division operator. The sin(x) function call of the division operator needs to be reset because the call to *replaceAllUsesWith* changed it.

After this LLVM-level transformation, the code generator needs no handling for a tan(x) function. The simple translations of the sin(x), cos(x) and division operation can be used. Other functions like pow(a, x), exp(x), log(x) and cot(x) are handled in the same pre code generator pass.

Global variable handling during GEP simplification

This example considers a special handling of global variables during GEP instruction simplification. The base addresses of global variables in LLVM are constant. LLVM uses

optimization passes to connect constant expressions. If there is a constant address and an offset like *address+4*, the optimization pass replaces these with a constant expression. PTX does not consider the base addresses of global variables as constant **(A2)**. An offset like *address+4* is therefore handled by an *Add-operation*, as explained in [MRh11].

To prevent a special handling of global variables in the code generator pass, a wrapper function call is inserted around each LLVM global base address. The wrapper prevents the LLVM optimization pass to build constant expressions, because a variable return value of the wrapper function is assumed. Listing 2 shows the insertion of the wrapper function. The code is extracted from the *simplifyGEPInstructions* method within the *BasicBlock* pass *PTXBackendInsertSpecialInstructions* in [PTP10].

```
simplifyGEPInstructions(GetElementPtrInst* GEPInst)
if(isa<GlobalVariable>(parentPointer)){
      Function* constWrapper = Function::Create(
            FunctionType::get(parentPointer->getType(), true),
            GlobalValue::ExternalLinkage, Twine(CONSTWRAPPERNAME));

      std::vector<Value*> params;
      params.push_back(parentPointer);

      CallInst* wrapperCall = CallInst::Create(constWrapper,
            params.begin(), params.end(), "", GEPInst);
      parentPointer = wrapperCall;
}
```

Listing 2: Insert wrapper function

If the *parentPointer* is a global variable, a wrapper function is created. Then a call instruction to the wrapper is created, using the *parentPointer* as argument. Finally, the call instruction to the wrapper function is assigned to the *parentPointer*, which is used in the subsequent GEP simplification.

POLISH PASS:

This pass removes helper constructs like the previously discussed wrapper function.

DISPARITIES B

The operations originating from the disparities B are handled in the final code generation pass. Table 3 gives an overview of the disparities B.

Title	Description
(B1) SSA property	**Different intermediate representation form**. LLVM uses the SSA form and Phi-instructions to determine values at control-flow nodes. PTX has no form restrictions and no Phi-instructions.
(B2) Integers	**Different behavior on signed and unsigned integers**. LLVM has no general distinguishing between signed and unsigned types. Some instructions are provided in two versions. PTX has signed and unsigned integers. The behavior of some instructions depends on the type.
(B3) Arbitrary bit sizes	**Different supported type sizes.** LLVM allows arbitrary bit sizes of primitive types. PTX restricts the size of primitive types to 8, 16, 32 and 64 bit.

(B4) Memory spaces	**Different memory spaces and qualifier**. LLVM has an address space qualifier but without fix semantic. PTX provides qualifier to access the different memory spaces of the NVIDIA GPU.
(B5) Kernel and device functions	**GPU-specific kernel and device functions** LLVM does not support these categories. PTX distinguished between kernel and device functions.
(B6) GPU-specific instructions	**Instructions that are hardware implemented on the GPU.** These are unknown to the LLVM. Intrinsic functions are used in a convention to map such functions to PTX instructions.

Table 3: Disparities B

GPU LANGUAGE EXTENSION CONVENTION

LLVM needs a language extension to cover the disparities B. Rhodin defines such a convention to extend the LLVM functionality. It ensures the commonly usage of GPU-specific extensions that need be recognized and mapped by the PTX code generator. The convention is used as an interface for frontends. Only code and frontends that support the convention (and therefore the GPU-specific features) can be translated to PTX code. Code without these extensions cannot be translated. Table 4 gives an overview of the disparities B handling.

(B1) The **SSA property** of LLVM code is not handled by the convention. It is handled in the code generation pass. The additional registers introduced by the SSA are removed by the optimization pass of the NVCC in the last step of the compilation pipeline.

