Automatic Packetization

Ralf Karrenberg

Saarland University
Saarbrücken, Germany

Master's Thesis
Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik
Master-Studiengang 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. Joachim Weickert, Universität des Saarlandes,
Saarbrücken, Germany

Eingereicht am / Thesis submitted:
July 1, 2009

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

Erklärung / Declaration:
Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und alle verwendeten Quellen angegeben habe.
I hereby declare that the work presented in this master's thesis is entirely my own and that I did not use any sources or auxiliary means other than those referenced.

Saarbrücken, July 1, 2009

Einverständniserklärung / Agreement:
Hiermit erkläre ich mich damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek der Fachrichtung Informatik aufgenommen wird.
I hereby agree on including my work into the inventory of the computer science library.

Saarbrücken, July 1, 2009

Abstract

Modern processor architectures provide the possibility to execute an instruction on multiple values at once. So-called SIMD (Single Instruction, Multiple Data) instructions work on *packets* (or *vectors*) of data instead of scalar values. They offer a significant performance boost for data-parallel algorithms that perform the same operations on large amounts of data, e.g. data encoding and decoding, image processing, or ray tracing.

However, the performance gain comes at a price: programming languages provide no elegant means to exploit SIMD instruction sets. Packet operations have to be coded by hand, which is complicated, unintuitive, and error prone.

Thus, *packetization*—the transformation of scalar code to packet form—is mostly applied automatically by local compiler optimizations (e.g. during loop vectorization) or with a lot of manual effort at performance-critical parts of a system.

This thesis describes an algorithm for *automatic packetization* that allows a programmer to write *scalar* functions but use them on *packets* of data. A compiler pass automatically transforms those functions to work on packets of the target-architecture's SIMD width. The resulting packetized function computes the same results as multiple executions of the scalar code.

The algorithm is implemented in a source-language and target-architecture independent intermediate representation (the *Low Level Virtual Machine (LLVM)*), which enables its use in many different environments.

The performance of the generated code is shown in a real-world case study in the context of real-time ray tracing: serial shader code written in C++ is automatically specialized, optimized, and packetized at runtime. The packetized shaders outperform their scalar counterparts by an average factor of 3.6 on a standard SSE architecture of SIMD width 4.

Contents

Chapter 1
Introduction

Modern hardware architectures for CPUs and GPUs focus on exploiting parallelism: multiple processors are connected on the same chip and each processor itself can work on multiple values in parallel using SIMD instructions (Single Instruction, Multiple Data).

A SIMD instruction performs an arithmetic or logic operation on multiple values in parallel, e.g. N additions or N comparisons. In general, N is flexible for types of different sizes: a SIMD register can store a fixed number of bytes that can e.g. hold N single-precision or $N/2$ double-precision floating-point values. Throughout this thesis, we will use N to refer to the number of single-precision data items a SIMD register of the target architecture can hold (in short: the *SIMD width*). On current CPUs, N is usually four (MMX, 3DNow!, SSE–SSE4, AltiVec).

These vector or *packet* instructions provide a significant performance boost to data-parallel algorithms. Examples are algorithms in image processing, data encoding and decoding, ray tracing, signal processing, or computer simulations in particle physics or climate models.

Unfortunately, programming languages do not define a suitable abstraction for these instruction sets. The user is forced to explicitly use uncomfortable, non-portable intrinsics. Some languages even do not provide this option (e.g. Java only allows scalar processing, although the virtual machine might utilize available SIMD instructions).

Parallelism & Vectorization We will shortly describe and distinguish the different notions of *parallelization* and *vectorization*. The next section defines *packetization*.

- *Parallelization* is a general term used to describe a transformation of scalar code to parallel code or to code that is executed in paral-

lel. This includes SIMD, SIMT (Single Instruction, Multiple Threads) and MIMD (Multiple Instruction, Multiple Data) computation models. Where SIMD describes the execution of one instruction with multiple values on one processor that implements special instruction sets, SIMT performs the same computation using multiple threads that work on different scalar processor cores in parallel. Modern GPU architectures mostly concentrate on this model of computation.

Parallelization also includes transformation of code to a MIMD model where entirely independent parts of a program are executed in parallel on different processors. This does not necessarily include any transformation of the scalar code itself.

- *Vectorization* describes transformations that *enable* parallelization by exposing data parallelism. This is usually accomplished by so-called *Loop Vectorization*, which comprises a number of local compiler optimizations that aim at exploiting the parallelism of loops.

  An example for such an optimization is combining data of several unrolled loop iterations to vectors. These can be exploited by vector instructions.

  This field has been widely studied and a multitude of different optimization techniques have been developed.

1.1 Automatic Packetization

Packetization (also: *Data Parallelization* or *SIMDfication*) describes the process of transforming scalar code, given by a control-flow graph (CFG) G, into a CFG G' that works on N scalar input values at once. One execution of G' is semantically equivalent to N executions of G. The performance benefit of a packetized function comes from the utilization of SIMD instructions which are able to perform a single operation on N values in parallel.

Unfortunately, implementing packet code by hand is cumbersome and error prone. Replacing the usual arithmetic operators with their intrinsic counterparts is annoying but comparatively easy. The major difficulty is modelling control flow, which can quickly become very complex:

Since we are executing N instances of the scalar code in parallel, control-flow might diverge. For example, we might execute the then-branch of an if-then-else for some input values and the else-branch for others. The packetized code has to execute both branches and merge the contents of the variables according to the branch condition. In compiler construction, this is called *predicated code*.

Similar complications occur for loops. Each loop has to be iterated until the loop-exit condition is `false` for the *last* element in the packet. This possibly involves operations that produce wrong values because the corresponding instance would have left the loop if it was executed in the scalar version. The effects of such operations have to be nullified during iteration in order to preserve correct results. Section 2.2 describes terminology for this and Section 4.3.1 details our solution.

Currently, programmers only employ packet code if an application is very performance critical and the algorithm is compute-intensive and can be expressed in straight-line code.

Compiler optimizations that try to automatically exploit parallelism in scalar programs are a topic of research since many years. However, they only target specific local constructs and do not transform entire functions or programs into packet code. Especially *loop vectorization* is a widely applied technique that transforms loops to perform several iterations in parallel.

True *automatic packetization* of a whole function has only been implemented for compilers of a few domain-specific languages (like shading languages in the field of computer graphics). These however require the programmer to work on a special language with all kinds of restrictions and limited portability (see Chapter 3).

To solve these problems, this thesis presents a source-language and target-architecture independent infrastructure for automatic packetization.

1.2 Contributions

In summary, our approach has the following advantages:

- We present a *platform-independent* compiler pass that performs automatic packetization of a *source-language independent* intermediate representation. Consequently, functions can be written in a *scalar fashion* in any language that compiles to the IR and are automatically packetized to any packet width for any SIMD architecture. Chapter 4 presents the packetization algorithm in more detail.

- The packetizer provides a simple interface: The user only has to implement a scalar function and provide a prototype for the corresponding packetized function. The implementation of this prototype is automatically generated by the packetizer.

- Classic compiler transformations work on the abstract syntax tree (AST) because it is easy to use and code generation is simple. However, the AST does not enable arbitrary optimizations, which is why we perform packetization on the control-flow graph (CFG) of a function instead. This allows us to perform aggressive optimization of the scalar source code before packetization.

- We evaluate our system in the context of real-time ray tracing. The packetizer is employed to automatically transform scalar material shaders (programs responsible for the appearance of an object) into packet code. The packetized shaders outperform their scalar counterparts by an average factor of 3.6 on a standard SSE architecture of SIMD width 4. Chapter 5 presents our results.

Outline

The thesis starts with a chapter on the required foundations and terminology, followed by an overview of the related previous work on data-parallel programming, automatic parallelization and shading languages. The main part introduces the automatic packetization algorithm and details its implementation. A case study shows the system's applicability in a real world scenario and the efficiency of the generated code. The thesis is finished by the conclusion and a brief discussion of possible future work.

Chapter 2
Foundations

In this chapter we describe the basic concepts and terminology the reader should be familiar with.

2.1 Control-Flow Graphs

The *control-flow graph (CFG)* of a function is a directed graph of basic blocks. A block may have one or two outgoing and arbitrarily many incoming edges, except for the entry block which has no predecessor. Each basic block holds a list of instructions that have to be executed in order (no branching).

We only consider *reducible* CFGs that only contain loops with a single entry edge. In this context there are additional important terms:

- The *header* of a loop is the single entry point of the loop, which has exactly two incoming edges: one from the preheader and one from the latch.
- The *preheader* of a loop is defined as the single predecessor block of the loop header that does not belong to the loop (the entry point of the loop).
- The *latch* of a loop is defined as the single predecessor block of the loop header that belongs to the loop (the last block of the loop that is executed before the next iteration starts). Note that the latch is not necessarily also the block where loop is left.

We rely on a transformation (called "loop simplification" in LLVM) that guarantees the properties of these blocks (see Section 4.1). An example for a CFG including a loop is shown in Figure 2.1.

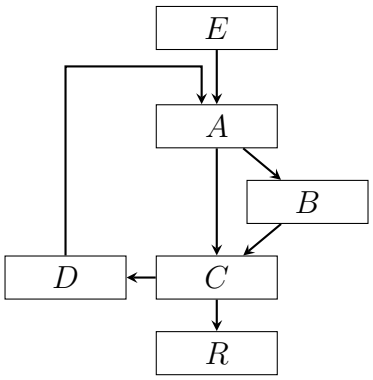

Figure 2.1: *An example CFG. The basic block E is the entry block and at the same time the preheader of the loop that consists of blocks A, B, C, andD.. Block A is the header of this loop, D is the loop latch. Block R is the final block that holds a* `return` *statement.*

SSA Form The packetizer works on an intermediate representation that uses the static single assignment (SSA) property. Every variable has only a single definition in the program text. If a source code variable had many definitions, an SSA variable is created for *each* definition of the original variable. At certain points, the so-called dominance frontiers [6], ϕ-functions are inserted to create a new uniform name for distinct definitions flowing into the block. One can think of ϕ-functions as control-flow dependent copies. Figure 2.2 gives an example for a usual and an SSA-form CFG.

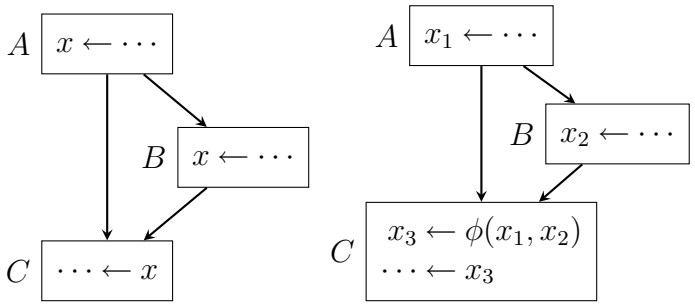

Figure 2.2: *A CFG and its SSA counterpart*

2.2 Packet Terminology

In this thesis, we will often juxtapose scalar and packet code. In order to be able to describe control-flow related transformations as exact as possible, we have to define some specific terminology.

Consider an execution of the scalar function SF (Figure 2.3): The *trace* of the function is the exact order of operations that are performed for specific input values. If SF is executed four times with different input values, it might produce four different traces, e.g. due to different branch decisions.

Now, consider an execution of the packet function PF that was generated by the packetizer with the same four input values. As we only have one trace now, it has to include all operations of the four scalar traces of SF: it represents their *union*. Figure 2.4 shows an example for such a trace of SF and PF.

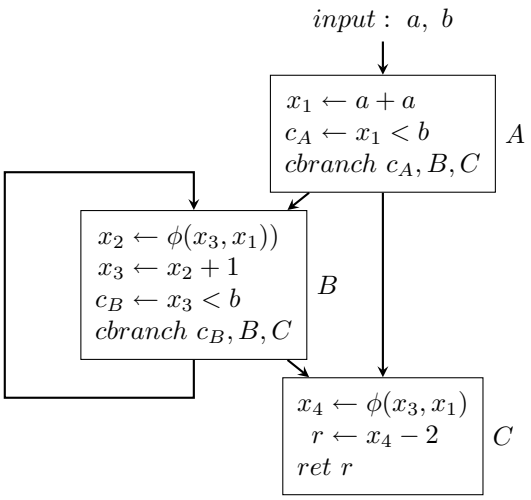

Figure 2.3: *The scalar function SF that increments x (which is $2 * input\ a$, represented by the SSA values x_1, x_2, x_3, x_4) as long as it is smaller than the input b and substracts 2 before returning the result.*

In the context of packet code, we will use the term *n-th scalar instance* or just *n-th instance* in order to refer to the trace of index n. This is especially necessary when comparing the generated packet code to the scalar code.

