Master Thesis

Variants in Whole-Function Vectorization

by Simon Moll

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

UNIVERSITÄT DES SAARLANDES

This page intentionally left blank

Supervisor:

Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany

Reviewers:

Prof. Dr. Sebastian Hack, Universität des Saarlandes,
Saarbrücken, Germany

Prof. Dr. Dr. h.c. Reinhard Wilhelm, Universität des Saarlandes,
Saarbrücken, Germany

Thesis submitted:

February 14, 2014

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,
Feburary 14, 2014

____
(signature)

Abstract

Whole-function vectorization transforms functions to make them operate on packs of argument bindings at once. The vectorized functions use nowadays common SIMD instructions to process the grouped threads in lock step where possible. Divergence in control-flow and side effects requires additional code in the vectorized function, which is a major performance issue. We developed a method that synthesizes dynamic tests for optimizable patterns given a specification. Our method is based on inductive program synthesis using SMT solvers and finds linear constraints over integers with arbitrary boolean structure. When applied to OpenCL kernels, we observe speed ups for control-flow optimizations up to 48% and up to 276% for memory access optimizations.

Contents

1 Introduction **3**

2 Background on Vectorization **8**
- 2.1 Low-Level Virtual Machine 8
- 2.2 OpenCL 8
- 2.3 Whole-Function Vectorization 10
- 2.4 Vectorization Analysis 11
  - 2.4.1 Consecutive 12
  - 2.4.2 Uniform 12
  - 2.4.3 Divergent 12
  - 2.4.4 Aligned 13
- 2.5 OpenCL Implementation 13

3 SAT modulo theory **15**
- 3.1 DNF over Linear Constraints 15
- 3.2 Background Theory 16
  - 3.2.1 Presburger Arithmetic 16
- 3.3 Complexity 16
- 3.4 Notation 17

4 Related Work **18**

5 Synthesis Procedure **21**
- 5.1 Overview 21
  - 5.1.1 Stages 21
  - 5.1.2 Problem Formulation 23
- 5.2 Constraint Model 24
- 5.3 Literal Discovery 25
  - 5.3.1 Edges 25
  - 5.3.2 Edge Cover Synthesis 26
  - 5.3.3 Approximative Edge Cover 26
  - 5.3.4 Algorithm overview 27
  - 5.3.5 Counter-example Query 27
  - 5.3.6 Edge Set proposal 28

5.3.7 Modulus Edge Proposal 30
5.4 Boolean Structure Synthesis 32
5.4.1 Clause Construction 32
5.4.2 Clause Refinement 33
5.5 Complexity 34
5.5.1 Literal discovery 34
5.5.2 Clause construction 34
5.6 Limitations 34
5.6.1 Over-approximative Line Cover 34
5.6.2 Modulus Constraints 35

6 Implementation 36
6.1 Variants 37
6.1.1 Complete Mask 37
6.1.2 Consecutive Memory Access 37
6.2 Mapping LLVM to the Problem Domain 37
6.2.1 Type and instruction mapping. 37
6.2.2 Vectorizing the expression tree. 38
6.3 Efficient pre-condition testing in loops 40
6.3.1 Loop-invariant Pre-condition 41
6.3.2 Optimizing the Loop Order 41
6.4 Synthesizing the Variant Test 44

7 Evaluation 45
7.1 OpenCL Performance 45
7.1.1 Complete Mask 46
7.1.2 Case: `DwtHaar1D` Benchmark 46
7.1.3 Case: `PrefixSum` Benchmark 47
7.1.4 Optimized Memory Access 49

8 Conclusions 51
8.1 Synthesis Procedure 51
8.2 Implementation 52

9 Future Work 53
9.1 Algorithmic Improvements 53
9.2 Super Vectorization 54
9.3 Improved Modulus Constraint Detection 54

Bibliography 55

Appendix A Benchmark kernels 58

Chapter 1

Introduction

With frameworks such as OpenCL[1] or CUDA[2] the data-parallel programming paradigm is a common occurrence in todays computing landscape. In this paradigm a function, the kernel, is applied to an array of data items. Usually, drivers implement these APIs for CPUs using Single-Instruction-Multiple-Data (SIMD) instruction sets. The instructions of SIMD-instruction sets operate on vectors of data. A scalar function processes a single item in one go whereas vectorized kernels operate on times the vector width many items at once. If the original kernel is straight line code, we can expect a speed-up in the order of the number of SIMD-lanes. Unfortunately, this transformation is non-trivial. SIMD-instructions are not directly applicable in the translation if control-flow divergence has to be expected. Divergence is dealt with by means of predication that requires additional code.

Limitations of Whole-Function Vectorization Whole-Function Vectorizers strive to be the answer to this problem. They automatically vectorize a kernel function given a set of SIMD-instructions. Vectorizers rely on static analysis to restrict on the locations where divergence can occur. During vectorization analysis, the vectorizer checks which values have optimizable properties and annotates them accordingly. The vectorizer knows efficient SIMD code patterns that can be used if their respective conditions apply.

Consider the fragment of a scalar kernel in figure 1.1. We are going to vectorize this example to SIMD-vectors of width 4. The code will be vectorized differently depending on what is known about the program variables. the `item_id` the id of the item processed by this thread. In the vectorized fragments, we will assume that the variable `c` will remain scalar.

So, without any additional knowledge the vectorized will transform the fragment into the code shown in Figure 1.2. The simple scalar load has be-

[1] [Khronos OpenCL Working Group, 2008]
[2] [CUD, 2007]

```
float r;
if (item_id < c) {
  r = A[item_id]
}
```

Figure 1.1: Scalar kernel fragment

come a cascade of scalar loads in the transformation. It may seem surprising at first that the memory access `z=A[item_id]` is expanded in this way.

The OpenCL driver will group work items into SIMD-groups that pass through the vectorized kernel. By construction, when stepping from one SIMD-lane to the next the `item_id` increases by one. However, the load is conditional. The vectorizer defaults and evaluates the condition for all SIMD-lanes separately. The result is the vector `m` that we refer to as the mask. The target machine of our example does not support predicated memory access. The mask test and the load are replicated for each lane.

```
bool4 m = int4(item_id) + int4(0,1,2,3) < int4(c)
float4 r;
if (m.x) {
  r.x = A[item_id]
}
if (m.y) {
  r.y = A[item_id + 1]
}
if (m.z) {
  r.z = A[item_id + 2]
}
if (m.w) {
  r.w = A[item_id + 3]
}
```

Figure 1.2: Naively vectorized

The situation improves if the vectorizer is told that the grouped items behave identical. That is, the mask is always `true` or `false` for all SIMD-lanes. Figure 1.3 shows how a vector sized load can be used in that case.

```
float4 r = load_float4(A,item_id)
```

Figure 1.3: Optimized fragment assuming that `item_id` $<$ `c`

The fragment in Figure 1.3 is more efficient than the naive code of Figure 1.2. Also, it can not be used in the general case. For example, if $c = 2$ and `item_id`$.x = 0$ the mask m will look as follows

$$m = (0011)$$

Without additional knowledge on c the vectorizer will use the safe default

code. Sometimes the vectorizer can find out that all values occurring in a comparison are aligned. Then, the vectorizer concludes that the condition will evaluate identical for all SIMD-lanes. It does so by a set of inference rules in an analysis stage called Vectorization Analysis.

Limitations of Vectorization Analysis The Vectorization Analysis checks if an optimizable condition holds for all program states. In our guiding example, the analysis checks if the mask register is always true or false

$$\forall state. m(state) = (1111) \lor m(state) = (0000)$$

The Vectorization Analysis applies this kind of reasoning to a variety of optimizable patterns.

$$\forall state. optimizable_condition(state)$$

However, if there is a single trace that fails the condition, vectorization analysis will default. This is an inherent limitation with how vectorization analysis works. This work tries to improve over this state of vectorization analysis. To alleviate this problem we ask instead if there is a *simple* condition under which *optimizable_condition* holds.

$$\forall state. test(state) \implies condition(state)$$

Using this condition *test* we can now sort out SIMD-groups that follow the optimizable pattern. The remaining traces are dispatched to the default code path. Applied to our scalar fragment of Figure 1.1, we can synthesize the pre-condition

$$test(\texttt{item_id}, c) = \texttt{item_id} + 3 < c$$

and obtain the optimized code path in Figure 1.4.

Motivation In our application, the `optimizable_condition` specifies the pre-condition for an optimizeable code path.

There is the question, why it does not suffice to test this specification directly. This has several reasons.