(B2) The LLVM **integer** is mapped to an unsigned PTX integer. If LLVM uses a signed instruction, the code generator changes to a signed PTX integer.

(B3) The PTX code generator extends not supported data **type sizes** to the next supported size.

(B4) The mapping of the LLVM **address space** attributes to the PTX memory spaces is handled by the convention.

LLVM address space attribute	PTX memory space
0	local memory
1	shared memory
2	global memory
3	constant memory
4	texture memory

A frontend has to support annotated code of the form: __address_space(1..4).
If no annotation is done, local memory is used.

The allocation of LLVM stack memory is therefore translated to local memory. GPUs have no stack. The local memory is used as "stack" with depth 1.

(B5) The PTX code generator can deduct the **type of the function**.

Device function	A function that is called by another function.
Kernel function	A function that is not called by another function and has void as return type.

Undecidable functions are mapped to device functions.

(B6) **GPU-specific functions**, not supported by LLVM, are represented by intrinsic functions.

Table 4: Disparities B handling

Consequences, resulting from the convention and the disparities **(B4)** and **(B5)** are:

- The LLVM address space annotations need to be used after the convention. Annotations in high-level source code are needed.
- GPU-specific functions are represented by intrinsic functions. They are defined in a header file. This file needs to be included in the C++ code.

Extended C++ code is syntactically equivalent to CUDA code. This may be of interest if a CUDA port to extended C++ is done.

CODE GENERATION PASS

This is the pass which generates the PTX code out of the LLVM bitcode. All LLVM bitcode instructions are mapped to corresponding PTX instructions.

LLVM is hierarchically built where every level has its appropriate LLVM instructions. To translate LLVM bitcode, every level of this hierarchy has to be inspected. The corresponding instructions need to be translated and PTX code is printed. If a level contains sub-levels, these have to be visited accordingly. Table 5 shows the LLVM code-representation hierarchy and its responsibilities in the PTX code generation pass.

Level	Responsibility
Module	• Prints global variables and function declarations.
Function	• Prints the function signature for every declared function. The **function type** qualifier is resolved after the convention **(B5)**. • Prints register declarations at the top of the function body. • SSA related instructions in the **SSA form** are transformed **(B1)**. • The arguments for kernel functions are stored in the shared memory space. → Load instructions are printed to load these arguments to the registers.
Block	• Prints a block mark. This mark is used by branch instructions to jump into the block.
Instruction	• A LLVM instruction is transformed to a PTX instruction by an appropriate visitor method. • Not supported LLVM **data type sizes** are extended to the next supported sizes **(B3)**. The extension of the data type size may have side effects in programs depending on overflows of the data type size.

Table 5: LLVM code representation hierarchy with responsibilities

A depth-first-traversal is used to visit all hierarchy level nodes. Every hierarchy level node has a handler implemented for the transformation. These handlers do the actual transformation. Table 6 shows some of the more complex instruction handlers.

Handler	Special considerations
Binary operations	This handler transforms binary operations. PTX requires special handling with operations that require a rounding mode qualifier. If PTX does not support bit-sizes of LLVM types, these bit-sizes are promoted to the next larger PTX type **(B3).**
Branch instructions	This handler prints the branch instructions by using PTX predication **(A5)**. First, a predicated branch condition is printed, followed by the consequence block. Then a predicated, negated branch condition is printed, followed by the alternative block. Phi-instructions are further handled **(B1)** and predicates are used for conditional expressions.
Comparison instructions	This handler translates integer and float comparisons.
Call instructions	A special handling is used to translate GPU-specific features that are represented by intrinsic functions **(B6)**. This handler prints included, hard-coded implementations for the intrinsic functions.
Alloca instructions	This handler translates the alloca instructions of arrays and structs. According to the convention, these elements are allocated in the local address space **(B4)**.
Load/Store instructions	The instructions translated by these handlers are used to load data from memory cells into registers or store register values to memory cells. The according PTX address space is determined by the LLVM address space attribute at the memory cell pointer. The address space mapping is done according to the convention **(B4)**.

Table 6: Instruction handler

FURTHER WORK

IS IT POSSIBLE TO COMBINE OCELOT AND RHODIN'S PTX GENERATOR?

Further work is required even though both use LLVM as common origin or target language.