For example, two packets a and b of size 4 are multiplied. a holds the values $1, 3, 5, 7$ and b the values $0, 2, 4, 6$. We say that the *third instance* of the packet multiplication performs the same operation as a scalar multiplication of 5 and 4.

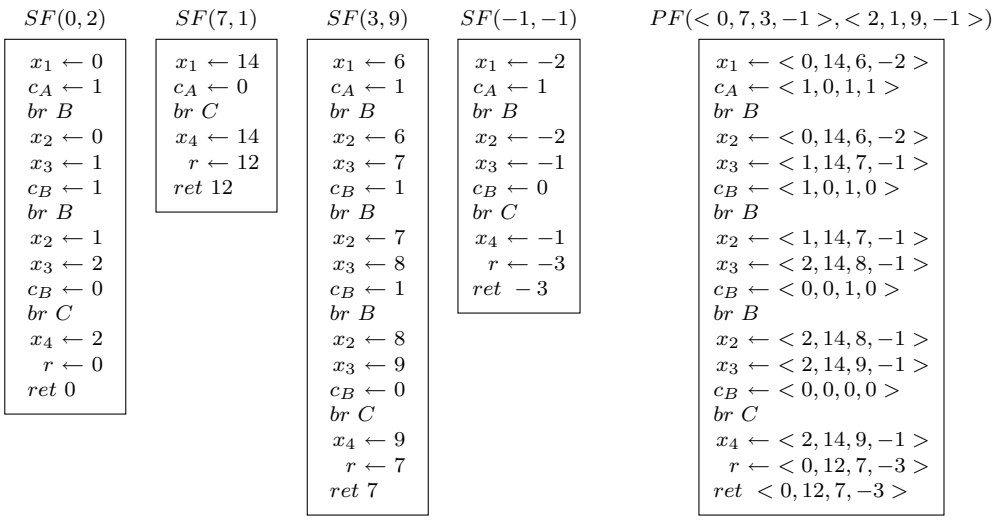

Figure 2.4: *Different traces for the scalar function SF and the corresponding trace of the packet function PF (input values in brackets). The trace of PF is the* union *of the scalar traces: it includes all their calculations and produces the same results. For the sake of simplicity,* masks *and* select *operations are omitted in the trace of PF (see Sections 4.2 and 4.3).*

If control flow of the instances diverges (e.g. during execution of a loop), we talk about *active* and *inactive instances*. This describes that specific blocks or instructions would or would not be executed by the scalar code for the corresponding input data.

For example, if a loop iterates a different number of times for different input values, some instances will become inactive. These inactive instances must not be modified until all instances have finished loop iteration. See Section 4.3.1 for more details including an explicit example.

2.3 LLVM

We use the *Low Level Virtual Machine (LLVM)* as the basis for our automatic packetizer. It provides a compiler infrastructure which made it easy to integrate our own system. The LLVM compiler framework includes:

- A language independent, typed intermediate representation in SSA
- A GCC-based C/C++ front-end
- Various backends, e.g. for X86, PowerPC, and CellSPU
- A link-time optimization framework
- A just-in-time compiler

The packetizer works entirely on LLVM's intermediate representation (IR), allowing the user to be source-language and target-architecture independent as long as LLVM provides the corresponding frontend or backend.

2.3.1 The Intermediate Representation

In Chapter 4, we will show a few examples in human-readable LLVM IR assembly that we describe in the following. LLVM's frontends compile programs to a compressed representation of this IR which is referred to as *bitcode*. The top-level construct stored in a bitcode-file is a *module*, which can contain functions and global constants and variables.

The LLVM IR is quite low-level compared to languages like C/C++ or Java: there are no such constructs as classes, if-then-else statements or explicit loops. All control-flow related constructs of a function are transformed into basic blocks connected by edges.

Each instruction corresponds to exactly one operation and represents an SSA value. Each value has a name that starts with "%" which is referenced everywhere that value is used. A typical instruction looks like this:

```
%r = add <4 x float> %a, %b     ; <4 x float> [uses=1]
```

The example shows the LLVM IR equivalent for the SSE2 vector addition intrinsic (_mm_add_ps(__m128 a, b) in C/C++). The left-hand side of the expression is the name of the value. On the right hand side, the operation is followed by its operands, each with its type preceding the name (if types are equal, they can be omitted for all operands after the first). A comment is started with the special character ";". Comments at the end of a line usually denote the type of the value (the return type of the operation) and the number of uses of this value.

2.3.2 Data Types

The LLVM IR supports a large set of different data types. Table 2.1 shows examples for the most important types and their C/C++ counterparts.

LLVM Type	C/C++ Type	Explanation
`i1`	`bool`	truth value (`true` (`1`) or `false` (`0`))
`i8`	`char`	single character
`i32`	`int`	32bit integer value
`i64`	`long`	64bit integer value
`float`	`float`	32bit floating-point value
`type *`	`type *`	pointer to arbitrary type `type`
`<4 x float>`	`__m128`	vector of 4 32bit floating-point values
`<4 x int>`	`__m128i`	vector of 4 32bit integer values
`{ types }`	`struct { types }`	structure of arbitrary types `types`
[N × `type`]	`type t[` N `]`	array of size N of arbitrary type `type`

Table 2.1: *Examples for the most important LLVM data types and their C/C++ counterparts.*

2.3.3 Important Instructions

Most instructions of the IR are standard instructions that can be found in most assembly languages and need not be described in detail. However, there are a few that require some additional explanations:

- *Phi*
  As described in Section 2.1, the `phi` instruction chooses a value depending on which predecessor block was executed:

  ```
  %r = phi float [ %a, %bb1 ], [ %b, %bb2 ]
  ```

  The value of `r` is set to `a` if control flow came from block `bb1` and to `b` if `bb2` was the executed predecessor block.

- *Select*
  The `select` instruction returns either its second or third operand depending on the evaluation of its condition:

  ```
  %r = select i1 %c, float %a, float %x
  ```

  The value of `r` is set to `a` if condition `c` is `true` and to `b` otherwise. If the select statement has operands of vector type, we generate code that

creates a new vector by *blending* the two input vectors on the basis of a per-element evaluation of the condition vector (see Section 4.5.1). The terms "select" and "blend" are thus used interchangeably for the same operation.

- *GetElementPointer (GEP)*
  The `GetElementPointer` instruction returns a pointer to a member of a data structure. It receives the data structure and a list of indices as inputs. The indices denote the position of the requested member on each nesting level of the structure. In the following example, the first `GEP` (`r`) extracts a pointer to the `float` element of the nested struct `SUBS` of the struct `S` using `GEP` and stores `3.14` to that location:

```
%struct.S = type { i8*, i32, %"struct.S::SUBS" }
%"struct.S::SUBS" = type { i64, float, i32 }
%r = getelementptr %struct.S* %S, i32 0, i32 2, i32 1
store float 0x40091EB860000000, float* %r, align 4
```

  The first index is required to step through the pointer, the second index references the third element of the struct (which is the nested struct) and the third index references the second element of that sub-struct.

- *InsertElement / ExtractElement*
  The `InsertElement` and `ExtractElement` instructions are required if single elements of a packet have to be accessed:

```
%p2  = insertelement <4 x float> %p, float %elem, i32 1
%res = extractelement <4 x float> %p2, i32 1
```

  The first instruction inserts the `float` value `elem` at position `1` into packet `p`, the second instruction extracts the same `float` from the new SSA value `p2`.

Chapter 3
Related Work

This chapter summarizes previous work related to automatic packetization and shading languages.

Data-Parallel Programming

Generating code for parallel hardware architectures is being studied since the emergence of vector computers and array processors in the eighties. There has been a variety of attempts to provide the user with a simple, portable language that automatically compiles to parallel and/or packet code.

Examples are general purpose *data-parallel languages* such as NESL, Ct, or CGiS [4, 10, 8]. These languages allow to write vector code using built-in vector primitives. They compile to code that exploits the target-architecture's possibilities for parallel computing (e.g. SIMD instructions) without putting that strain on the programmer.

There are also target specific languages like the *Compute Unified Device Architecture (CUDA)* [20] or AMD-ATI's *Close To Metal (CTM)* [3]. These languages generate SIMT (Single Instruction, Multiple Threads) code for GPUs by managing execution of scalar *kernels* (small distinct functions) by multiple threads in parallel.

The section on shading languages gives examples of *domain-specific languages (DSLs)* that allow the user to program scalar code that is executed in parallel for different input data.

These approaches each implement their own compiler tool-chain with parsers, abstract syntax trees, classic and custom optimizations and code generation in order to compile to parallel code, reinventing the wheel time and time again. We avoid this by using LLVM as a compiler infrastructure that provides us with frontends, backends and optimizations while still

allowing for easy integration of custom analyses and optimization passes.

Automatic Parallelization

The term *Instruction-Level Parallelism (ILP)* describes the implicit parallelism that is available in scalar code due to instructions that do not depend on each other and therefore can be executed in parallel. *Automatic vectorization* aims at exploiting this parallelism, usually by transforming loops to vector form, e.g. after exposing ILP by loop unrolling.

The Parallel FORTRAN Converter (PFC) [2] that automatically translates FORTRAN programs to vector form is an early example for this. The system uses data dependency analysis to determine where vectorization is possible and employs *if conversion* [1] in order to have larger code fragments to work on. Although PFC transforms complete programs, only innermost loops are vectorized while the rest of the code remains scalar.

Superword-Level Parallelism (SLP) describes the occurrence of independent isomorphic statements (statements performing the same operations in the same order) inside a basic block. Such statements can be packed together and executed in parallel using SIMD (Single Instruction, Multiple Data) instructions. Unfortunately, this introduces overhead for the packing and unpacking of vectors that makes the approach unusable for smaller fractions of code. Larsen and Amarasinghe [17] implemented a compiler pass that successfully exploited SLP on AltiVec architectures, resulting in speedups of the vectorized function over the scalar source in the range of 1.2 to 6.7. Shin [29] extended the approach to also work in the presence of control flow by using predicates.

Such *predicated execution* [21] is supported by certain hardware architectures and provides the possibility of conditionally discarding results of an operation. This can be used as an alternative to relying on the speculative execution and branch prediction capabilities of a processor. Predicated execution also enables linearization of control flow [7] (except for loop edges) as we need it for our packetized code.

All these approaches have in common that they do not change the semantics of the program when transforming scalar to packet code. We make use of the knowledge that a program executes the same instructions on different data, exploiting *Data-Level Parallelism (DLP)* instead of ILP. Thus, we are able to transform entire functions or even programs instead of being bound to uncovering implicit parallelism between a few instructions inside a function.

Shading Languages

We evaluate our system in the context of real-time ray tracing (Chapter 5) and shading languages in general. Shading is possibly the largest bottleneck of rendering and thus, many approaches to shading languages, custom compilers, and infrastructures involve some kind of parallelization.

Related work in this area can be separated into the following categories:

- The shading language is interpreted by the rendering system either directly or by first compiling it to some intermediate representation.
- Shaders are compiled into the rendering system or are dynamically loaded as binaries. The rendering system and the shaders are executed on the same target platform. A separate compiler is used to translate the shading language to a general purpose language, which is then compiled to machine code.
- Shaders are executed on a different platform than the rendering system (e.g. on a GPU). A shading language compiler is used to compile machine code of the target platform.

A widely used, and de facto standard, shading language is the *RenderMan Shading Language (RSL)* [25]. The RSL compilers for the *REYES* rendering architecture, *Photorealistic RenderMan*, and *Blue Moon Rendering Tools (BMRT)* [5, 16, 11] transform RSL programs into byte-code representation and evaluate them at run time [13]. The byte-code instruction set is SISD (Single Instruction, Single Data) but execution happens in a virtual SIMD manner. However, since parallelization is virtual and instructions are interpreted, it is difficult to achieve maximum performance using such approach.

In real-time rendering systems, shading language compilers leverage the parallelism provided by the target architecture. On CPUs, this involves automatic packetization and optimization of scalar (SISD) shader programs as implemented in *RTSL* [22]. On GPUs, vectorization is implemented in hardware and *Cg*, *HLSL*, and *GLSL* operate in SISD [18, 23, 26]. Still, a translator from the shading language to machine instructions is required.

Sh [19] targeted GPUs and avoided the need for a parser by using C++ meta programs as a shading language embedded in the application. However, it still required a separate compiler to emit GPU opcode. Furthermore, the poor capabilities of C++ to serve as a meta-language host for domain-specific languages (e.g. the lack of control structures overloading) lead to syntactical and practical inconveniences.

A common deficiency of all above mentioned approaches is that they involve the task of implementing specialized parsers and compilers. Such complex infrastructure requires significant development effort, which has led to restricting the features and expressiveness of the shading languages. Additionally, specialized CPU compilers (e.g. for RTSL) have to perform most optimizations themselves. They produce vectorized C code and essentially use a standard compiler as a machine code emitter.