- *test* may be cheaper to compute than *spec*.
- *test* has a simpler structure and may expose optimization opportunities to the compiler.
- In the setting of vectorization, the specification often uses values from the vectorized program. If we evaluated the specification directly we would need to set-up the necessary support structure (mask predicates, ...). However, the purpose of the specification is to check if execution

```
float4 r
if (item_id + 3 < c) {
  r = load_float4(A,item_id)
} else {
  bool4 m = int4(item_id) + int4(0,1,2,3) < int4(c)
  if (m.x) {
    r.x = A[item_id]
  }
  if (m.y) {
    r.y = A[item_id + 1]
  }
  if (m.z) {
    r.z = A[item_id + 2]
  }
  if (m.w) {
    r.w = A[item_id + 3]
  }
}
```

Figure 1.4: Specialized path with a pre-condition test.

can take a path that does not need these support structures. Except for the optimized path, this makes direct testing little better than naive vectorization.

On the other hand, we restrict *test* to only use uniform values which do not introduce vectorization overhead. This solves the problem, as the support structures will only be set-up if the condition fails.

- We do not pursue this idea yet, but *test* is simple enough to quantify how often the optimized path will be taken.
- We will see that the structure of *test* is ideal for branch prediction. So, if *test* can be generated, it may be wise to introduce an optimized path, even if it will only be taken in a small fraction of the cases.

Note that, even though we use our method in the context of Whole-Function Vectorization, the algorithm may be applicable in other domains as well.

The thesis is layed out as follows. We will discuss the foundations of our synthesis procedure in the Background Chapters 2 and 3. In Chapter 5, we are going to introduce our method for inferring linear predicates over $\mathbb{Z}^n$ from a given specification. How this was applied in the context of Whole-Function Vectorization will be detailed in Chapter 6. Finally, we will discuss results on real-world benchmarks in Chapter 7.

Contributions

- We show a novel approach for optimizing control-flow vectorized kernel functions that significantly improves runtime performance.

- We conceived a synthesis procedure for boolean predicates of arbitrary boolean structure.
- Our Variant Analysis pass automatically maps program code in the SMT domain.

Chapter 2

Background on Vectorization

2.1 Low-Level Virtual Machine

The Low-Level Virtual Machine (LLVM) is a compiler framework by [Lattner and Adve, 2004]. In its core, LLVM consists of a program representation called LLVM-IR and the tools for manipulating it. In LLVM-IR functions are given as Control-Flow Graphs and data-flow has SSA-form. Backends provide the logic for transforming LLVM-IR to actual hardware machine code. While being simpler than a high-level language, LLVM-IR is abstract enough to map to a variety of architectures (x64, ARM, PowerPC). Frontends translate programs from other languages to LLVM-IR. The functional unit for program analyses and transformations is a *pass*. By operating on the common representation passes are, once implemented, accessible by a broad audience. The LLVM framework has support for the SSE, AVX SIMD instruction sets.

2.2 OpenCL

OpenCL [Khronos OpenCL Working Group, 2008] is a framework and programming language for processing highly parallel computing problems. The OpenCL API abstracts away from concrete devices and specific architectures to an unified framework. The user specifies a kernel function in the programming language and attaches memory buffers through the API. In OpenCL each parallel problem instance is called a work item. All work items execute on the same set of memory buffers for input and output. The work items are coordinated on a 1,2 or 3-dimensional global grid. This grid is again subdivided into local *work groups* of equal size. This structure can be seen in figure 6.1a. Threads can access their coordinate in the global (*get_global_id*) and local (*get_local_id*) grid.

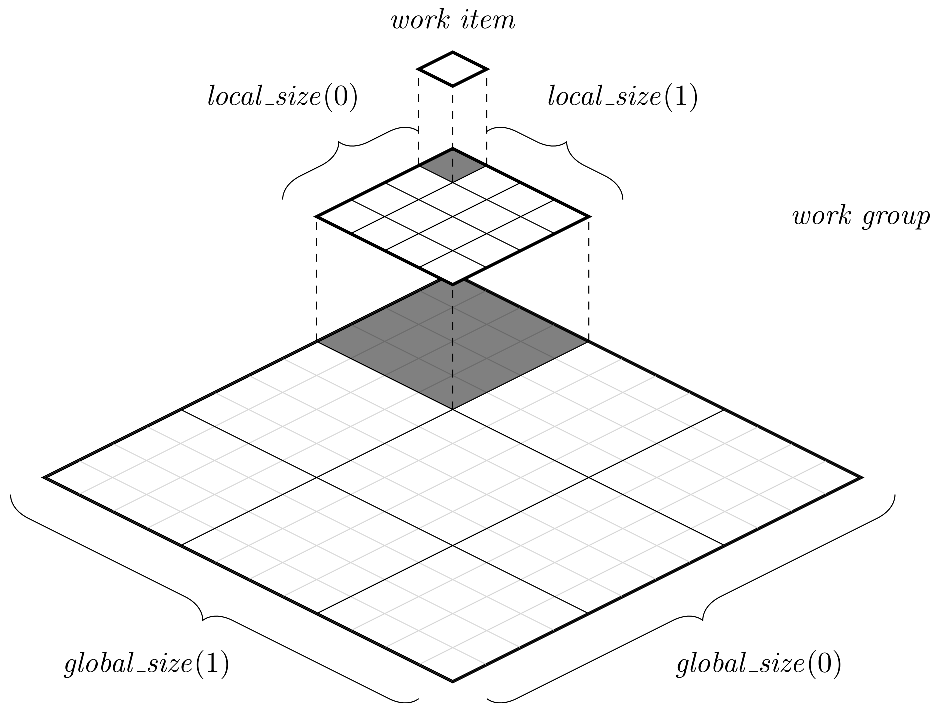


Figure 2.1: OpenCL Programming Model

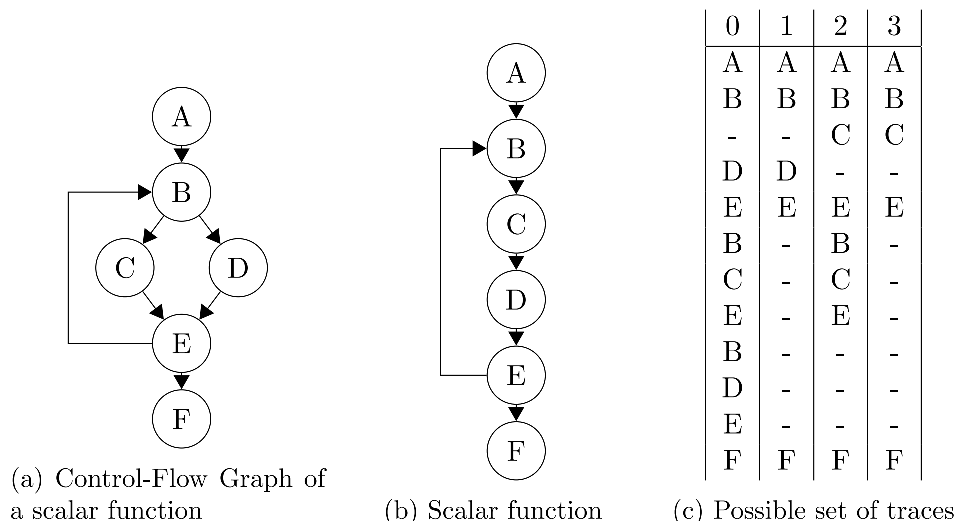
Global scope The global grid is used to specify the total problem size. The input and output buffers are located in the so-called *global memory*. These buffers can be read and written by the user and the kernel function. Global memory is thought of to be slow and plenty to hold all problem data.

Work Group On the other hand, work groups are actual units of execution. Work items of the same work group share access to buffers of *local memory*. This memory is only persistent while a work group is executing. Local memory is usually spare and fast which makes it popular for caching data (for example from global memory).

Synchronization During normal operation, OpenCL does not assume any synchronization across work items even of the same work group. However, an executing work group synchronizes its work items at calls to the

intrinsic *barrier* function (also *memfence*). Execution will not continue beyond the barrier until all work items have reached it. Barriers establish an order between memory operations before and after the synchronization point. This is used to exchange data between work items and in particular frequent with local memory.

2.3 Whole-Function Vectorization



0	1	2	3
A	A	A	A
B	B	B	B
-	-	C	C
D	D	-	-
E	E	E	E
B	-	B	-
C	-	C	-
E	-	E	-
B	-	-	-
D	-	-	-
E	-	-	-
F	F	F	F

(a) Control-Flow Graph of a scalar function (b) Scalar function (c) Possible set of traces

Figure 2.2: Diverging traces in a vectorized function.