- Ocelot: LLVM frontend, PTX -> LLVM
- Rhodin's PTX generator: LLVM backend, LLVM -> PTX

Rhodin's PTX generator requires the convention described in the section GPU Language extension convention. Ocelot does not fulfill this convention. Thus, further work would be required to fill such a convention-gap.

When translating from PTX to LLVM with Ocelot, information about PTX concepts are lost. One of these concepts is the thread hierarchy and the information about the exact amount of used threads. Ocelot uses the *thread fusion* technique to represent thread bulks with loops on many-core architectures. This changes the control-flow in the LLVM representation. In a combination with Rhodin's PTX generator, the data parallel control-flow would be needed to be reconstructed.

Ocelot translates GPU-specific functions to function calls into an emulation library. Rhodin's PTX generator uses intrinsic functions. A combination would need an LLVM custom pass that translates the library calls to the intrinsics.

Ocelot does not provide memory space annotations or qualifiers during PTX to LLVM translation. The convention of Rhodin's PTX generator describes the usage of LLVM annotations for memory spaces. An LLVM custom pass would be needed to reconstruct the memory space information.

The discussed combination-case shows that even with LLVM as a common language, work is required to reconstruct parallel depending information. LLVM is a good basis for data parallel compilers, but data parallel depending elements like defined address space annotations or elements to transfer thread configuration information could be added.

CONCLUSION

The PTX generator presented by H. Rhodin is discussed in this paper. Since the NVCC is source closed, an open source PTX compiler is asked to give possibilities for custom optimizations and custom extensions.

The PTX generator is the first LLVM backend to generate PTX code. In combination with Clang, it can be used as an alternative to the first part of the NVCC. Most PTX features are implemented by the PTX generator and the evaluation brought similar results in comparison to the NVCC.

LLVM has been chosen as intermediate language on the compiler pipeline from extended C++ to PTX code. LLVM has the advantage that it has a stable and mature codebase and it is open for custom optimizations and extensions. Further advantages of LLVM for the PTX generator is the similarity of LLVM bitcode and PTX instructions and existing LLVM frontends.

This paper regards the design of the PTX generator. The disparities between LLVM and PTX are discussed. It is showed how the PTX generator clears these disparities. Some of them are solved during the (pre) code generation passes, others need a language extension.

A convention extends the functionality of LLVM to handle memory spaces and GPU-specific functions. This shows a disadvantage of LLVM as intermediate language. Although it is possible to transfer data parallel specific information by the extensibility of LLVM, commonly defined elements for a data parallel usage are missing.

REFERENCES

[Rho10] Rhodin H, *A PTX Code Generator for LLVM*. Bachelor Thesis, 2010. http://switch.dl.sourceforge.net/project/llvmptxbackend/Rhodin_PTXBachelorThesis.pdf

[Dia10] Diamos G, Kerr A, Sudhakar Y and Clark N, *Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogenous Systems, 2010*

[PTX22] NVIDIA, *PTX: Parallel Thread Execution ISA Version 2.2*, 2010

[Pla09] NVIDIA, *PLANG: Translating NVIDIA PTX language to LLVM IR Machine* http://llvm.org/devmtg/2009-10/Grover_PLANG.pdf

[Ops08] K.O.W. Group, *The OpenCL Specification*, 2008, http://www.khronos.org/opencl/

[Opi09] OpenCL, http://llvm.org/devmtg/2009-10/OpenCLWithLLVM.pdf

[Cud11] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, http://www.nvidia.com/object/cuda_home_new.html

[Lat04] C. Lattner, V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, 2004, http://llvm.org/

[Zad09] Zadeck, F. Kenneth, *Presentation on the History of SSA* at the SSA'09 Seminar, 2009

[Amd09] AMD Stream SDK OpenCL implementation, 2009, http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx

[Dev07] LLVM Developer's meeting 2007, clang: a C language family frontend for LLVM http://clang.llvm.org/

[MRh11] H. Rhodin, private e-mail communication, 28.04.2011

[FLY72] M. J. Flynn, Some computer organizations and their effectiveness, 1972

[PTP10] H. Rhodin, PTXPasses.cpp, part of LLVM to PTX backend, 08.08.2009