In contrast to all custom languages that involve compilation, our approach did not require building our own parser, intermediate representation and optimizations. We can benefit from every additional optimization that is implemented in LLVM directly.

Those approaches that circumvent compiler issues by interpreting the language directly suffer from bad performance penalties which largely limit the applicability of more sophisticated optimization techniques. Our system is able to optimize the scalar source code before packetization and can integrate the generated code seamlessly using inlining and specialization — the just-in-time compiler allows to even do this at runtime.

Considering portability, we are also much more flexible than the approaches we discussed. Due to the fact that the packetizer works on LLVM's IR, we can make use of all existing frontends and backends for LLVM. If support for a new language or target is required, it can build on LLVM's infrastructure that was designed to easily allow such expansions.

Finally, our approach transforms the control-flow graph of an already *optimized* function. This allows us to perform aggressive optimization of the scalar source code that might even restructure the CFG. In a structural approach (as usually employed by domain-specific languages), such optimizations are not possible because the correspondence to the AST is lost.

Chapter 4

Automatic Packetization

This chapter describes how our system automatically generates data-parallel, *packetized* functions from scalar code.

The algorithm receives three inputs: the scalar function, the (empty) prototype of the packetized function and the packetization size n. It replaces the prototype with packet code that produces the same results as calling the scalar function n times in parallel.

The packetization algorithm consists of six phases:

1. Preparatory transformations (Section 4.1)
2. Mask generation (Section 4.2)
3. Select generation (Section 4.3)
4. CFG linearization (Section 4.4)
5. Instruction packetization (Section 4.5)
6. Wrapper generation (if required) (Section 4.6)

First, preparatory transformations like scalar optimizations and loop simplification are performed. Then, masks that model control flow by keeping track of the active and inactive instances are computed for each block. Select instructions required to blend values of different instances are inserted. Linearization of the control-flow graph removes all control flow except for loops. Finally, the scalar instructions that are required to be executed in parallel are transformed to their packet counterparts. If the packetization size exceeds the SIMD width of the target architecture, a wrapper around the packetized function is generated.

The rest of the chapter is organized as follows: First, a short description explains how the system can be used by a programmer, followed by an

overview of supported types and constructs. Then, each step of the packetization is explained in detail. The last section lists the current restrictions of the system.

Usage of the System

The packetizer is implemented in C++ as an LLVM module pass. Figure 4.3 shows an example of how the packetizer can be used from within standard C/C++ with linked LLVM and packetizer libraries. The file "program.bc" is compiled from any source language that is supported by LLVM to target-independent bitcode. Figure 4.2 shows C code for this file that is compiled to bitcode using llvm-gcc. It runs a small test that compares results of the scalar and packetized functions. Note that all the programmer has to do is to implement the scalar function, declare a prototype for the packet function, use it as if it was implemented, and run the packetizer with the corresponding function names.

Figure 4.1 shows an overview of the whole process.

Supported Language-Constructs

The current implementation of the system includes support of the following constructs:

- Arithmetic and logic instructions
- Conditional branches
- Loops (including multiple exits)
- Calls to built-in packetized functions (sin, cos, log, exp, sqrt, rcp, ...)
- Calls to user-defined packetized functions (optionally masked)
- Load/store/scalar calls ("extract-execute-insert")
- Switch statements (through lowering)

It is basically capable of handling most code generated by the llvm-gcc frontend. The most significant remaining limitations are detailed in Section 4.7. The next section explains type restrictions that are necessary to allow packetization.

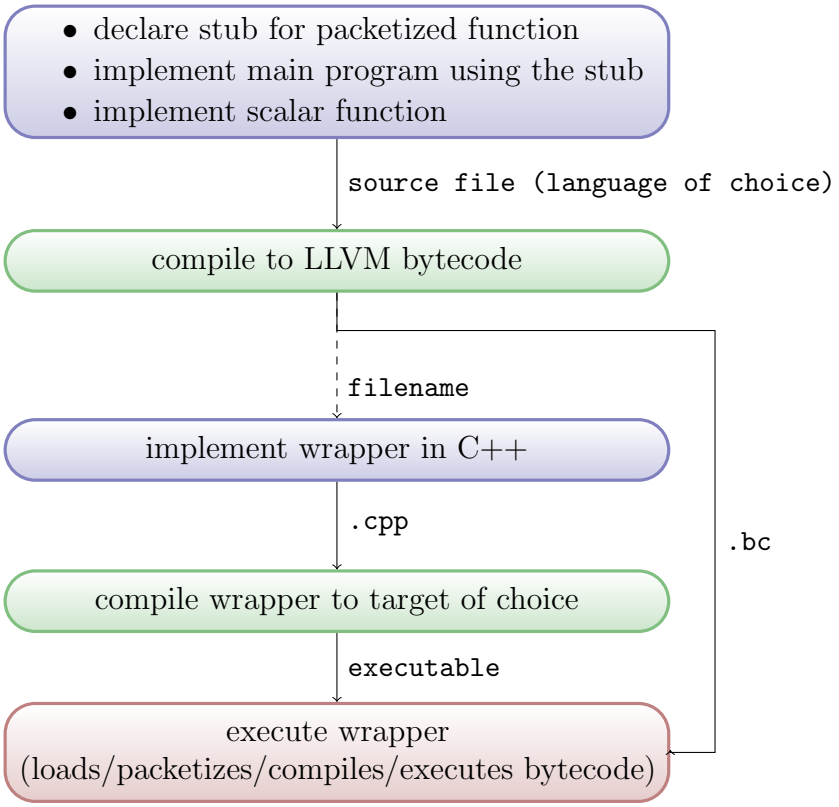

Figure 4.1: *Overview of how the packetizer can be used by a programmer. See Figures 4.3 and 4.2 for concrete code examples.*

```
#include "xmmintrin.h" //SSE2 intrinsics

//extract ith element of a __m128
inline float get(__m128& v, unsigned idx) {
  return ((float*)&v)[idx];
}

//empty prototype of packet function
__m128 generatedFn(__m128 a, __m128 b);

//implementation of scalar function
float scalarFn(float a, float b) {
  float x = a + b;
  float y = x * x - b;
  return x + y;
}

int main(int argc, char** argv) {
  //set vecA to 0.5/0.6/0.7/0.8
  __m128 vecA = _mm_set_ps(.5f, .6f, .7f, .8f);
  //set vecB to 1.0/2.0/3.0/4.0
  __m128 vecB = _mm_set_ps(1.f, 2.f, 3.f, 4.f);

  // compute result of generated function
  __m128 generatedRes = generatedFn(vecA, vecB);

  // compute results of scalar function
  float scalarRes0 = scalarFn(.5f, 1.f);
  float scalarRes1 = scalarFn(.6f, 2.f);
  float scalarRes2 = scalarFn(.7f, 3.f);
  float scalarRes3 = scalarFn(.8f, 4.f);

  // compare results
  bool equal = scalarRes0 == get(generatedRes, 0) &&
               scalarRes1 == get(generatedRes, 1) &&
               scalarRes2 == get(generatedRes, 2) &&
               scalarRes3 == get(generatedRes, 3);

  return (int) equal;
}
```

Figure 4.2: *Example C-code that could be compiled to LLVM bitcode by the llvm-gcc frontend. In Figure 4.3, this bitcode file ("program.bc") is loaded and the packetizer replaces the call to the prototype* ***generatedFn*** *by calls to the packetized version of* ***scalarFn****.*

```
#include "packetizationWrapper.h"

int main(int argc, char** argv) {
  // 1. create module from bitcode file
  const std::string filename = "program.bc";
  llvm::Module* mod = Wrapper::createModuleFromFile(filename);

  // 2. packetize desired function
  const unsigned packetSize = 4;
  const std::string sName = "scalarFn";
  const std::string pName = "generatedFn";
  Wrapper::packetizeFunction(packetSize, sName, pName, mod);

  // 3. get pointer to main() in module
  void* mainPtr = Wrapper::getPointerToFunction(mod, "main");

  // 4. compile with JIT, execute main and return result
  return Wrapper::executeMain(mainPtr, argc, argv);
}
```

Figure 4.3: *Example how the packetizer can be used. The file "program.bc" contains LLVM bitcode compiled from an arbitrary frontend (e.g. from llvm-gcc as shown in Figure 4.2). The packetizer is used to transform the function* ***scalarFn*** *of this program. Afterwards, the program's* ***main****, which now contains calls to the newly packetized function* ***generatedFn****, is executed.*

Data Types

The packetizer currently only allows functions as input that use the following data types:

- scalar types (float, int)
- pointers
- flat structs (no nesting)
- arrays

Figure 4.4 shows the set of rules that determine how scalar types are mapped to their packetized counterparts.

The packetizer uses these rules to determine if the function types of the scalar function and the packetized target signature specify a correct mapping. It also performs a check whether any instructions of the scalar function have unsupported types. Note that this especially does not allow usage of packetized types in the scalar code. For example, instead of using an SSE float vector to represent a coordinate, the user is required to use a struct or three or four separate floating point variables.

The checker allows two exceptions from these rules: First, packetized parameters are allowed to *match* their scalar counterparts in order to allow

$$
\begin{array}{rcl}
\texttt{packetize}(float) & \rightarrow & < N \times float > \\
\texttt{packetize}(int) & \rightarrow & < N \times int > \\
\texttt{packetize}(\tau *) & \rightarrow & \texttt{packetize}(\tau) * \\
\texttt{packetize}(\{\ \tau\ \}) & \rightarrow & \{\ \texttt{packetize}(\tau)\ \} \\
\texttt{packetize}([\ \tau\ ]) & \rightarrow & [\ \texttt{packetize}(\tau)\ ]
\end{array}
$$

Figure 4.4: *Type packetization rules for SIMD width N. Less-than and greater-than "brackets" denote a packet, curly brackets a struct, square brackets an array.*

per-packet constant parameters. Second, boolean values are allowed to map to vectors of *integer* type.

4.1 Preparatory Transformations

This section briefly summarizes the required preparatory transformations. They are basically simple standard-transformations and therefore not described in detail. Note that all transformations preserve the semantics of the scalar code.

Classic Optimizations Before starting any packetization related transformations, the scalar function is optimized using LLVM-internal classic optimizations like constant propagation, global value numbering, or control-flow simplification.

These optimizations are likely not to have a large impact if applied to packetized code, which lacks important explicit information (e.g. control-flow information is encoded in masks).

Return Unification The function is required to only have a unique return statement because some instances might return earlier than others. This way, the function is guaranteed to only return in a synchronized way after all instances finished their computations. Figure 4.5 depicts a situation that requires this transformation.

All return statements are transformed into unconditional branches that lead to a new exit block which holds a single return. In case the function has a non-void return value, the return statement references a ϕ-function which selects the correct value from all different incoming edges.

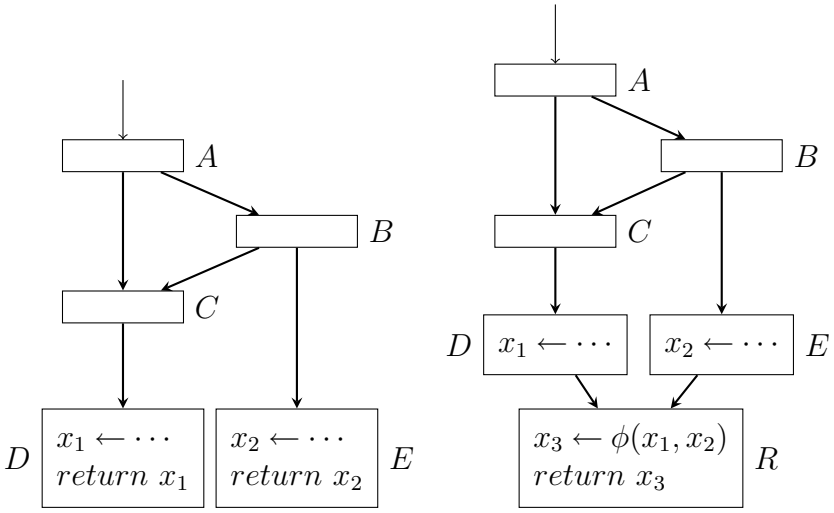

Figure 4.5: *The return unification pass transforms the CFG on the left to the CFG on the right. The transformation guarantees that the function only has one return statement. This is necessary because the packetized function must not return if any instance is active on a different path.*

Loop Simplification The loop simplification step ensures that each loop (and each nested loop) has exactly one incoming edge and one backedge. This implicitly guarantees the existence of a loop preheader (the block from which the loop is entered) and a loop latch (the block from which an edge leads back to the header).