Our work is based on the Whole-Function Vectorizer of [Karrenberg, 2014]. The vectorizer operates on LLVM Bitcode. Whole-function vectorizer modify a function, such that it is evaluated for n calls in a single execution. In the original program scalar instructions operate on a single thread. The vectorized program operates on a SIMD-group of threads instead. During vectorization, the scalar instructions are replaced by their respective SIMD-variants. The SIMD-instructions operate on vector registers of size n. We refer to every component of these registers as a lane. Using SIMD-instructions the vectorized program processes the original threads in lock-step fashion. However, care must be taken in the presence of control-flow. If a single thread branches out all other threads have to go through its target block as well. This deviation from their actual trace must not alter their state. To this end, every thread signals through a mask bit if it will actively execute the next block. There is two different ways to implement masking: Instructions without side-effects (arithmetic), can be executed independently of the mask. The program only checks at the end of the block if the computed values should replace the old state or not. This is called a *blend*. As this operation transforms control-flow (the former branch) to

data-flow (blend on branch predicate) the resulting program is linearized. Take for example the scalar program in figure 2.2a. For every set of n traces through the scalar function, there must be a trace through the vectorized program. So, it gets linearized as in figure 2.2b. Figure 2.2c shows a schedule for a SIMD-group of size 4. The letters refer to active participation in the execution of the block of that name (the mask bit is set). Otherwise, the mask bit is off and the thread passes the block without any effects on its state.

Linearized programs may be easier for out-of-order CPUs to schedule ahead. However, each disabled thread adds a dead-lane reducing the advantage of SIMD-execution. In a binary branch, a thread will only participate in the execution of one of the targets. For the other block, the execution of this thread will be entirely in vain. This is different for instructions with side-effects (memory operations,unknown function calls). They must only be executed if the mask is set. This means if there is no SIMD-variant of the instruction or function (that would take a predicate), the instruction will not actually be vectorized but replicated. The predicate will be a branch testing the flag. So in this case, the vectorized is doing little more than replicating the code n times.

2.4 Vectorization Analysis

```
kernel void FooBar(int c, global float * data)
{
  float x, y;
  ...
  if (c > 5) {
    *a = add_float(x,y);
  }
}

kernel void FooBar_SIMD_4(int c, global float * data)
{
  float4 x, y;
  ...
  if (c > 5) {
    *a = add_float4(x,y);
  }
}
```

Figure 2.3: Uniform branch that is preserved in the vectorized program.

In naive vectorization, control-flow is always linearized and executed with a mask predicate. Sometimes the result of an instruction may not depend on the threads of the SIMD-group, a property which we call *uniform*. This happens also if the instruction computes a branch predicate. In that case all threads will branch to the same block and the other can be skipped

(see figure 2.3). This coins the notion of an uniform branch. As in the example, the computations controlled by uniform branch do not need to be uniform themselves. Nevertheless, the branch can be preserved in the program (exceptions apply for loops).

The vectorizer determines these and other properties during the Vectorization Analysis stage. We introduce the following non-complete list of variable annotations from [Karrenberg, 2014]. $v(i)$ denotes the valuation of variable v at SIMD-lane i when processing a range of threads. The SIMD-group has size *width*. Variant Analysis considers these properties for vectors of integers, pointers or reals. In the scope of this thesis, we restrict the definitions to integer vectors. That covers pointers as offsets into data buffers as well. So, v is an integer vector that stands for the valuation of a variable across all threads in a SIMD-group.

2.4.1 Consecutive

We call v consecutive if it increases by one when going from item i to item $i + 1$.

$$\forall_{i,j \in \{0,..,width-1\}} v(i + j) = v(i) + j$$

Advantage If v is a consecutive array offset, memory accesses can be realized by vector stores or loads. Extracting or inserting an element into the transfered vector is then equivalent to individual transfers with offset pointers.

2.4.2 Uniform

We say v is uniform if the value is the same for all lanes.

$$\forall_{i,j \in \{0,..,width-1\}} v(i * width + j) = v(i * width) + j$$

Advantage If a pointer is uniform, all threads accessing it will see or write the same result. Uniform memory instructions can remain scalar (we assume there are no data races in writes). If a boolean value is uniform, any branch it controls can remain in the vectorized program.

2.4.3 Divergent

We call a value v divergent, if it is neither uniform or consecutive. Vectorization Analysis resorts to *divergent* as the weakest annotation meaning no relation between the lanes could be found.

2.4.4 Aligned

Finally, v is aligned, if in every execution it holds that

$$\forall_{i \in \{0,..,width-1\}} v(i) \bmod width = 0$$

Advantage On some architectures (for example x86_64, Nehalem core), vector memory instructions must only access aligned pointers. On more recent architectures ((x86_64, Intel core_i7), non-aligned accesses are still slower.

2.5 OpenCL Implementation

We use the OpenCL implementation by [Karrenberg and Hack, 2011]. The driver translates OpenCL programs to LLVM IR. For the most part, the OpenCL execution model maps directly to the SIMD paradigm. The driver subdivides work group again in SIMD groups. Without synchronization points, it loops over the set of SIMD-groups processing each by a call to the vectorized kernel. This model finishes a SIMD-group before starting the next making it not directly applicable to synchronized kernels.

Synchronization Consider, we are dealing with a synchronizing kernel as in figure 2.4a. To cope with synchronization, the kernel function is split at the synchronization points. Each barrier-free segment gets a loop over all work items. Finally, the segment loops are merged back into the kernel function. The resulting function operates on a work group range of items and respects synchronization points (see 2.4b).

Vectorizing Kernels The vectorizer is applied to the barrier-free segments. The driver chooses a vectorization dimension and processes consecutive items in that dimension in SIMD-groups. This makes calls to *get_global_id* / *get_local_id* consecutive in that dimension. The common structured values of OpenCL are listed in table 2.1.

Value	Structure
$get_global_id(d)$	Consecutive ($d = 0$), Uniform otherwise
$get_local_id(d)$	Consecutive ($d = 0$), Uniform otherwise
$get_*_size(d)$	Uniform
$get_group_id(d)$	Uniform
Argument	Uniform

Table 2.1: Structured values in vectorized OpenCL kernels

```
__kernel void
myKernel
    (int uni, ...)
{
  int x = get_global_id(0);
  A(x,uni, ...);
  barrier();
  if (uni == 0) {
    B(x,uni,...);
    barrier();
  }
  C(x,uni,...);
}
```

(a) Original OpenCL kernel

```
void
OpenCL_Kernel
    (int uni,...)
{
  for (int x = 0; x < local_size(0); ++x) {
    A(x, uni, ...);
  }

  if (uni == 0) {
    for (int x = 0; x < local_size(0); ++x) {
      B(x, uni, ...);
    }
  }

  for (int x = 0; x < local_size(0); ++x) {
    C(x, uni, ...);
  }
}
```

(b) Transformed kernel ready for execution

Figure 2.4: Implementing synchronization points

Chapter 3

SAT modulo theory

Extensions for enabling variant generation Our extended OpenCL driver adds two additional intrinsics. *first_global_id* and *first_local_id* return the thread coordinate of the first lane in the SIMD-pack. With this simple trick the default vectorization analysis will discover uniform branches. Modifications to the vectorizer itself were not necessary.

3.1 DNF over Linear Constraints

$$(x_0 \wedge x_1 \wedge \overline{x_2}) \vee (\overline{x_1} \wedge y_0)$$

Figure 3.1: A formula in Disjunctive Normal Form

A boolean formula is in Disjunctive Normal Form (DNF), if it satisfies the following structural properties:

- Negation only occurs in literals.
- Literals are only used directly in conjunctions ($\wedge$).
- Disjunctions ($\vee$) can only join conjunctions or single literals.

A conjunction over possible negated literals is called a *clause*. A DNF-formula is finally a disjunction over clauses. Direct conversion of an arbitrary boolean formula to DNF may expand the formula to exponential size. Recent methods improve little over this result (see, for example [Miltersen et al., 2003]).

A valuation assigns boolean values (values from the set $\mathbb{B}$) to the literals in the formula. A valuation is called an*interpretation* of the formula. A formula is called *feasible*, if there is a satisfying valuation to its literals.

The satisfiability problem (SAT) is concerned with finding a satisfying interpretation for a given boolean formula. SAT-solver check if a boolean formula can be satisfied and has a solution. We use SAT-solvers that furthermore will return a valid solution, if it exists.

3.2 Background Theory

SMT (SAT modulo theory) replaces the boolean literals in SAT by parametric functions. The set of functions is given by a *background theory* and can have arguments of an arbitrary type. The problem generalizes to finding an interpretation of the variables that satisfies the underlying theory. An example of a formula over arithmetic is shown in figure 3.2. In our setting we will encounter arithmetic operations over integer variables.

$$\big(x_0 > 0 \land x_1 \bmod 5 = 3 \land \overline{x_2 + 3 < 2}\big) \lor \big(\overline{x_1 > 7} \land y_0 < 0\big)$$

Figure 3.2: A formula with integer arithmetic

However, for arbitrary predicates no general decision procedure exists and it was even shown that the halting problem can be encoded in non-linear integer arithmetic.

3.2.1 Presburger Arithmetic

Presburger Arithmetic is a decideable fragment of integer arithmetic. Each variable and constant is an integer value and the only allowed arithmetic operator is addition. The arithmetic allows devisibility predicates (for example. $x|5$ means x is divisible by 5). Note, that multiplication by a constant can be expressed by repeated addition. Therefore, the synthesized terms are expression in Presburger Arithmetic.