The transformation is required in order to reduce the generated code during select generation: If a loop was allowed to have n backedges, we would have to generate n times as many `select` statements (see Section 4.3).

Having a single preheader and a single latch also reduces the complexity of the mask and select generation and CFG linearization algorithms that benefit from the simplified recursive CFG traversal when loops are involved. For example, it would be far more complicated to find the correct order of blocks of a loop which has multiple preheaders and/or multiple latches (see Section 4.4.

Phi Canonicalization During packetization, all ϕ-functions have to be replaced by blend operations that select from two values based on a condition (see Section 4.5). However, a ϕ-function may in general have arbitrarily many incoming values whereas a `select` operation can only blend two values at once. Hence, we insert dummy blocks with ϕ-functions that have at most

two incoming values.

This issue could also be handled at the packetization stage by transforming ϕ-functions with n incoming values into $n-1$ `select` instructions.

4.2 Mask Generation

As already mentioned, control flow may diverge because a condition might be `true` for some of the elements of the packet and `false` for other ones. Consequently, *all* code is executed. The explicit transfer of control is modeled by mask variables (in short: masks, also often called predicates) on control-flow edges. If a mask of a CFG edge $B \to B'$ is set to `true` at position i, then the i-th instance of the code took the branch from B to B'. Thus, the mask denotes which elements in a packet contain valid data on the corresponding control-flow edge.

The edge masks implicitly define entry masks on blocks: The entry mask of a block is either the disjunction of the masks of all incoming edges or—in case of a loop header—a ϕ-node with incoming values from the loop's preheader and latch. The masks of the control-flow edges leaving a block are given by the block entry mask and a potential conditional. If a block exits with an unconditional branch, the mask of its single exit-edge is equal to the entry mask. If the exit branch is conditional, the exit mask of the "`true` edge" of the block is the conjunction of its entry mask and the branch condition. The exit mask of the "`false` edge" is the conjunction of the entry mask and the negated branch condition.

Figure 4.6 shows three basic blocks A, B, C with corresponding block entry masks $(m_A, \dots)$, edge masks $(m_{A\to B}, \dots)$ and their computation.

The algorithm recursively traverses the CFG bottom-up, starting at the exit block of the function. If the currently processed block already has an associated mask, that mask is returned and recursion stops. Loops are handled by keeping track of all visited blocks, stopping recursion if a block is encountered a second time.

4.3 Select Generation

At control-flow merge points, packets might need to be *blended*, i.e. combined to a single one. Consider the example in Figure 4.6. The variable x has been defined in the left and the right predecessor of B. In the packetized version, control flow is no longer present and all code is executed. Hence, the correct contents of the *packet* for x in B have to be produced by masking out the

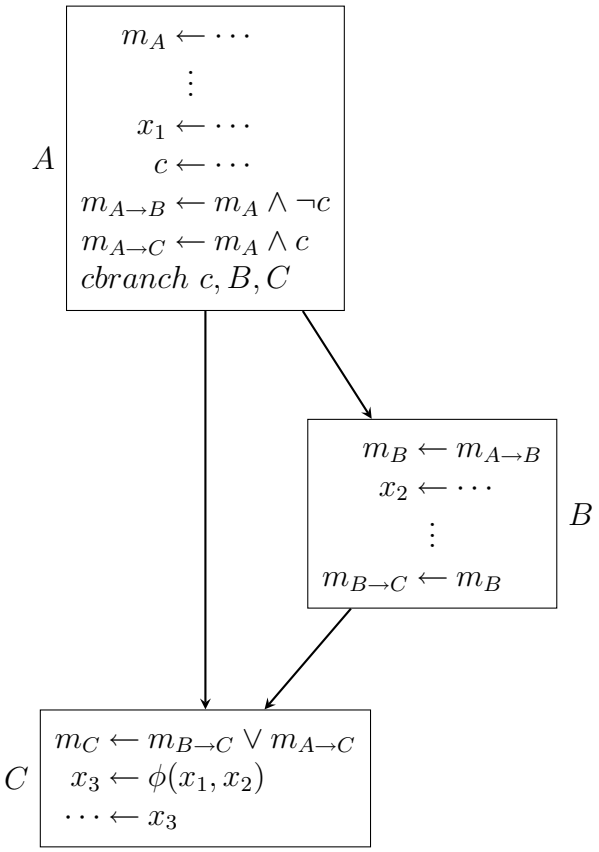

Figure 4.6: *Edge and block entry masks. m_A, m_B, and m_C are the entry masks of the corresponding blocks A, B, and C. $m_{A\to B}$, $m_{A\to C}$, and $m_{B\to C}$ are the block exit masks connected to the edges $A \to B$, $A \to C$, and $B \to C$.*

irrelevant parts of each edge and combining the results. This is usually done with `and`, `or`, and `xor` operations; some architectures (such as SSE4.1) feature special blend instructions (see Section 4.5.1).

The select generation pass replaces a subset of the ϕ-functions of LLVM's SSA form (see Section 2.1) by scalar `select` instructions (described in Section 2.3.1).

All ϕ-nodes are replaced except for those related to values in loops. These are still required after packetization because loop headers still have more than one incoming edge.

The `select` instruction is generated by using the mask of one incoming edge of the block as the condition and the corresponding incoming value of the ϕ-function as the first operand of the `select`. The second operand is set to the other incoming value, whose elements are selected if the mask is `false`.

4.3.1 Loops

Generating `select` operations for loops requires special attention. In the packetized version, the loop condition is a packet, not a scalar value. Hence, the loop iterates until the *last* element in the condition's packet becomes `false` (see Figure 4.7). This means that the loop is still running although (in the scalar version) an instance might already have left it. By using special loop-exit masks and corresponding blend operations we make sure that all elements of packets of such an instance are not modified after the instance leaves the loop.

Loop-Exit Masks First, a ϕ-function is generated in the loop header for each edge leaving the loop (loop exits). This ϕ-function holds the mask of that exit for the current iteration. The mask is `true` for each scalar instance that has left the loop through this exit. This implies that when the loop is finally left in the packetized version, all exit masks combined have exactly one value per index that is `true` (not considering instances that initially did not enter the loop).

When the loop is entered, all elements of all exit masks are initialized with `false`. At the time an instance leaves the loop, the corresponding new exit mask is computed as the disjunction of the exit condition and the mask of the current iteration (which is the result of the ϕ-function connected to this exit mask). If the backedge of the loop is taken and the header is executed again, the corresponding ϕ-function returns this new mask. Figure 4.10 shows how exit masks are computed and updated.

```
int x;
while (x < 8) {
  ++x;
}
```

```
int x0, x1, x2, x3;
while (x0 < 8 || x1 < 8
    || x2 < 8 || x3 < 8) {
  if (x0 < 8) ++x0;
  if (x1 < 8) ++x1;
  if (x2 < 8) ++x2;
  if (x3 < 8) ++x3;
}
```

```
VEC x;
VEC cond = x < VEC(8);
while (some element of cond is true) {
  VEC tmp = x + VEC(1);
  x = select(cond, tmp, x);
  cond = x < VEC(8);
}
```

Figure 4.7: *C code for a scalar loop, a scalar implementation that mimicks the required behavior of the packet code, and pseudo-code for a possible packet implementation. The packet code performs one (vector) addition, followed by a select operation that resets the values of all inactive instances (those where the element of the packet x is larger than 8): If the scalar version is executed 4 times on the input values $0, 2, 4, 6$, the scalar loop iterates $8, 6, 4,$ and 2 times respectively. The packet loop on the other hand iterates 8 times on a packet input with the same values, but only in the first two iterations all instances are active (and thus the* `select` *returns all elements of* `tmp`*). After the second iteration, instance four becomes inactive, after the fourth iteration instance three and so on until all have finished. The state of all inactive instances is "freezed" by the select operation.*

Loop Live Values In order to be able to generate the correct `select` instructions, we need to know what values have to be blended. All local values of the loop can be ignored because they do not have any effect visible outside the scope of a single iteration. The values remaining for select generation are all those that are *live across loop boundaries*.

A value is defined as *live across loop boundaries* if it is used either in a subsequent iteration or outside the loop. In favor of better readability, the term *live* in the following always stands for *live across loop boundaries*.

Live values are collected per block of the loop by iterating over the uses of all instructions of that block. If an instruction has at least one use outside the loop or a use in the loop header that is a ϕ-function, it is considered *live across loop boundaries*.

If a live value is not connected to a ϕ-function in the loop header (e.g. a value that is defined in the loop body but only used outside the loop) a dummy ϕ-node is generated. This is required in order to be able to blend with the value of the last iteration.

Figure 4.8 shows code for the collection of loop live values.

```
LiveValueMap findLoopLiveValues(Loop* loop, PHINode* maskPhi) {

  LiveValueMap liveValues;

  forall blocks BB in loop {
    forall instructions I in BB {
      if (findUseOutsideLoopBoundary(I, loop)) {
        //found live value with use outside loop-boundary
        PHINode def = findLoopPhiForInstruction(I, loop);

        if (def not found) {
          //generate dummy phi
          def = generateLoopPhiForInstruction(I, loop);
        } else if (def is loop mask phi) {
          //preceding def for value is loop mask phi -> ignore!
          continue;
        } else if (def is loop induction variable) {
          //preceding def for value is induction phi -> ignore!
          continue;
        }

        //found preceding def for value, add to map
        liveValues.insert(std::make_pair(I, def));
      }
    }
  }

  return liveValues;
}
```

Figure 4.8: *Pseudo-code for collecting all values of a loop that are live across loop boundaries*

```
void generateLoopSelectsForExitingBlock(BasicBlock* exitingBB) {
  forall live values LV {
    //if exitmask is true, conserve result of previous iteration
    //if it is false, blend iteration-result
    PHINode* resultPhi = getLiveValuePhi(LV);
    Value* trueVal = resultPhi;
    Value* falseVal = resultPhi->getIncomingValue(latchBB);

    SelectInst* select = SelectInst::Create(
      combinedLoopExitMask, trueVal, falseVal, latchBB);

    //make sure we do not generate a select that is not
    //dominated by its second operand
    //(first operand is phi -> always dominates select)
    if (isDominatedBy(falseVal, select))
      falseVal->moveBefore(select);

    //now we have to set the corresponding resultphi
    //to the appropriate last definition
    resultPhi->setIncomingValueForBlock(latchBB, select);
  }
}
```

Figure 4.9: *Pseudo-code for select generation of a loop exit.*

Loop Selects To preserve correctness of all different instances in the packetized function, `select` instructions have to be added to the loop latch. These `selects` guarantee that the *state* of all instances that left the loop remains stable even if the loop is still being iterated for other active instances.

The *state* of an instance inside a loop is defined as the set of all values of that instance that are *live across loop boundaries*.

Maintaining a stable state for an inactive instance is achieved by "freezing" all corresponding live values at their state after the last complete iteration before leaving the loop. This is obtained by generating a `select` instruction for each live value in the loop latch. Before the loop is executed again, these `selects` *reset* all live values of inactive instances to their freezed state of the last iteration, discarding all effects of the current iteration: If the corresponding loop-exit mask (the condition of the `select`) is `false`, the instance is still active and the new value computed in the current iteration is selected. If the mask is `true`, the instance has left the loop at this exit and thus the value of the previous iteration is selected. Figure 4.10 gives an example and 4.9 shows pseudo-code for the algorithm.

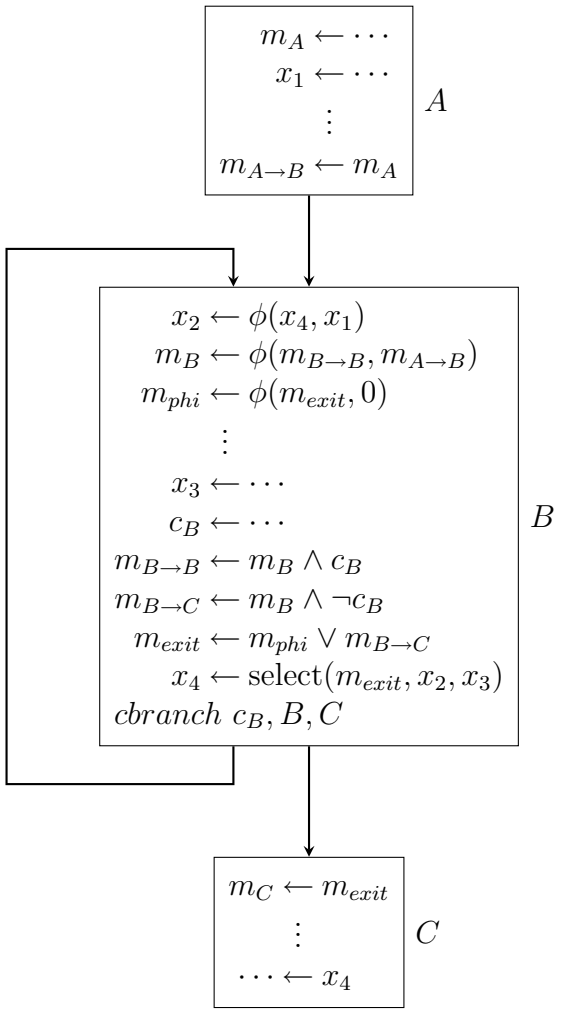

Figure 4.10: *Mask and select generation for a loop. In general, each exit is assigned a mask operation m_{exit} and a ϕ-function m_{phi}. The mask operation updates the exit mask by setting elements of instances that leave the loop in the current iteration to* `true`*. The ϕ-function holds the current exit mask. Note that, after this pass, the mask of the edge $B \to C$ is m_{exit} instead of $m_{B\to C}$. The stable state of all instances that leave the loop early is maintained by the* `select` *instruction x_4 in the latch. As long as the packetized loop iterates, all inactive instances are always reset to the value of the last iteration (the result of the ϕ-function x_2) by this* `select`*.*

Multiple Loop-Exits Allowing loops with multiple exits leads to the following problem: `Select` statements that blend values coming from different paths may produce wrong results for elements which belong to instances that are active on a distinct, third path. In a CFG without loops, those garbage values are corrected by other `select` statements in the block where the paths join later.

However, if such a situation occurs inside a loop and the third path leaves the loop through a different exit, the garbage value is not masked out. This is because the mask that is used for blending only incorporates the information of the *last* loop-exit before the latch, so the incorrect value is transferred into the next iteration. Figure 4.11 illustrates the problem.

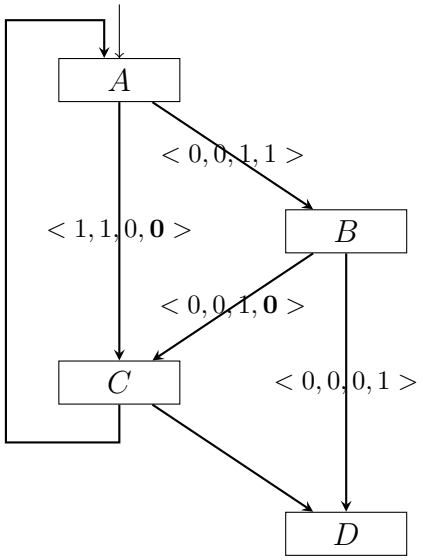

Figure 4.11: *Example CFG where packetization of a loop with multiple exits would produce incorrect code without maintaining a combined loop-exit mask. The vectors show a snapshot of the masks of all edges when control flow reaches block C. It can easily be seen that the two joining masks from A and B are not complementary: both of their last indices are* `false`*. The* `selects` *at the entry of this block produce wrong values for this instance by selecting a value from one of the two paths although the instance has left the loop. Without additional modifications, these garbage values would not be reset before the next iteration. Thus,* `select` *operations in the latch reset all inactive instances to their state of the last iteration. By using a combined mask, it does not matter where they left the loop.*

Our solution for this is to mask out results of inactive instances *inde-*

pendent of where they left the loop. To this end, an additional, *combined* loop-exit mask is required that combines the masks of all loop exits: after each update of an exit mask, the combined mask is also updated. This way, the combined loop mask keeps track of *all* inactive instances, which allows us to maintain their stable state with only one `select` instruction per live value. This `select` resets all effects on the corresponding live value for all instances that are inactive, no matter where they left the loop. Figure 4.12 gives a more complex example of a CFG with a multiple-exit loop.

4.4 CFG Linearization

After all mask and select operations are inserted, all control flow except for loop backedges is effectively encoded by data flow and can thus be removed. To this end, the basic blocks have to be put into a sequence that preserves the execution order of the original CFG G: If a block A executed before B in every possible execution of G then A has to be in front of B in the flattened CFG G'. If the CFG splits up into two paths, one path is chosen to be executed entirely before the other. The decision which path to execute first is currently non-deterministic, other possibilities are discussed in Section 7.2.

Such an order is determined by topologically sorting the blocks recursively over the loop tree of G. The result is a CFG that only has conditional branches remaining at loop exits and unconditional branches at loop entries. All other branches can be removed. Figure 4.13 shows the flattened CFG of the example in Figure 4.6.

4.4.1 Block Ordering

The linearized block ordering is computed by a recursive bottom-up traversal of the CFG and stored in a list.

First, all blocks that do not belong to a loop are collected in execution order. This is straightforward, as we only have to add all those blocks to the list that do not belong to a loop while we return from recursion.

Then, ordered block lists of all top-level loops are computed, including recursive ordering of all subloops. Each loop by itself is ordered by adding all blocks to the corresponding list that belong to the same loop and have the same loop depth. This is again obtained by a bottom-up traversal of the CFG starting at the latch of the loop. The algorithm keeps track of the visited blocks and stops recursion if it encounters a block that does not belong to the current loop or a block that was visited before. After all orderings of subloops and the ordering of the loop itself are determined, the ordered list

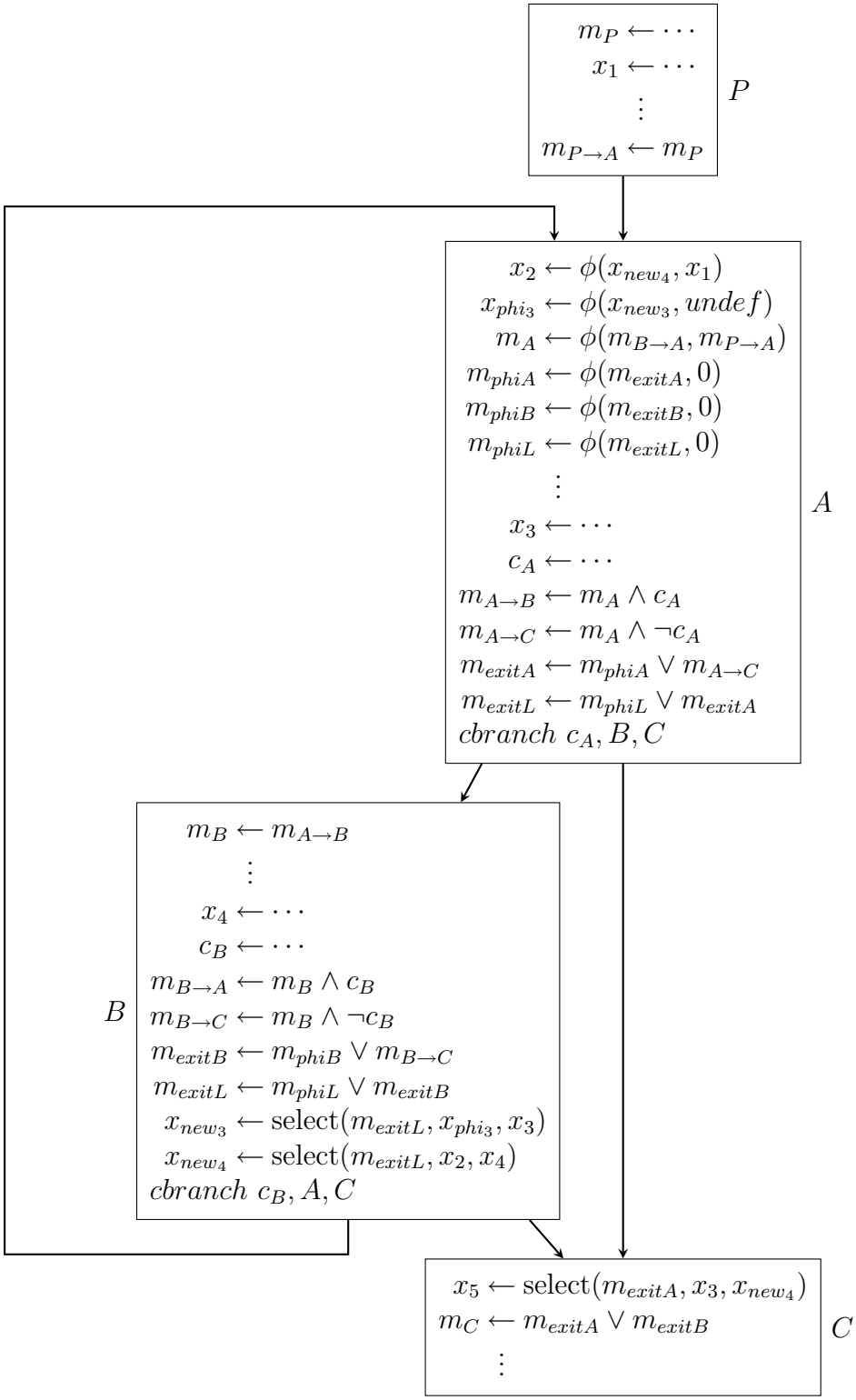

Figure 4.12: *Mask and select generation for a loop with multiple exits. The stable state of all inactive instances is maintained by the ϕ-functions x_2 and x_{phi_3} together with the combined mask m_{exitL} and the* `select` *instructions. For each live value we need one* `select` *in the latch, e.g. x_{new_4} discards the results of x_4 for all inactive instances by setting the corresponding elements back to x_2.*

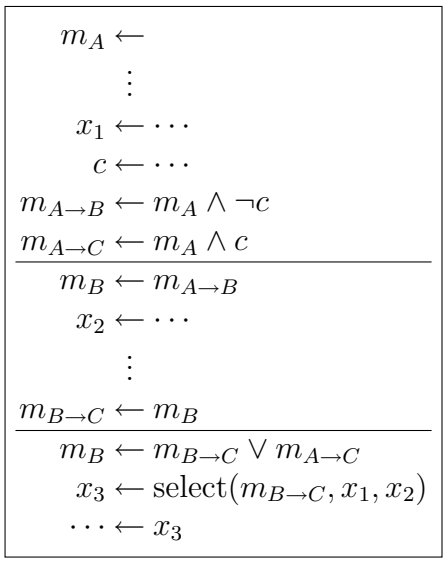

Figure 4.13: *The flattened control-flow of Figure 4.6 with value blending.*

of each subloop is merged into the list of the parent loop at the position behind its preheader. The preheader is guaranteed to be in the list because it belongs to the parent loop by definition.

Finally, all top-level loops are merged into the final list by the same technique.

4.4.2 Linearization

After a correct ordering is determined, linearization is straightforward. Each block in the ordered list is first tested for a backedge to a loop header.

If it has none, the block's branch (no matter if conditional or unconditional) is replaced by an unconditional branch to the next block in the list. In combination with the select generation pass, this implicitly performs *if conversion*.

If the block has a conditional branch with a backedge, the other edge that leaves the loop is replaced by an edge to the next block in the list.

If the block only has a backedge to the header, its unconditional branch is replaced by a conditional branch with edges to the header and the next block in the list with the exit mask of the block as the branch condition.