3.3 Complexity

Finding a solution for SAT is already an NP-complete problem. Quantifier-Free Boolean Algebra over Presburger Arithmetic (QFBAPA) is still a NP-complete problem (see [Kuncak, 2007]). [Fischer et al., 1974] showed that solving quantified formulas over Presburger Arithmetic has double-exponential worst-case time complexity. Despite these theoretical limitations, there are decision procedures that are efficient on real problems.

3.4 Notation

SMT-formulas with quantifiers are considerably harder to check than those without. To this end, we use the symbols $\bigwedge$ and $\bigvee$ to denote quantification over known finite sets that will be instantiated for the query. For example the fragment $\bigwedge x \in \{a, b, c\}\,(x)$ will show up in the query as $a \wedge b \wedge c$. On the other hand, actual quantifier symbols $\forall$, $\exists$ will occur as such in the query. An example for this would be $\forall x \in \mathbb{Z}\,(p(x))$.

Chapter 4

Related Work

Variants The Whole-Function Vectorizer of [Karrenberg, 2014] can generate more efficient code, if its requirements are met in all executions. BOSCC by [Shin et al., 2009] are branches that skip parts of the linearized program if the mask is disabled for all lanes. In lack of a static heuristic, the decision where to test is based on profiling data and an instruction count cost model. Still, their results show that dynamic tests can improve program performance. They report that improvements where achieved for branches where a SIMD-group behaves quasi uniform.

[Karrenberg et al., 2013] use a SMT-solver to augment the vectorization analysis for consecutivenes. Their approach samples the entire set of program states that satisfy the optimizable condition. The exhaustive sampling is expensive and only possible if the predicate has finite support (they only consider values in the range $\{0, .., 2^{32} - 1\}$). For some instances`FastWalshTransform` their approach fares in the range of minutes making it unsuitable for real-world applications. Our approach will only sample on demand and works in the integer domain of infinite support. We supplement the synthesis procedure that complements their approach. For lack of a SMT solver with support for Quantified Non-Linear Integer Arithmetic, our synthesis procedure still requires manual reasoning for their benchmarks. However, we also apply our procedure also to optimizable control-flow patterns. For those benchmarks, our implementation finds the pre-conditions automatically starting from kernel code.

Vectorization analysis The Whole-Function Vectorizer by [Karrenberg, 2014] includes a static vectorization analysis pass based on pattern matching and fix-point iteration. It statically identifies *uniform*,*consecutive* and *divergent* memory accesses and *uniform* or *divergent* control-flow. We rely on these annotations to restrict the possible valuations of the program variables.

Synthesizing predicates [Srivastava, 2011] relies on fixed-size function templates. The synthesis procedure fills in the holes in the templates. The function templates are more expressive than the restricted set of constraints that we consider. However, different than in their work our predicates can grow arbitrary large.

Relation to Abstract Interpretation Pre-condition synthesis is related to the concept of abstract interpretation. Abstract interpretations over-approximate the set of possible valuations at a point in the program. Our approach differs in two aspects. Firstly, our synthesis technique generates an under-approximation. Secondly, the synthesized predicate only models program states that satisfy the specified pre-condition. Note, however, that the two can be reduced on another. Say, we inserted a branch that tests the specification directly. An abstract interpretation procedure can now over-approximate the set of states that will not take this branch. The inversion of that set is an under-approximation of the branch condition. The branch condition models the specification and so we get an under-approximation of the program states that will take the branch. Equivalently, we can create a specification that holds if the program variables are in a valid state. The synthesis procedure can generate an under-approximation of the inverse. The inversion of that finally models the feasible program states. There is a rich corpus of work on finding abstract interpretations of convex sets [Cousot and Halbwachs, 1978, Laviron and Logozzo, 2009, Brauer et al., 2013, Miné, 2006], to name a few. Note, that the synthesis procedure is more powerful because it can model specification with quantifiers.

Boolean Structure Generation Approaches of program synthesis are commonly restricted to conjunctions of base terms [Itzhaky et al., 2010]. This applies in particular to works in the domain of Abstract Interpretations. These include [Cousot and Halbwachs, 1978] [Laviron and Logozzo, 2009] [Brauer et al., 2013] [Miné, 2006]. Again, [Srivastava, 2011] is restricted to manually specified boolean formulas. [Dillig et al., 2013] can discover additional boolean templates from the inspected function. Our approach can synthesize DNF formulas given a finite and complete set of literals. [Kuncak et al., 2010] restrict the boolean structure to improve the problem complexity.

Our algorithm follows the approach proposed by [Itzhaky et al., 2010]. They model the function directly by iterative refinement using samples. Each sample is drawn to be counter example for the current candidate solution. If no such sample can be found, the solution is correct and the method terminates. They assume that the boolean structure is a conjunction of base terms. By virtue of edge sampling, our approach decomposes the problem in Literal Discovery and Boolean Structure Synthesis. We allow arbitrary

boolean structures (modeled by a finite DNF formula).

In the literature, we often find the paradigm of composing a function from a library of buildling blocks [Gulwani et al., 2011a, Itzhaky et al., 2010, Gulwani et al., 2011b]. The Literal Discovery stage of our algorithm can be interpreted as generating such a library. The Boolean Structure Synthesis stage only draws primitives from this library. This relates our work to this paradigm.

[Dillig et al., 2013] derive loop invariants from programs. Their approach uses a set of inference rules to infer the weakest pre-condition for a proposed variant.

[Brauer et al., 2013] conceived an approach for finding intervals using SAT-solvers. Their approach combines the abstract domaints of polyhedra and linear congruences. The integer values are restricted to a 16bit architecture and can not handle unbound domains.

[Paige and Koenig, 1982] introduced the notion of a derivative for blocks of code. We use the concept when specializing loops for our synthesized pre-condition.

[Barthe et al., 2013] use SMT-solvers to synthesize vectorized programs. Their work is orthogonal to ours. We already have a vectorizer and want to improve the performance of the generated code. We reckon that the approaches could be combined in an optimizing function vectorizer based on code synthesis.

[Simbürger et al., 2013] give an empirical study of how often polyhedral loops occur in a broad set of benchmarks. The applicability of our approach currently suffers from shortcomings in the vectorizer. Nevertheless, does this empircal study hint that there is a high potential for linear constraints in relevant benchmarks. [Brauer and King, 2012] infer octagons constraints for bounded integer values.

[Kuncak et al., 2010] synthesize straight-line programs from a specification. They use introspection to identify the theory used in the specification. Solutions are computed using witness computation techniques akin to quantifier eliminiation. The synthesized condition corresponds to the synthesized pre-condition in their model. Importantly, they assume that the precondition is explicitely stated in the specification. In constrast, we synthesize the condition exclusively relying on queries to the SMT solver.

[Colón et al., 2003] infer loop invariants. The infered invariants are polynomial inequalities over the real numbers. They understand their work as a fremwork that will become more efficient/relevant as solvers improve. In constrast, we are interested in efficiently synthesizing pre-conditions for a variant condition in the loop.

Chapter 5

Synthesis Procedure

Our method for pre-condition synthesis relies on a synthesis procedure over integer variables. The procedure is given a specification *spec* and synthesizes an efficient pre-condition test *model*. In our application, these specifications model the pre-conditions for optimized code paths. Before dealing with the actual implementation, we are going to introduce our method for predicate synthesis over linear constraints.

5.1 Overview

We do not look into the arithmetic structure of *spec* directly. Instead, our algorithm relies on repeated tests that check whether the current hypothesis on the structure of *spec* is correct. This is a process called Counter-Example Guided Inductive Synthesis (CEGIS). We carry out the tests as repeated queries to a SMT-solver.

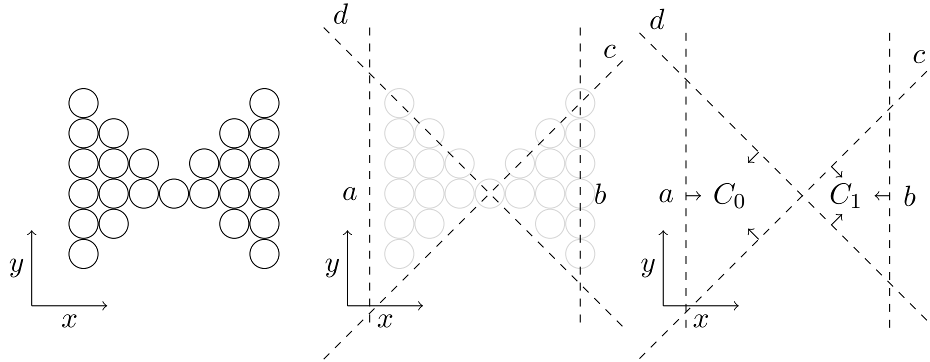
5.1.1 Stages

Our approach breaks up the problem of modelling the specification in two parts. Assume, we are given the speification in two variables x and y depicted in Figure 5.1a.