Figure 4.14 shows the linearization of a CFG with complex control flow.

```
void scalarFn(float a, float b, Color& color) {
  float x = a + b;
  float y = x * x - b;
  float z = y;

  for (int i=0; i<y; ++i) {
    z += a;
    if (z / a < x) z += a;
    else {
      z -= b;
      if (a > b) z -= b;
      else {
        z *= z-y;
        if (b == a) {
          z = a+3;
        } else {
          ++z;
        }
      }
      for (int j=0; j<100; ++j) {
        if (i < j) z += a;
        else z -= 13.2f;
      }
    }
  }
  z = z-y;
  color = Color(z);
}
```

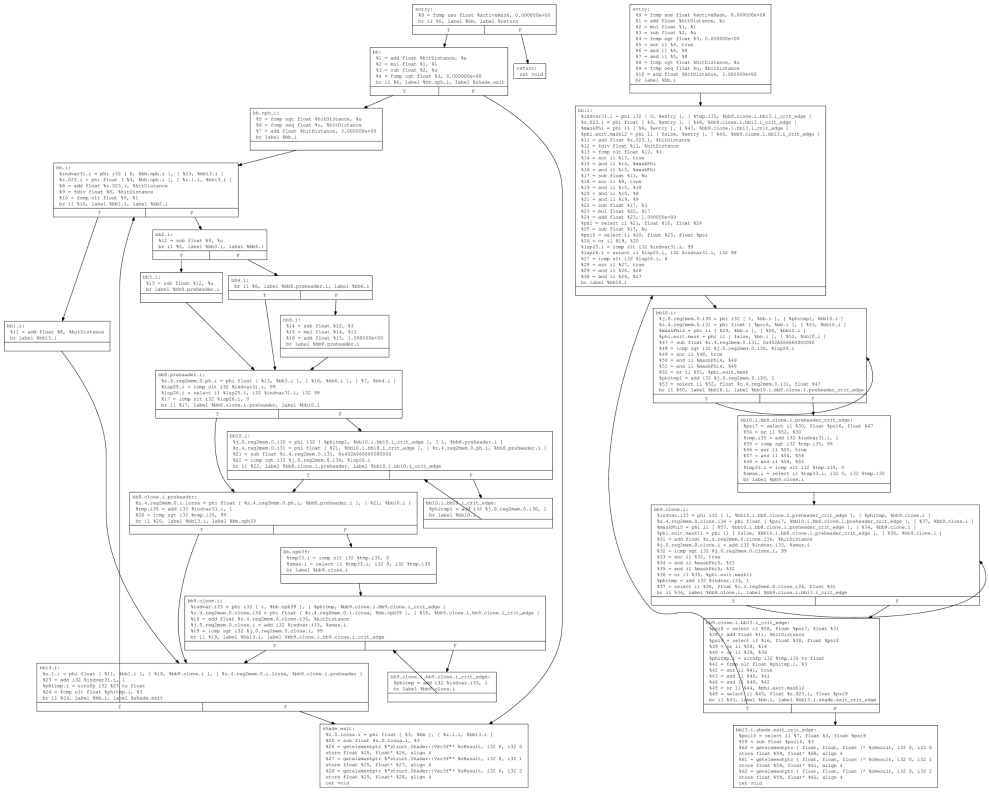

Figure 4.14: *A function with complex control flow and its corresponding CFGs before and after transformations. The left CFG is the (optimized) input of the packetizer, the right CFG shows the same function after the linearization pass.*

4.5 Packetization

After blocks have been linearized, the scalar instructions are transformed into their packetized counterparts. The instructions to be packetized are determined by dependence analysis on the function's outputs (return, store, and call instructions). All operations that are data-dependent on packetized inputs (parameters of the function) are packetized as well. Finally, all expressions that mask computations depend on are also packetized.

By this, we ensure that all paths that are packet-invariant and therefore do not require packetization are left untouched.

Packetizing a single instruction is basically a one-to-one translation from the scalar instruction to its SIMD counterpart (see Figure 4.15). This holds for all instructions except for `GetElementPointer` (`GEP`), `Select`, `Load`, `Store`, and `Call` that have to be handled separately. This is because they require more complex operations, which will be described in the following sections.

```
define <4 x float> @generatedFn(<4 x float> %a, <4 x float> %b) {
entry:
  %0 = add <4 x float> %a, %b   ; <<4 x float>> [#uses=3]
  %1 = mul <4 x float> %0, %0   ; <<4 x float>> [#uses=1]
  %2 = sub <4 x float> %1, %b   ; <<4 x float>> [#uses=1]
  %3 = add <4 x float> %0, %2   ; <<4 x float>> [#uses=1]
  ret <4 x float> %3
}
```

Figure 4.15: *LLVM assembly produced by packetization of* `scalarFn` *from Figure 4.2.*

4.5.1 Select Instructions

Architectures supporting SSE4.1 provide the `BLENDVPS` intrinsic that performs a vector select on the basis of a mask in a single operation. If this (or similar intrinsics known by the LLVM code generator) is not available, bitwise operations have to be used for blending the vectors:

```
define <4 x float> @blend(<4 x float> %a, <4 x float> %b, <4 x float> %m) {
entry:
  %m2   = bitcast <4 x float> %m to <4 x i32>
  %a2   = bitcast <4 x float> %a to <4 x i32>
  %b2   = bitcast <4 x float> %b to <4 x i32>
  %xor  = xor <4 x i32> %m2, < i32 -1, i32 -1, i32 -1, i32 -1 >
  %and1 = and <4 x i32> %b2, %xor
  %and2 = and <4 x i32> %m2, %a2
  %or   = or <4 x i32> %and2, %and1
  %res  = bitcast <4 x i32> %or to <4 x float>
  ret <4 x float> %res
}
```

On current CPU architectures including our test system (see Section 5.4), `BLENDVPS` unfortunately is only a compound instruction and therefore does not result in noticeable speedup over the bitwise operations.

On the code generation side, LLVM (as of version 2.5) unfortunately is not able to generate code for `select` instructions that receive three vectors of type `float`. The code generator only accepts `vector selects` with a condition of type `<4 x i1>` which would require insertion of additional comparison operations. Hence, we implemented packetization of `select` instructions by hand, replacing each `vector select` either by the bitwise operations or the `BLENDVPS` intrinsic depending on the architecture's support of SSE4.1.

Pointer Selects

Select instructions that have pointers as input values can not be packetized on SSE architectures. This is because vectors of pointers are not allowed (there is no need for such constructs without scattered loads/stores).

There are two methods for dealing with such constructs apart from forcing the frontend not to generate them in the first place (if an appropriate flag exists). One way is to "pull the uses through the `select`". This means to first load from each of the input values of the `select` and then blend the loaded values instead. However, this results in unnecessary loads. Thus, we chose to have the packetizer split such `selects` and their uses into scalar code. Uses may only be load or store instructions or additional `pointer selects`. Due to the fact that load and store instructions in most cases have to be split up anyway (Section 4.5.3), the performance overhead is negligible.

The algorithm works as follows: First, each `select` is split into N scalar `selects`. Each use that is a load is split into N scalar load instructions that load from the results of the different scalar `pointer selects`. The results are then inserted into a vector using N `InsertElement` instructions. Each use that is a store is split into N scalar instructions that store to the results of the different scalar `pointer selects`. In case the `select` is used by another `pointer select`, the algorithm recurses.

4.5.2 Native Function Calls

If the function contains any calls that could not be inlined, the packetizer first tries to find a *native*, packetized version of that function. Such versions can either be built-in or supplied by the user.

Built-in packetized functions currently include: sine and cosine, logarithm, exponential function, square root and inverse square root, reciprocal, and round/floor/ceil.

An example for user-supplied functions are API calls of a shader that call renderer-specific functions (see Section 5.2.2). These functions can also specify an additional parameter for a mask, allowing called functions to profit from that information.

The packetizer currently does not perform recursive packetization of unknown functions (see Section 7.2).

4.5.3 Instructions with Side-Effects

Calls to scalar functions that cannot be packetized, as well as loads and stores, have to be split into N masked scalar statements. This is because we have to conservatively expect these instructions to produce side effects that we do not want to occur for inactive instances. Thus, we have to guard each scalar execution by an `if` construct that skips the instruction if the mask of that packet-index is `false`. Figure 4.16 shows such an *if-cascade*.

Unfortunately, this involves a lot of extract- and insert-operations that reduce the overall benefit of packetization. The only way to circumvent this is hardware support through conditional load/store instructions (see Section 7.2).

When packetizing a `call` instruction, scalar values might have to be replicated. If the call expects a packet argument but the supplied value is scalar, the packetizer automatically generates a packet by replicating the value N times.

4.5.4 GetElementPointer Instructions

A `GEP` instruction can only be transformed directly if it has no packetized index. In such a case the instruction remains untouched because it still returns a pointer to a valid element after packetization. A packetized index implies that several elements that possibly reside at different locations in memory should be accessed by load or store instructions.

This requires *splitting* of all such `GEP` instructions because LLVM currently does not support scattered loading or storing and thus does not allow vectors of pointers.

The splitting algorithm works as follows: First, each `GEP` is replaced by N `GEP` instructions that return scalar pointers. Then, all its uses that are either loads or stores are also split into N scalar instructions that use the scalar `GEP` of the corresponding instance as their pointer operand. If the use was a load, the results of the N load instructions are gathered by `InsertElement` instructions to form a single result vector of size N. An example is shown in Figure 4.17.

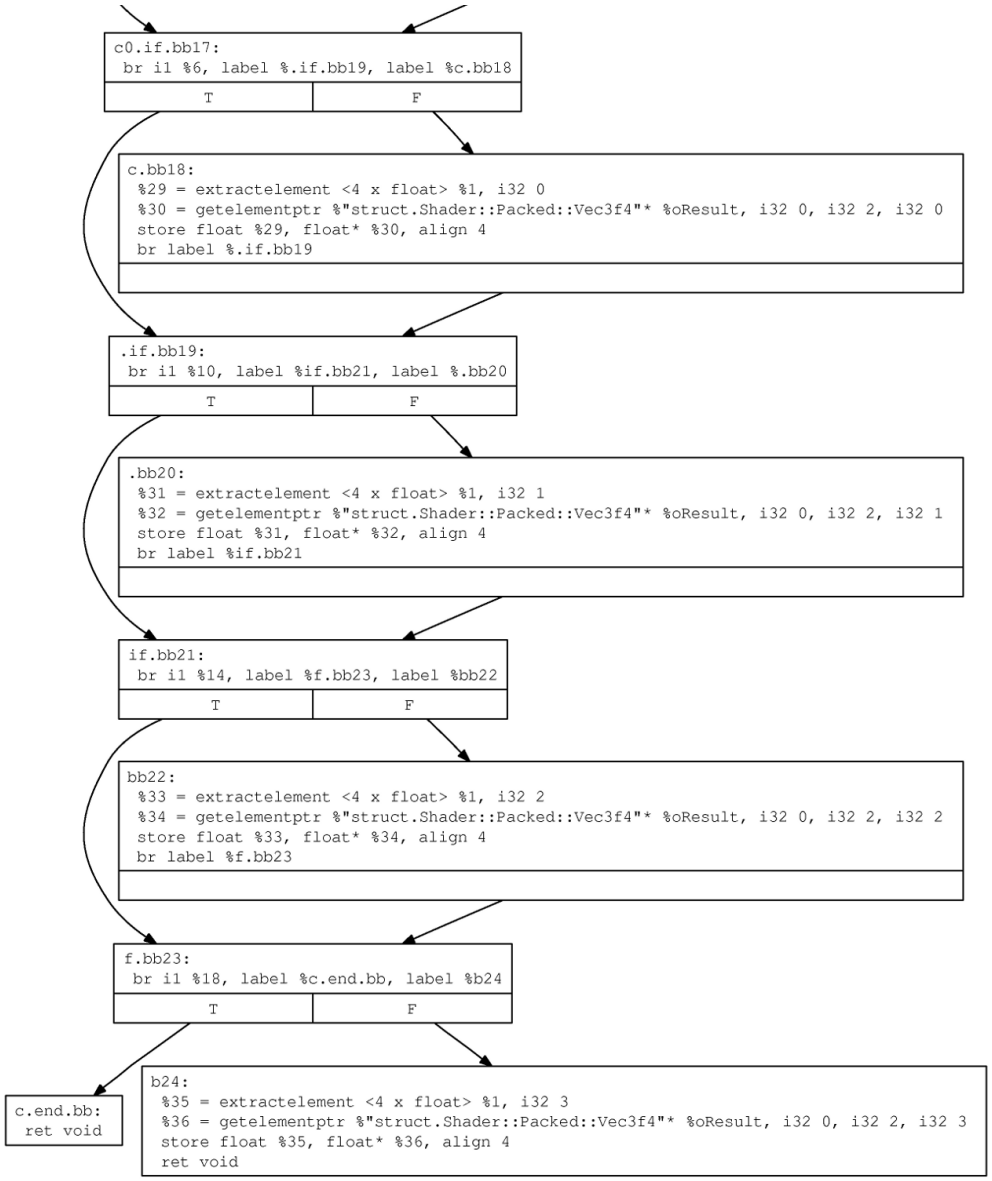

Figure 4.16: *The last blocks of the CFG of a shader function (Chapter 5): An* if-cascade *stores back result values to a pointer argument of a packetized function. This is necessary because we can not store back the entire packets: inactive instances must not modify the result.*

```
%struct.Color = type { float, float, float }
%r = getelementptr %struct.Color* %color, i32 0, i32 2
store float 0x40091EB860000000, float* %r, align 4
```

```
%struct.ColorVec = type { <4 x float>, <4 x float>, <4 x float> }
%r = getelementptr %struct.Color* %color, i32 0, i32 2
store <4 x float> 0x40091EB860000000, <4 x float>* %r, align 16
```

```
%struct.ColorVec = type { <4 x float>, <4 x float>, <4 x float> }
%m0 = extractelement <4 x i32> %mask, i32 0
%m1 = extractelement <4 x i32> %mask, i32 1
%m2 = extractelement <4 x i32> %mask, i32 2
%m3 = extractelement <4 x i32> %mask, i32 3
%r0 = getelementptr %struct.ColorVec* %colorvec, i32 0, i32 2, i32 0
%r1 = getelementptr %struct.ColorVec* %colorvec, i32 0, i32 2, i32 1
%r2 = getelementptr %struct.ColorVec* %colorvec, i32 0, i32 2, i32 2
%r3 = getelementptr %struct.ColorVec* %colorvec, i32 0, i32 2, i32 3
if (m0) store float 0x40091EB860000000, float* %r0, align 4
if (m1) store float 0x40091EB860000000, float* %r1, align 4
if (m2) store float 0x40091EB860000000, float* %r2, align 4
if (m3) store float 0x40091EB860000000, float* %r3, align 4
```

Figure 4.17: *LLVM-code that stores `3.14` to the second element of struct `Color`. The first code-snipped shows the scalar code, the second shows the one-to-one packetization, which is illegal due to the packetized index to the `GEP`. The third snipped depicts the valid packet code (with pseudo-guarded `store` operations for better readability, see Figure 4.16). In the packetized code, the struct contains three packets instead of scalar values. The `stores` must not be executed for inactive instances, so they have to be* split *and guarded by* if *constructs using the block's mask. As the `GEP` returns a pointer to a packet, it also has to be split and receive an additional index for the element of the packet.*

4.6 Wrapper Generation

After packetization is finished, a cleanup phase performs a few optimizations, e.g. removing redundant bitcast instructions, control-flow simplification and dead code elimination. This finishes the generation of the packetized function that works on packets of size N.

If the desired packetization size S is a multiple of N, a wrapper is generated. This wrapper basically loops over the generated function i times, where i is given by division of S by N as can be observed in Figure 4.18.

In order to call the function with the correct arguments it performs extract operations using the loop iteration index. If the function has struct parameters, the wrapper also allocates memory for the "smaller" structs that the function is called with. In each iteration, the values of the struct that correspond to the current iteration are extracted and stored to that temporary struct.

Additionally, the wrapper needs to allocate memory both for the result (if the function does not return void) and all pointer arguments because it has to make sure that no effects of the function are lost. To this end, the result and all values of pointer arguments are stored back after the call returns.

4.7 Restrictions

While our system is able to packetize a large fraction of the bitcode generated from C/C++, it does have some restrictions.

Trivially, the packetized function, and everything that is called by it, must allow to be executed in parallel. We do not analyze for parallelizability but leave the choice to the user.

Furthermore, the used data types must be restricted to what is supported by the SIMD architecture on the target machine. For example, on SSE, `double` arithmetic cannot be used, if the packet size is 4.

Chapter 7 explains additional features that are not implemented yet, e.g. recursive packetization and support for nested structs.

```
const unsigned size = packetizationSize / simdWidth;

struct S {
  __m128 x[size];
  __m128 y[size];
  __m128 z[size];
};

struct SUBS {
  __m128 x, y, z;
};

__m128 generatedFn(__m128 a, __m128 b, SUBS* s);

__m128* wrapperFn(__m128* a, __m128* b, S* s) {
  __m128* ret = new __m128[size];
  SUBS subs;

  for (unsigned i=0; i<size; ++i) {
    subs.x = s->x[i];
    subs.y = s->y[i];
    subs.z = s->z[i];
    ret[i] = generatedFn(a[i], b[i], &subs);
    s->x[i] = subs.x;
    s->y[i] = subs.y;
    s->z[i] = subs.z;
  }

  return ret;
}
```

Figure 4.18: *C code that corresponds to the the wrapper that we generate directly in LLVM's IR.* ***generatedFn*** *is the generated function that only works on SIMD packets that are "smaller" than the arguments of the desired function by a factor of "size". This means we have to extract values for each iteration from the argument-arrays and the struct. In case of a struct, this means we have to use a temporary struct of the "smaller" size (extract elements from struct of type S and build struct of type SUBS). In case of a pointer argument, we also have to store back the corresponding value(s) after the call returns.*

Chapter 5

AnySL: Language-Independent Shading

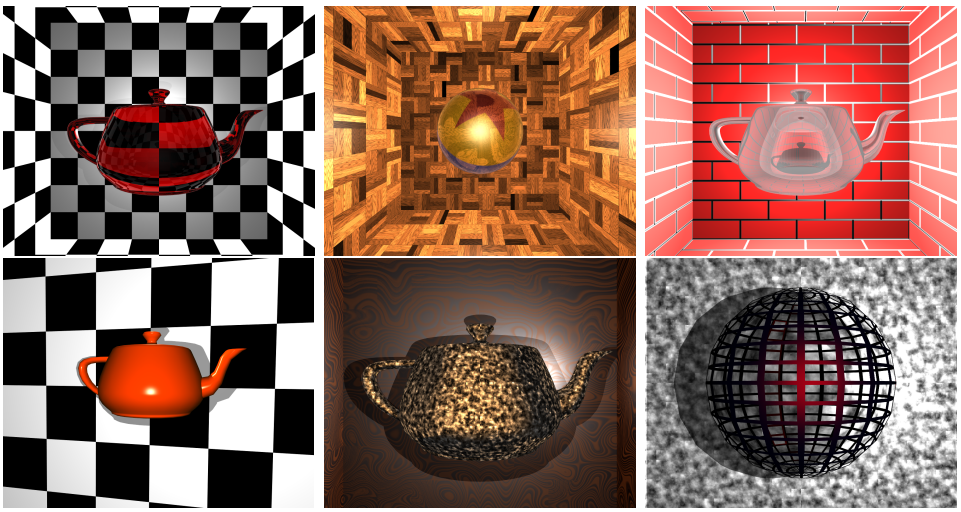

Figure 5.1: *Scenes rendered with our ray tracer: glass teapot on a checkerboard, refractive starball on parquet, whitted teapot on a brick wall, phong teapot, dented teapot in front of wood, metal screen in front of granite. All surface shaders are written in* scalar *C++ and compiled to a platform-independent intermediate representation. The renderer loads, specializes, optimizes, and packetizes the shaders at runtime. The packetized shaders outperform their scalar counterparts by an average factor of 3.6.*

This chapter presents a case study where automatic packetization is successfully applied to the shading system of an interactive ray tracer.

AnySL is a shading system that provides a poweful environment to shader programmers: shader code can be written scalar, using the language of choice, and independent of the target platform while matching the performance of hand-optimized, target-specific packet code.

We integrated AnySL into the shading system of the generic ray tracer *RTfact* [9].

5.1 Overview of the System

Our system uses LLVM to load, optimize, specialize, packetize, and link scalar shaders *at runtime.* First of all, we expose the functionality available to shaders (like shooting new rays, iterating over light sources, etc.) by a simple API that hides the internals of the renderer from the shader. Other renderers that provide an adapter for this API can reuse the shaders.

The shader itself is also wrapped by an adapter that implements wrapper functions to adapt to all packet sizes available in the renderer. For example, RTfact supports packet sizes of 1, 4, 16, 64, and 256. The shader itself is only compiled to a scalar version and to a packet version whose size matches the SIMD width of the architecture. Wrappers for other packet sizes are provided by the shader adapter.

Both adapters are not a part of the renderer binary but are separately available as bitcode. When the renderer wishes to instantiate a new shader, it loads the bitcode files of the shader and the adapters. All function calls between the adapters and the shader are removed by inlining. The remaining calls to the renderer core are linked. Possible calls to special functions (`sin`, `cos`, `noise`, `floor`, `ceil`, etc.) are inlined as well. If the user specifies constant parameter values, the shader is specialized by replacing the occurrences of the corresponding parameter variables with the concrete values.

Then, the complete machinery of LLVM optimizations is applied to the shader before the packetizer transforms the shader to packet code.

Finally, LLVM generates machine code for the packetized shader and returns function pointers (one per packet size) for the compiled code. The shader can then simply be invoked via a function pointer.

5.2 AnySL Infrastructure

In order to connect some shading language to the AnySL infrastructure, its front end needs to emit a bitcode file. In the examples in this section we use C++ and its LLVM frontend. Features typical to existing shading languages,

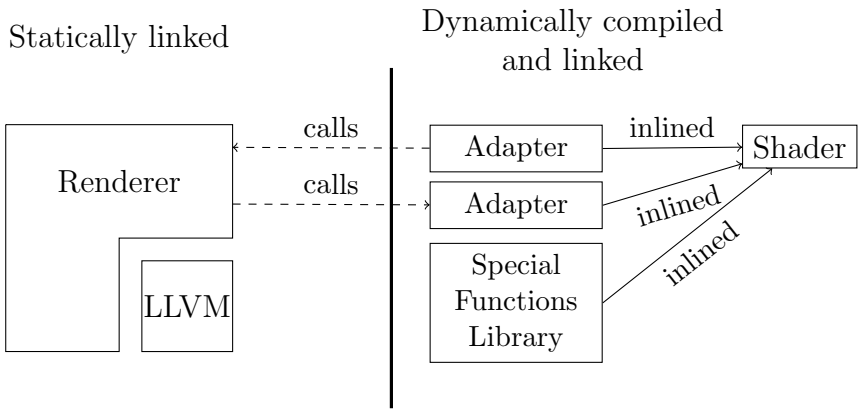

Figure 5.2: *Schematic overview of our system*

such as RSL's "`illuminance`" statement, have to be translated to calls to the renderer API which we describe below in more detail.

5.2.1 Shaders and the AnySL API

Since our system is shading language independent, we cannot directly use special language constructs (e.g. RSL's `illuminance` loop) or implicit globally available variables (e.g. RSL's `Ci` and `Co`). The functionality available to the shading languages is specified in a C++ header file, which defines a shading API in the form of functions, and provides the usual data types used in shaders – 3D vector, point, normal, and color, with appropriately overloaded operators.

In order to provide the appropriate "look and feel" to the shader writer, we also provide some common shading language features in the form of functions and macros. Utility functions, such as the RSL-compatible `SmoothStep`, or `FaceForward`, are directly implemented in the API header file. Some of the more advanced functionality, such as `SampleTexture` and `Noise`, is available as bitcode in our system. Functions, such as `TraceRay`, are forwarded to the rendering backend. These differences, however, are completely transparent to the shader writer. Figure 5.3 illustrates a shader written in C++ that uses our shading API.

5.2.2 Renderer API

The renderer API provides a small set of functions describing the required functionality from the renderer, such as tracing rays and querying light

```
#include "Shader.hpp"

PARAMETER float ringscale = 10.0f;
PARAMETER Color lightWood = Color(0.3f, 0.12f, 0.03f);
PARAMETER Color darkWood = Color(0.05f, 0.01f, 0.005f);

PARAMETER float Ka = 0.2f;
PARAMETER float Kd = 0.4f;
PARAMETER float Ks = 0.6f;
PARAMETER float roughness = 0.1f;

SURFACE_SHADER
{
  const Point P = origin + hitDistance * dir;
  const Vector & IN = dir;  /* normalized incident vector */
  const Normal & N = normal;  /* surface normal */

  /*
   * Compute the forward-facing normal NN and the vector
   * toward the ray orgigin V, both normalized.
   * These vectors are used by "specular" and "diffuse". */
  Normal NN = FaceForward(N, IN);
  Vector V = -IN;

  Point PP = P;
  PP += Noise(PP);

  /* compute radial distance r from PP to axis of "tree" */
  float r = Sqrt (PP.y * PP.y + PP.z * PP.z);

  /* map radial distance r into ring position [0, 1] */
  r *= ringscale;
  r += Abs (Noise(r));
  r -= Floor (r);

  /* use ring poisition r to select wood color */
  r = SmoothStep (0.f, 0.8f, r)
      - SmoothStep (0.83f, 1.0f, r);
  Color Ci = Mix(lightWood, darkWood, r);

  /* shade using r to vary shininess */
  Color C_diffuse(0.0f, 0.0f, 0.0f);
  Color C_specular(0.0f, 0.0f, 0.0f);
  float invRoughness = 1.0f / roughness;

  BEGIN_ILLUMINANCE_LOOP(P) {
    C_diffuse  += diffuseComponent(L_dir_norm, P, NN, Cl);
    C_specular += specularComponent(L_dir_norm,
                                    P, NN, V, Cl,
                                    roughness,
                                    invRoughness);
  } END_ILLUMINANCE_LOOP;

  result = Ci * (Ka + Kd  * C_diffuse)
         + (0.3f * r + 0.7f) * Ks * C_specular;
}
```

Figure 5.3: *An AnySL wood shader written in C++*

sources, as well as how shaders are invoked. The shading API hides all renderer-specific functions and data types from the shader. Hence, shaders in both source and bitcode representation are portable among all renderers that implement the renderer API.

5.3 Shader Instantiation

To instantiate a shader, the renderer is provided with the shader's name and a list of parameter name/value pairs. The renderer employs LLVM to load the corresponding bitcode file and make the code and data of the bitcode file accessible as *data structures* inside the renderer.