Each circle stands for a pair (x, y) where the specification evaluates to $true$.

Literal Discovery During the Literal Discovery, the procedure finds a set of literals that can model the specification. We do not care about the underlying boolean structure in this stage. For the example of Figure 5.1b we find the following set of border constraints

$$a(x, y) = x = 1$$
$$b(x, y) = x = 8$$
$$c(x, y) = y - x = 1$$
$$d(x, y) = x - y = 9$$



(a) The specification over two variables x and y. (b) Border lines a, b, c, d *spec* found by Literal Discovery (c) Final formula *model*

Figure 5.1: Stages of the synthesis procedure

They enclose the regions where the specification holds.

Boolean Structure Synthesis Boolean Structure Synthesis, takes these literals and constructs a DNF formula from them that approximates the specification. To this end, we replace the $=$ in the border lines with inequalities $>$ and let Boolean Structure Synthesis generate clauses from it. It will come up with the solution shown in Figure 5.1c where the clauses C_0 and C_1 are

$$C_0(x, y) = (x > 1) \wedge (x - y < 1) \wedge (y - x \leq 1)$$

and

$$C_1(x, y) = (x \leq 8) \wedge (x - y \geq 1) \wedge (y - x > 1)$$

Pipeline We summarize the complete synthesis procedure as follows. The procedure is given a boolean predicate over integer variables, the specification *spec*. It outputs a predicate *model* that under-approximates *spec*.

- Detect border lines *spec* and output a set of literals L.
- Synthesize a formula *model* drawing literals from L.
- Verify the solution *model*.

$$\underset{model}{\text{minimize}}$$

$$cost(model) \tag{5.1a}$$

$$\|\{\sigma \in \Sigma | spec(\sigma) \wedge \overline{model(\sigma)}\}\| \tag{5.1b}$$

$$\text{subject to}$$

$$\forall_{\sigma \in \Sigma}. model(\sigma) \implies spec(\sigma)$$

Figure 5.2: The multi-objective optimization problem approached by our work.

5.1.2 Problem Formulation

Let $V = \{0, .., m-1\}$ be an enumeration of m distinct program variables. The set of possible valuations is given by $\Sigma \subseteq \mathbb{Z}^m$. For sake of simplicity, we are going to assume that $\Sigma = \mathbb{Z}^m$. We write $\sigma \in \Sigma$ to refer to a state of the program variables.

The boolean predicate $spec \in \Sigma \to \mathbb{B}$ is a specification over the program variables.

We are trying to infer a predicate $model \in \Sigma \to \mathbb{B}$ that under-approximates the specification *spec*. This leads to the multi-objective optimization problem seen in Figure 5.2

The approximation should be inexpensive to compute (5.1a) and also be as precise as possible (5.1b).

In the following, we will not adhere strictly to the mathematical specification of the optimization problem. We will use it, however, to argue for the design decisions made in our system.

Chosen technique The optimization problem is undecidable for arbitrary theories. Unfortunately, even state-of-the-art SMT-solvers will only find single satisfying solutions to the specification *spec*. To this end, we follow the common approach of inductive program synthesis. Our algorithm, relies on repeated queries to a SMT-solver to refine the current best solution. Iterative improvement by counter-examples dates back to works of [Gold, 1967] and [Shapiro, 1983].

5.2 Constraint Model

We restrict our approach to a linear model. The model is made up out of linear inequalities of the form[1]. These formulas are expressive while still being decideable (QFBAPA theory).

$$l(\sigma) = (a_l(\sigma - b_l)) \bmod c_l > 0$$

a_l may only have components from $\{-1, 0, 1\}$. The models allows $3^m - 1$ distinct normals in the inequalities for m variables. Take figure 5.3 for an example.

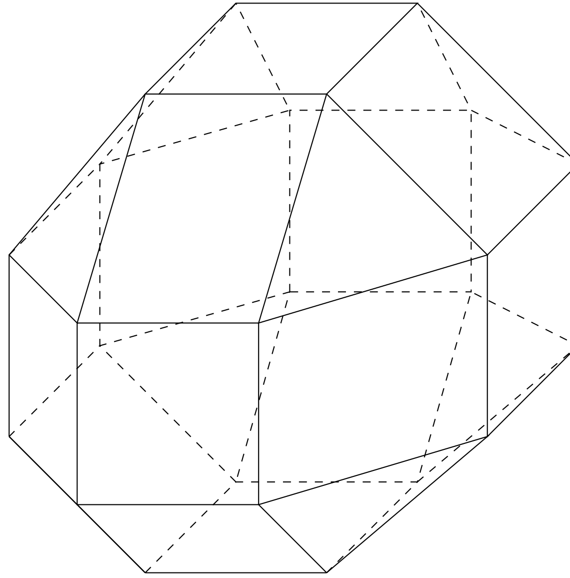


Figure 5.3: Visualization of the 26 possible normals for three variables.

The boolean structure of the predicate is encoded in in disjunctive normal form (DNF). So *model* is a disjunction over clauses c.

$$model(\sigma) = \bigvee_{c \in F} c(\sigma)$$

where each clause is a conjunction over literals.

$$c(\sigma) = \bigwedge_{l \in C} l(\sigma)$$

If all literals in c have infinite modulus, the clause can be interpreted as describing a polyhedron. A clause may describe multiple or even countable infinite polytopes, if a single literal uses modulus seen in Figure 5.4.

[1]We define $a \bmod \infty = a$ to allow constraints without modulus.

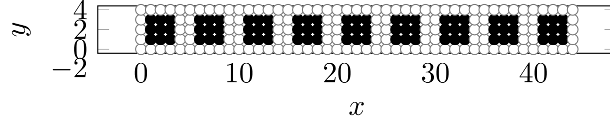


Figure 5.4: A clause with modulus describing countably infinite polytopes $t(x, y) = (x - 1) \bmod 5 < 3 \wedge y > 0 \wedge y < 4$.

5.3 Literal Discovery

The Literal Discovery stage finds a complete set of linear inequalities to describe the specified predicate. If *spec* flips in the neighborhood of a given value at least one underlying constraint flips as well. We call this an *edge* in the predicate. Literal Discovery covers all edges in *spec* with lines. If it succeeds, it turns the lines in linear inequalities for the Boolean Structure Synthesis stage. Before detailing the process of Literal Discovery, we are going toformally introduce the notion of an edge.

5.3.1 Edges

An edge of a predicate f in variable v is a value of σ such that up_v^f is *true*[2].

$$up_v^f(\sigma) = f(\sigma) \oplus f(\sigma[v \to v + 1])$$

There is an *up*-edge if the sign of the condition f flips when increasing v by one. We define the notion of a *down*-edge

$$down_v^f(\sigma) = f(\sigma) \oplus f(\sigma[v \to v - 1])$$

which corresponds to the notion that there is an edge when decreasing v by one. Now at each point σ the edge environment is given by the pair $(up_v^f(\sigma), down_v^f(\sigma))$. If the predicate is dropped as in up_v, we refer to the specification *spec* and mean up_v^{spec}.

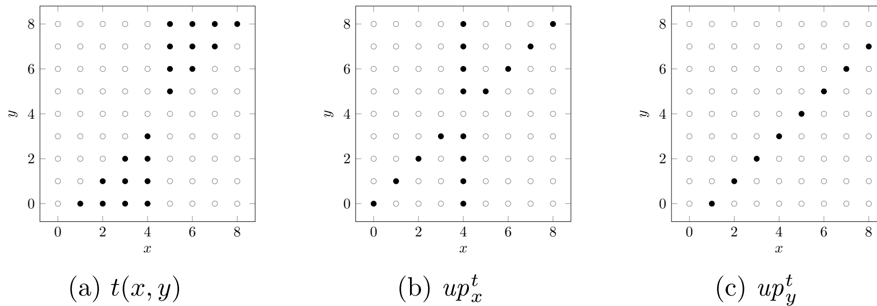


Figure 5.5: The predicate $t(x, y) = x > 4 \oplus y < x$ and the edge sets for x and y.

[2] $\oplus$ denotes exclusive-OR

Figure 5.5 shows the edge environments of a simple predicate. Note, that up_y does not reflect the constraint $x > 4$ because it does not use y.

We observe that a perfect solution f for *spec* will satisfy

$$\forall_{\sigma\in\Sigma,v\in V}\left(up_v(\sigma)\Leftrightarrow up_v^{model}(\sigma)\right)$$

5.3.2 Edge Cover Synthesis

Literal discovery does not make any assumptions on the underlying boolean structure of *spec*. To this end, it synthesizes an overapproximative model of the edges in *spec*. Technically, literal discovery synthesizes functions f_v such that

$$\forall_{\sigma\in\Sigma,v\in V}\left(up_v(\sigma)\implies f_v(\sigma)\right)$$

where the functions f_v describe sets of lines

$$f_v(\sigma\in\Sigma)=\bigvee_{l\in E}\left(a_l(\sigma-b_l)\bmod c_l=0\right)$$

that cover all the edges of *spec*. By prohibiting zero vectors for a_l, f_v can not be set to a trivial solution (constant *true* or *false*).