The instantiation starts by specializing the shader with respect to its parameters: Every load instruction in the shader's bitcode that loaded a global parameter variable is replaced by an instruction that directly produces the value the parameter was bound to. This does not only optimize away costly memory accesses but also exposes further optimization potential: variables in the shader code are replaced by their actual values. Depending on the shader, aggressive constant propagation and dead code elimination can remove large parts of the shader's code [12].

If the shader makes use of the AnySL API (see Section 5.2.1), the bitcode of the AnySL API adapter is loaded. All calls to the API are inlined into the shader. This removes one level of call indirection and allows for further optimization of the adapter in the context of the shader.

After specialization and adapter inlining, we perform several of LLVM's optimizations on the shader code, such as: Conditional constant propagation, conditional expression propagation, global value numbering, control-flow simplification, and dead code elimination. Especially constant and conditional propagation and dead code elimination prove to be very useful in improving the shader after specializing the parameters and inlining the adapter. If needed, the user can add further passes implemented in LLVM, or extend LLVM by its own optimizations.

The *scalar* shader is now optimized and ready to use. If the ray tracer is able to exploit SIMD architectures by shooting packets of rays, the packetizer generates optimized packet code for the shader. The performance boost obtained by automatic packetization is shown in the next section.

5.4 Results

In this section we discuss the performance of RTfact including the AnySL shading system with respect to rendering and compilation time. All experiments were conducted on a Core 2 Quad at a clockrate of 2.8GHz, the resolution was set to 512x512 pixels and all scenes include two point light sources.

We compare the rendering performance of automatically packetized shaders against scalar shading where packets are split and the scalar shader is executed sequentially. Most of the shaders we evaluate are very complex and would require a substantial amount of time to convert to a packet version by hand. The development and debugging of *scalar* shader code is already time consuming and difficult. Converting complex shaders to packet code by hand requires even more effort. This is mainly because the programmer has to implement all masking and blending code (see Sections 4.2 and 4.3) by hand.

Scene	Scalar (fps)	Packetized (fps)	Speedup
brick	3.7	14.9	4.0x
checker	4.1	18.3	4.5x
checker2	2.5	12.0	4.8x
glass	1.3	6.5	5.0x
glass2	0.73	4.2	5.7x
granite	2.8	2.9	1.0x
parquet	3.3	4.2	1.3x
phong	25.0	68.0	2.7x
screen	1.9	9.3	4.9x
starball	2.1	5.2	2.5x
starball2	0.19	0.56	3.0x
whitted	1.7	7.7	4.5x
whitted2	1.2	5.7	4.8x
wood	3.6	7.3	2.0x

Table 5.1: *Performance of our ray tracer in different scenes, measured in frames per second. Packetization is most effective for shaders with high computational density while it is ineffective for noise due to the indexed array accessing which requires scattered loads. The packetized shaders also profit from calls to the renderer API (shoot secondary ray, get light contribution) which are also available as packet code and use available mask information.*

Although still lacking some optimizations, the packetized versions of the

Shader	Scalar (ms)	Packetized (ms)
brick	470	550
checker	390	450
glass	540	500
granite	580	700
parquet	1180	1300
phong	360	500
screen	450	420
starball	430	550
whitted	500	530
wood	760	870

Table 5.2: *Compilation times of the just-in-time compiler in milliseconds. The shader compilation time is short enough to allow for runtime recompilation, e.g. for specialization of the shader code to interactively changed parameters.*

shaders already outperform their scalar counterparts by an average factor of 3.6 (see Table 5.1). Compute-intensive shaders and shaders that use the packetized API calls to the renderer can be significantly sped up. In combination with the amplified cache coherence that comes with the packet shading, we achieve speedups of factors up to 5.7.

Shaders using the noise function cannot profit as much from packetization. This is because Perlin noise [24] makes many accesses to a permutation table. Due to the lack of a scattered load in the SSE instruction set, these table accesses cause scattered loads and long sequences of instructions that break packets apart and reassemble them. Several inlined calls to the noise function also substantially increase the size of the shader code, which may exceed the instruction cache. This limits the performance gained by packetization for shaders that use the noise function.

The scalar shaders are loaded from unoptimized LLVM bitcode files at runtime. Before packetization, they are optimized using LLVM's internal passes and specialized into our shading system as described in Section 5.3. The whole procedure is efficient and allows for recompilation at runtime (see Table 5.1). This enables features like dynamic modification of shader parameters without sacrificing performance (see Section 7.3).

Although the speedup is quite good already, the code size growth of some shaders during packetization (Table 5.3) indicates a lot of untapped potential for further optimization.

Shader	LOC C++	LOC BC	LOC VEC
brick	75	403	829
checker	46	193	397
glass	118	317	486
granite	38	742	1230
parquet	151	2252	2686
phong	50	160	850
screen	55	214	389
starball	37	310	843
whitted	55	270	668
wood	57	1185	1712

Table 5.3: *Code size of 10 different AnySL shaders written in C++ (Lines of code, LOC), the corresponding scalar bitcode (LOC BC) and the packetized bitcode (LOC VEC). Note that the bitcode in contrast to the C++ code has all calls inlined. There is a clear connection between code size and packetization time visible. Compilation of large shaders that inline one or more calls to the noise-function (granite, parquet, wood) requires much more time (see Table 5.2). It is also clearly visible that the packetizer still has a lot of potential for optimization as the code size still grows too much for some of the shaders (e.g. phong, starball).*

Chapter 6
Conclusion

This thesis presented an algorithm for automatic packetization of scalar functions.

The system is implemented in a source-language and target-architecture independent intermediate language (LLVM), which makes it highly portable and easily extendable and provides a large and growing set of compiler optimizations.

A simple interface is provided that does not require any knowledge about a special language, data parallelism, or the target architecture: The user only has to implement a scalar function and declare a prototype for the corresponding packetized function. The implementation of this prototype is automatically generated by the packetizer.

A case study in the context of shader packetization for real-time ray tracing showcased the applicability of the system: Although it is still in an early stage and not optimized, the ray tracer already runs on average 3.6 times faster with the generated, packetized shaders than with the scalar source shaders on a standard SSE architecture of SIMD width 4.

Chapter 7
Future Work

The packetizer currently is in a "working prototype" state. Although the first results are very good already, there is plenty of tasks left to be done. This section describes possible directions for future work.

7.1 Data Types

Support for structs is still immature and does not allow any nesting. However, the task is not too difficult as it only requires recursive functions for the generation of correct `extract` operations.

Much more interesting is to give the user a larger degree of freedom for packet data types, e.g. to chose *array-of-structs* layout instead of *struct-of-arrays*. One way of achieving this goal is to generate a mapping out of the specified data types of the scalar and packetized functions' arguments. The system then automatically tries to apply this mapping to all data types and build the appropriate transformation rules that are currently fixed (Section 4).

7.2 Code Generation

This section will detail the addition of missing functionality, interesting features, and optimizations for the packetizer.

Recursive Packetization An important addition is to implement support for recursion. This includes supporting packetization of recursive functions on one side and recursive packetization on the other.

Packetization of recursive functions is only a small modification that adds the currently packetized function to the included native functions. This way, a recursive call to the function can be packetized without any further modifications.

Recursive packetization could be interesting if inlining of functions is not enforced but coupled to a heuristic that takes into account code size (which is already available in LLVM). The system would recursively packetize those functions and add another parameter for a mask. This way, the packetized function could include calls without sacrificing performance by splitting packets or losing the advantages of masked execution.

Exploiting Homogeneous Masks One very obvious optimization is to exploit cases where the condition of a branch decision is entirely `true` or `false`. Generating jumps around the specific code segments can result in large performance gains as possibly very complex code is skipped at the cost of only a comparison and a jump [28].

It would also be possible to only generate these jumps for paths that exceed a certain length or computational complexity.

Conditional Load/Store The packetizer currently explicitly computes masks and stores them as usual values. However, special hardware architectures provide support for predicated execution, i.e. each instruction has a mask associated and blending is performed automatically on a per-instruction basis.

In order to support such architectures, the mask generation pass has to be abstracted and provide different interfaces to the user.

Parallel Path Linearization The CFG linearization pass currently randomly choses what path to execute first if a branch is encountered. Employing a special heuristic instead is likely to improve the generated code. Such a heuristic can e.g. take into account the code size or register pressure of the different paths.

Additional Native Functions Adding more native functions is a natural extension for the packetizer: These functions can be implemented as efficient as possible and tuned by hand while at the same time providing the functionality independent of the target-architecture. If desired, it is still possible to include additional, tailored versions of native functions for specific architectures. Important examples include mathematic functions such as *cross product*, *dot product*, or *vector normalization*. Adding *noise* as a native function is especially interesting for shading languages.

7.3 Ray Tracing & Shading

The case study demonstrated the applicability of the packetizer to shading (Chapter 5). In the long term, we are aiming at packetizing a complete scalar ray tracer.

Additionally, we would like to extend the approach to include support for ray tracing on GPUs as well as targeting Larrabee [27] as soon as possible. AVX [15] and the Cell Broadband Engine [14] are two other interesting target architectures that we might consider in the future.

In the context of shading, there are also a few interesting follow-ups possible. Interactively changing shader parameters using sliders is only a small step ahead with the packetizer already being integrated into RTfact and RTSG and all functionality basically being ready.

A more complex topic could be to enhance the system to provide functionality for interactive shader debugging [30]. This could include functionality that lets a shader programmer run the renderer and then interactively select a specific pixel or area of pixels for investigation. The system could respond by e.g. displaying information about the executed shader, geometry properties, lighting information or neighboring pixels/geometry areas. One could even think about including parameters of the ray, highlighting paths through an acceleration structure, or recursively following sampling directions to light sources.

Acknowledgement

First of all I would like to thank my supervisor Prof. Sebastian Hack for his support, his help, and his ideas throughout all phases of this thesis and especially for the great working atmosphere in his group.

I also want to thank Prof. Philipp Slusallek, Dmitri Rubinstein and Iliyan Georgiev of the Computer Graphics chair for all the discussions that initially led to this thesis and the effort they put into writing the paper.

Simon Moll deserves special thanks for his commitment during implementation of the connection of the packetizer to RTfact.

Finally but most importantly I want to thank my girlfriend and my family for their support and understanding.

Bibliography

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM.

[2] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987.

[3] AMD-ATI. *ATI CTM Guide*, November 2006.

[4] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 102–111, New York, NY, USA, 1993. ACM.

[5] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The REYES Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 1987)*, pages 95–102, July 1987.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[7] Jeanne Ferrante and Mary Mace. On linearizing parallel code. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1985. ACM.

[8] Nicolas Fritz, Philipp Lucas, and Philipp Slusallek. CGiS, a New Language for Data-Parallel GPU Programming. In Bernd Girod, Hans-Peter

Seidel, and Marcus Magnor, editors, *Vision, Modelling, and Visualization 2004 (VMV), November 16-18, Stanford (CA), USA*, pages 241–248, 2004.

[9] Iliyan Georgiev and Philipp Slusallek. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, pages 115–122, August 2008.

[10] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture. *Intel Technology Journal*, 11(04), November 2007.

[11] Larry Gritz and James K. Hahn. BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools*, 1:29–47, 1996.

[12] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing Shaders. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 343–350, New York, NY, USA, 1995. ACM.

[13] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 24(4):289–298, August 1990. ISBN: 0-201-50933-4.

[14] IBM. *The Cell Broadband Engine*, 2005.

[15] Intel. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, 2008.

[16] Tal Lancaster. Pixar's PhotoRealistic RenderMan version 13. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 8, New York, NY, USA, 2006. ACM.

[17] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, 2000.

[18] William R. Mark, R. Steven, Glanville Kurt, Akeley Mark, and J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics*, 22:896–907, 2003.

[19] Michael D. McCool, Zheng Qin, and Tiberu S. Popa. Shader Metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[20] NVIDIA. *CUDA Programming Guide*, 2009.

[21] Joseph C. H. Park and Mike Schlansker. On Predicated Execution, 1991.

[22] Steven Parker, Solomon Boulos, James Bigler, and Austin Robison. RTSL: A Ray Tracing Shading Language. *IEEE Symposium on Interactive Ray Tracing*, 2007.

[23] Craig Peeper and Jason L. Mitchell. Introduction to the DirectX 9 High-Level Shader Language, 2003.

[24] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[25] Pixar. *The RenderMan Interface*. San Rafael, September 1989.

[26] Randi J. Rost, John M. Kessenich, and Barthold Lichtenbelt. *OpenGL Shading Language*. Addison Wesley, 2004.

[27] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[28] Jaewook Shin. Introducing Control Flow into Vectorized Code. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[30] M. Strengert, T. Klein, and T. Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 81–88. Eurographics Association, 2007.