Relation to the original predicate The synthesized functions f_v are in fact a model for the edges of a different predicate. Assuming that *spec* was known, we could replace its boolean structure by $\oplus$.

$$spec^{\oplus}(\sigma)=\oplus_{C\in F}\oplus_{l\in C}\,l(\sigma)$$

By definition $\oplus$ preserves the edges introduced by the literals. Therefore, $spec^{\oplus}$ will have at least the edges of *spec*.

$$\forall_{\sigma}\left(up_v(\sigma)\implies up_v^{\oplus}(\sigma)\right)$$

Here, $up^{\oplus}$ refers to the *up*-edges in $spec^{\oplus}$. We set E to be the set of all literals of all clauses. The functions f_v are now a precise model for the edges in $spec^{\oplus}$. $up_v^{\oplus}$ is again an over-approximation of up_v. It follows by transitivity that f_v is in fact an over-approximative edge cover for *spec*.

5.3.3 Approximative Edge Cover

We only create a single edge cover for all up_v of all variables $v\in V$. This approximation makes the procedure simpler while sacrificing completeness. We will show in Section 5.6 how the approximation affects the procedure in detail.

5.3.4 Algorithm overview

Finding edges can be understood as a combinatorial game The game is played in turn by player A, who wants to find a compact description of all edges in *spec* and player B, who repeatedly points out edges that player A has missed. The algorithm outline is charted in algorithm 1. Player A makes his decision in the procedure *update* (see section 5.3.6). The turn of Player B is realized by running the query 5.6 in the SMT-solver.

Algorithm 1 Literal discovery algorithm

```
procedure FIND_LITERALS(spec)
    E^0, L^0 ← ∅
    for i = 0, . . . do
        σ* ← query(E^i)
        if σ* = ϵ then
            return L^i
        else
            L^{i+1}, E^{i+1} ← update(L^i, σ*)
        end if
    end for
end procedure
```

The edge cover game Player A proposes E, a set of equality constraints, to player B. Player B then queries the solver engine to find an edge sample σ^* that is yet not covered by the constraints in E.

The game ends when player B fails to find a counter example or player A withdraws.

If player A wins, the set of literals L is fed to the Boolean Structure Synthesis stage.

We set a round limit to decide whether player A shall withdraw. This bounds the literal set and the number of invocations to the SMT-solver.

5.3.5 Counter-example Query

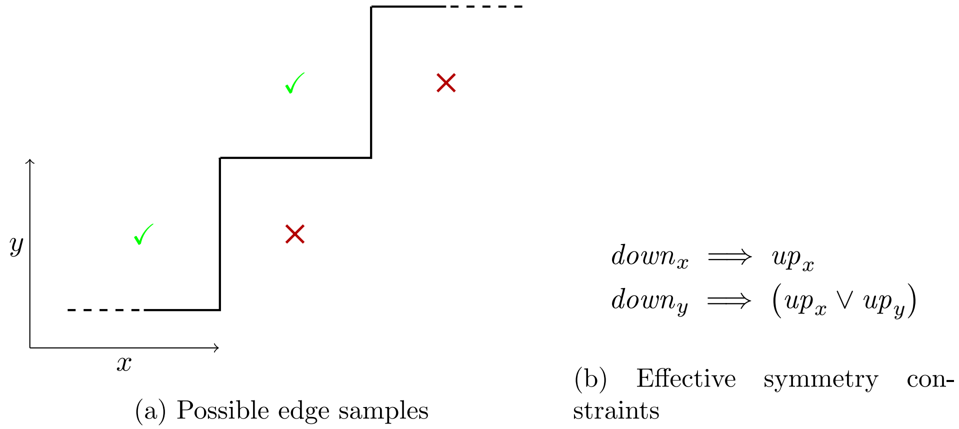
The turn of player B is given by the following query (figure 5.6) to the SMT solver The first three constraints follow directly from the game description. The returned σ should be valid, not covered by any line in E and there must be an edge in at least one variable.

Symmetry Constraint An edge could be detected by samples on either side of a constraint. Say, the specification contains the constraint $x < 5$. Then both $up_x(4)$ and $down_x(5)$ are true. This is due to the symmetry of $\oplus$ as edge detection operator.

Input	$E \subseteq \Sigma \rightarrow \mathbb{B}$
Output	$\sigma,\ up_v(\sigma),\ down_v(\sigma)$
Query	$\sigma \in \Sigma$
	$\bigwedge_{e \in E} \overline{e(\sigma)}$
	$\bigvee_v up_v(\sigma)$
	$\bigwedge_v \left(down_v \implies \bigvee_{v' \preceq v} up_{v'} \right)$

Figure 5.6: Counter-example query

The last condition in 5.6 enforces, that an edge can only be approached form one side. We require a total order $\prec$ over the program variables. To this end, we use the order induced by the initial enumeration of the variables in V. The effect of symmetry breaking can be seen in the simple example of figure 5.7.



(a) Possible edge samples

(b) Effective symmetry constraints

Figure 5.7: Restricting edge detection to one side of the edge by enforced order $(x \prec y)$

Note, that this is only one possible solution to the symmetry problem. We could alternatively check if any step away from σ taps on a value in E. This would, however, require replicating the equalities in E for each variable and direction.

5.3.6 Edge Set proposal

In this section we describe how the algorithm proposes a new set of equality constraints E from a set of edge samples S. The decision logic of player B

is implemented in procedure *update*. We restrict the proposed equalities to contain linear monomials with a sign of $\{-1, 1\}$. This allows us, to conclude from a single edge sample to a new linear constraint using the algorithm 2.

Algorithm 2 *update* procedure without modulus

```
procedure UPDATE_LINEAR(E, σ*)
    e ← 0
    for v ∈ V do
        if up_v(σ*) then
            e ← e + (v − σ*_v)
        else if down_v(σ*) then
            e ← e − (v − σ*_v)
        end if
        E ← E ∪ {e = 0}
        L ← L ∪ {e > 0}
    end for
end procedure
```

One call to UPDATE_LINEAR will propose a new linear constraint without modulus. With respect to the symmetry constraint it is always safe to accept up_v. Only if up_v does not hold, we know that $down_v$ is consistent with the symmetry constraint. If both up_v and $down_v$ hold, we dismiss the down-edge for this sample.

5.3.7 Modulus Edge Proposal

`FIND_LITERALS` will diverge if the specification can only be captured by modulus constraints. Modulus constraints will provide edges for infinitely many linear constraints. The algorithm is given in listing 3.

We consider the process to be divergent if the number of samples exceeds a given *limit*. In that case, the procedure `DETECT_MODULUS` proposes modulus constraints based on the current set of constraints. The constraints have the form

$$(v - r) \bmod m = 0$$

We restart sampling, fixing Σ and the edge steps to stay in the linear set given by the constraint. Finally, the algorithm excludes the linear set for future samples and returns to sampling in Σ with unit size steps.

Algorithm 3 derives linear constraints from edge samples and infers modulus constraints upon divergence

---

procedure UPDATE_MODULUS(E, σ^*)
  if $|E| < limit$ **then**
    $E, L \leftarrow$ `UPDATE_LINEAR`$(E, sigma^*)$
  else
    $E_m, v, step \leftarrow$ `DETECT_MODULUS`(E)
    $L \leftarrow \{l \in L | a_l^v \neq 0 \wedge \forall_{v' \neq v} a_l^{v'} = 0\}$
    $E \leftarrow \{e \in E | a_e^v \neq 0 \wedge \forall_{v' \neq v} a_e^{v'} = 0\}$
    $tmp \leftarrow up_v$
    $up_v \leftarrow \lambda(\sigma)\ (spec(\sigma) \oplus spec(\sigma[v+ = step]))$
    for $e \in E_m$ **do**
      $\Sigma_m \leftarrow \{\sigma \in \Sigma | e(\sigma) = 0\}$
      $L_m \leftarrow$ `FIND_LITERALS`(E, Σ_m)
      $L \leftarrow L \cup L_m \cup \{e(\sigma) > 0\}$
    end for
    $E \leftarrow E \cup E_m$
    $up_v \leftarrow tmp$
  end if
end procedure

The actually algorithm for proposing modulus can be seen in listing 4.

Algorithm 4 proposes a modulus constraint based on the current line cover

procedure DETECT_MODULUS(E, σ^*)
  for $v \in V$ **do**
    $E^* \leftarrow \{e \in E | a_e^v \neq 0 \wedge \forall_{v' \neq v} a_e^{v'} = 0\}$
    $N \leftarrow \{a_e b_e \in E^* | e \in E^*\}$
    for $i = max_{mod}, .., 3$ **do**
      $Histo \leftarrow \{(r, \sum_{n \in N} n \bmod i = r) | r \in \{0, .., i-1\}\}$
      $r_1, c_1 \leftarrow argmax_{r,c}\{c | (r, c) \in Histo\}$
      $r_2, c_2 \leftarrow argmax_{r,c}\{c | (r, c) \in Histo \wedge c < c_1\}$
      if $c_1 + c_2 = |N|$ **then**
        $e_1 \leftarrow \lambda(\sigma)\,((\sigma_v - r_1) \bmod m = 0)$
        $e_2 \leftarrow \lambda(\sigma)\,((\sigma_v - r_2) \bmod m = 0)$
        return $\{e_1, e_2\}, v, i$
      end if
    end for
  end for
  fail
end procedure

The algorithm, although very primitive, finds basic repeating patterns in a single variable. It is incomplete for more complex patterns. The algorithm gathers the offsets N of all planes that use only the variable v. It then checks for a set of candidate moduli, how many planes would have the same remainder. If the top two bins cover all cases, it proposes that modulus and the respective remainders as offsets. If that never happens, the algorithm aborts the synthesis procedure altogether. The algorithm can be configured by setting a maximum modulus candidate max_{mod}.

5.4 Boolean Structure Synthesis

After literal discovery has completed we are left with a complete set of literals L for the specification *spec*. During boolean structure synthesis, the literals are pieced together to form conjunctive clauses that imply *spec*. If Boolean Structure Synthesis is given a complete set of literals, it is only limited by the solvers abilities.

Given a complete set of literals $L \subseteq \Sigma \rightarrow \mathbb{B}$ find a DNF formula f such that

$$\forall_\sigma \left(f(\sigma) \implies spec(\sigma)\right)$$

5.4.1 Clause Construction

Input	$L \subseteq \Sigma \rightarrow \mathbb{B}$
Output	$enable, negate \in \mathbb{B}^{\lvert L \rvert}$, σ^*
Query	$\sigma^* \in \Sigma$
	$spec(\sigma^*)$
	$clause(\sigma^*)$
	$def\ \ clause(\sigma) := \bigwedge_{l \in L} (enable_l \implies negate_l \oplus l(\sigma))$
	$\forall \sigma \in \Sigma \left(clause(\sigma) \implies spec(\sigma)\right)$

Figure 5.8: Initial clause construction query

The algorithm operates by repeatedly synthesizing clauses where each clause is given by

$$c_i = \bigwedge_{l \in C_i} l$$

To this end, it finds a specific σ^* and a subset of the literals L such that (σ^*) holds. We continue by constructing a clause around this positive example, such that

$$c(\sigma^*) \wedge \forall_\sigma \left(c(\sigma) \implies spec(\sigma)\right)$$

The positive example makes sure that c is at least feasible for σ^*. The complete query is shown in Figure 5.8. This deviates the need of having an additional existential quantifier to ensure the feasibility of the clause. The all-quantification is necessary to make sure that the clause is pure. We will not further discuss the methods for eliminating this quantifier.

5.4.2 Clause Refinement

Clause Refinement is concerned with optimizing a clause with respect to our optimization problem 5.2. We minimize the number of literals occurring in each clause while still satisfying *spec*. At the same time, we maximize the subset of Σ covered by the clause. Clause Refinements implements this logic by finding an initial clause. It then greedily queries for a new clause that improves both objectives at once. If no such clause exists it returns the last clause it could generate in the process.

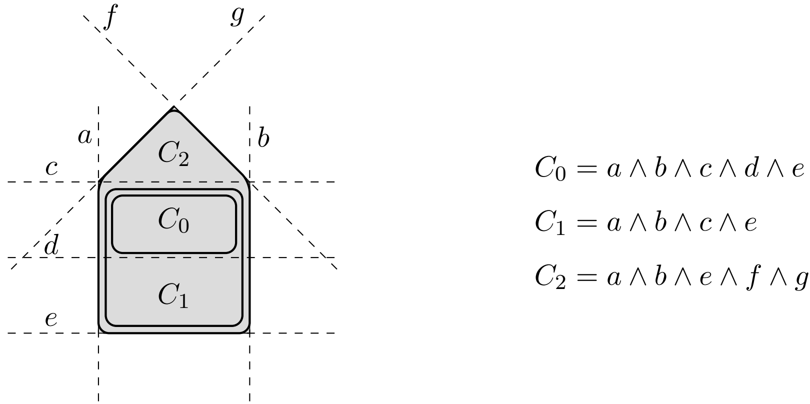


Figure 5.9: Iterations of Clause Refinement.

Let c and c' be clauses that satisfy *spec*. We say c' subsumes c, if

$$\forall_\sigma \left(c(\sigma) \implies c'(\sigma)\right)$$

Furthermore, we call a clause c minimal if there is no clause that subsumes c. An initial clause may contain literals with no discriminative power as the only requirement is that they are feasible and satisfy *spec*. Consider figure 5.9. The procedure found the initial clause C_0. The literal d can be dropped giving way to a more general clause C_1. Similarly, by replacing the literal c and adding the literals f and g the clause grows further. As C_2 is maximal the procedure adds it to the formula and continues with the next clause.

Refinement We refine clauses by repeating the initial query with the known example σ^* and with additional constraints. Let $C' = \{l_1, .., l_n\}$ be a clause, then we run the initial query again with the additional constraints that

$$\bigvee l \in C' \left(\overline{enable_l}\right)$$

and

$$\forall_{\sigma \in \Sigma}. \left(c'(\sigma) \implies c(\sigma)\right)$$

The new clause c must differ from c' in at least one literal. We repeat the queries, keeping the last c' as the clause that should be simplified. Finally, the last feasible clause is added to the DNF formula.

Termination and correctness We will sketch a proof for termination and correctness. Each produced clause subsumes the initial clause because this is an explicit constraint in the refinement query and the subset relation is transitive. Consider, we dropped the correctness constraint from the query. Then the process will eventually yield the empty clause. As it is empty no literal can be dropped and the process terminates. The question remains how many iterations are required to reach the empty clause. Remember, that the set of literals is finite and so only a finite number of clauses can be constructed. This shows termination but also means that the number of iterations can be exponential.

5.5 Complexity

5.5.1 Literal discovery

The literal discovery algorithm produces additional constraints in BAPA. Each solver run yields an additional constraint. The complexity of the specification is unknown. The total complexity is the number of required constraints times the complexity of the solver-procedure for the specification.

5.5.2 Clause construction

The current implementation of clause construction uses QBAPA (Presburger Arithmetic with Quantifiers) queries for each clause. Each iteration construct a clause of the final DNF formula.

5.6 Limitations

Our model is necessarily incomplete even without modulus constraints.

5.6.1 Over-approximative Line Cover

Literal Discovery over-approximates the line cover by using a single set of lines for edges in all variables. Due to this, Literal Discovery may terminate with a set of literals that is not complete. Consider the example in Figure 5.10. Assume, that Literal Discovery has drown samples of Σ at the positions marked with "*". The *dashed* lines denote the resulting line cover.

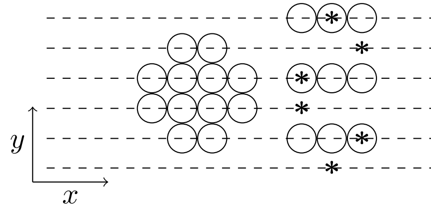


Figure 5.10: Complete approximate line cover that does not model all edges.

This line cover hides the edges in the left-hand side structure. The resulting literal set will be incomplete. This effect makes our procedure theoretically incomplete but does not occurr in our application.

5.6.2 Modulus Constraints

The modulus proposal algorithm shown in Listing 4 is very basic and can not deal with the following situations.

Mixed Modulus Our modulus detection model can not deal with multiple frequencies in the modulus constraints as in

$$(x \bmod 5 = 3 \land x < 3) \lor (x \bmod 7 = 2) \land x > 20$$

Multi-variable Modulus The model does not support modulus constraints that rely on multiple variables. So, it will not find

$$x + y - z \bmod 7 > 2$$

Chapter 6

Implementation

We now have a procedure that will find a boolean predicate for a specification over integer variables. The procedure is the core component of our actual algorithm. We will show, how the procedure can be used to synthesize pre-conditions for optimized code paths in vectorized programs. We implemented the variant analysis as a pass in the Whole-Function Vectorizer by [Karrenberg and Hack, 2011] which is based on the LLVM framework [Lattner and Adve, 2004]. Although the implementation operates on LLVM-IR, we are going to present it in pseudo C code for enhanced readability. The variant analysis runs after the mask and vectorization analysis have completed but before the actual vectorization. At this point, the program is still scalar. The variant analysis can use and modify analysis results and introduce new code paths.

The variant analysis pass has to accomplish the following:

- Extract a set of program variables V describing the vectorized program, from scalar LLVM code.
- Restrict the set of possible program variable states. This includes the *value range* (lower/upper value bounds) and *alignment* (the value is a multiple of the vectorization width). For example, in OpenCL the global `thread_id` is a non-negative value that is aligned.
- Specify a condition on the program variables that is true whenever a specialized code path could be used (the specification *spec*). The conditions model optimizable patterns in memory accesses (*Consecutive*, *Uniform*) and control-flow (code executed by all SIMD-lanes).
- Run the Synthesis Procedure to find a pre-condition for the specification.

6.1 Variants

As shown below, variant conditions are constraints on single values of the LLVM program. For the tested variants this is either the entry mask of a Basic Block or the subscript value in an array access. The entry mask of a Basic Block is the bitvector of SIMD-width that is true for all SIMD-lanes that execute the block.

6.1.1 Complete Mask

If the mask for a section is always completely set, there is no need for additional predication code. As in our motivating example, complete masks often occur when a range of threads will taken one direction and only few will go into an alternative direction. If m is the mask predicate then the specification is given by:

$$spec(\sigma) = \bigwedge_{0 \leq i < width} m_i(\sigma)$$

6.1.2 Consecutive Memory Access

Many current SIMD instruction sets only feature vector loads/stores for consecutive address ranges. Furthermore, the addresses need to be aligned. Threads in the same SIMD-group executing the same memory instructions frequently access completly unrelated addresses. Only if consecutiveness can be proven, the vectorizer will use an efficient SIMD instruction.

$$spec(\sigma) = \bigwedge_{0 \leq i < width-1} a_i(\sigma) + 1 = a_{i+1}(\sigma)$$

6.2 Mapping LLVM to the Problem Domain

The variant analysis translates LLVM Control-Flow Graphs to expression trees. We use the SMT-LIB v2.0 format which is widely supported by SMT solvers (for a full specification refer to [Barrett et al., 2010]). These expression trees have to be realized for all lanes in the SIMD-package because the variant conditions refer to all lanes at once. They also refer to a single value only which is the root of the expression tree.

6.2.1 Type and instruction mapping.

Similar to [Karrenberg et al., 2013] we adhere the SMTlib 2.0 definition of the modulus operator [1]. The definition used in OpenCL follows the C99 standard. We could alternatively define a custom modulus operator

[1] `http://smtlib.cs.uiowa.edu/theories/Ints.smt2`

in SMTlib 2.0 syntax and use that definition instead. However, the two definitions coincide for non-negative operands.

Integer values in LLVM have always a fixed bit width. We map every LLVM integer variable to a SMT-variable that may assume any value in $\mathbb{Z}$. If we can model the specification in this generalized domain it will also hold for a fixed-size values. There exists a theory fragment for fixed-width integer variables in SMTlib 2.0[2]. The fragment accurately models the arithmetic and defines functions for bit-level operations (for example shifts). Constraints of the considered kind do not usually arise from bit-level operations. Integer overflow is explicitly not defined in the OpenCL standard. To keep the implementation simple, we do not use this extended theory and skip instructions that rely on bit-level operations.

Leaf values. We expand the data dependencies starting form the root value until we hit unsupported operations or data types. We replace any instruction that does not directly map to the SMT-format by a fresh variable. These include instructions with side effects, such as loads and calls to external functions returning values.

PHI-nodes and Control-Flow. The data dependencies in actual programs are often cyclic. This is neither supported by current SMT solvers nor our synthesis technique. We break cyclic dependencies by introducing leaf variables for PHI-nodes that use loop-carried values. Alternatively, PHI-nodes can occur in the joins of acyclic control-flow. In that case, we apply canonic control- to data-flow conversion.

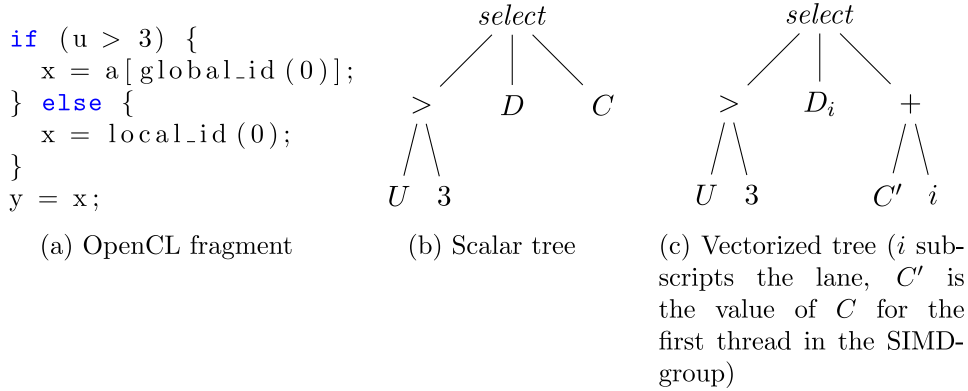
Mapping of mask predicates. The Mask Analysis pass built into the Vectorizer returns for every Basic Block a boolean expression tree. This expression evaluates to true whenever a thread in the vectorized program reaches it. When the root value tested by our specification is a mask, we traverse the mask tree. This tree will eventually refer to constants or instructions in the actual program. The traversal continues seamlessly with those values using the logic we described earlier.

6.2.2 Vectorizing the expression tree.

We could vectorize the expression tree naively by replicating it once for every lane, introducing independent variables for the leaves. However, we can do better as shown in Figure 6.1. The vectorization analysis can tell us how the results of the same instruction are related among the SIMD-lanes. It infers three kinds of relations: U for uniforms, C for consecutive and D meaning the relation is unknown (for a detailed introduction refer to Chapter 2). If

[2] `http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2`

an instruction is annotated as uniform or consecutive, its result vector is determined by the the first lane. This only matters for leaf values where we now only need to introduce a single free variable. The actual value for each lane can be computed.



(a) OpenCL fragment

(b) Scalar tree

(c) Vectorized tree (i subscripts the lane, C' is the value of C for the first thread in the SIMD-group)

Figure 6.1: Derivation of a vectorized expression tree (U,C and D refer to uniform, consecutive and divergent values

The set of variables Σ for the synthesis procedure is the union of the leaf variables of the vectorized tree.

Value Alignment Alignment is an important property of values in vectorized program. A value is aligned if it is always a multiple of the vectorization factor. In the case of OpenCL, this holds in particular for the consecutive thread IDs (*get_global_id* and *get_local_id*). The Vectorization Analysis can detect alignment of values derived from other aligned values. We substitute aligned base values by a fresh variable multiplied with the vectorization factor. The synthesis procedure is thus oblivious to alignment.

Distinguishing thread from uniform state We distinguish uniform variables that are completely independent of the thread ID as Σ_u. All others depend on the thread ID in some way and are placed in the set Σ_t. We write,

$$\sigma = \sigma_u | \sigma_t$$

meaning we can access these two partitions of σ by the subscripts u (uniform values) and t (consecutive and divergent values).

6.3 Efficient pre-condition testing in loops

We optimize the kernels by writing a case distinction based on the pre-condition in the scalar kernel. Initially, the code for both paths is the same. We assert to the vectorized that the optimizable condition of the specification holds in that path. The vectorizer will create optimized code in the path guarded by the pre-condition and the same code as before for the default path. Take for example the fragment in Figure 6.2. Note, that the `thread_id`-loop is not the outer-most loop. This loop will be vectorized to width 4 such that 4 consecutive threads will be processed using vector instructions loop body. This happens whenever there is a `barrier`-intrinsic in the original kernel code (for details refer to Chapter 2). If we insert a case distinction for the complete mask pre-condition, we get the result in Figure 6.3.

```
limit = 1 << n
for (..) {
  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id < limit) {
      ... // default code
    }
  }
  limit = op(limit)
}
```

Figure 6.2: Original kernel fragment

```
limit = 1 << n
for (..) {
  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id + 3 < limit) {
      ... // optimized code
    } else if {thread_id < limit) {
      ... // default code
    }
  }

  limit = op(limit)
}
```

Figure 6.3: Kernel with a naive case distinction

If we have a definition for `op`, we can actually do better than testing the pre-condition naively in every iteration. Say, `op` is defined as

$$\texttt{op}(\texttt{limit}) = 2 * \texttt{limit}$$

6.3.1 Loop-invariant Pre-condition

In order to be able to hoist the condition test out, we are rather interested in a pre-condition that implies that the optimizable condition holds for all threads. In the example of Figure 6.3, the *default* code path will not be taken once `limit` is a multiple of 4.

We capture this property generally in the definition of the loop-invariant pre-condition. We are given a specification *spec* on the program variables. The loop-invariant specification is then

$$lispec(\sigma_u) = \forall_{\sigma_t \in \Sigma_t} spec(\sigma_u, \sigma_t)$$

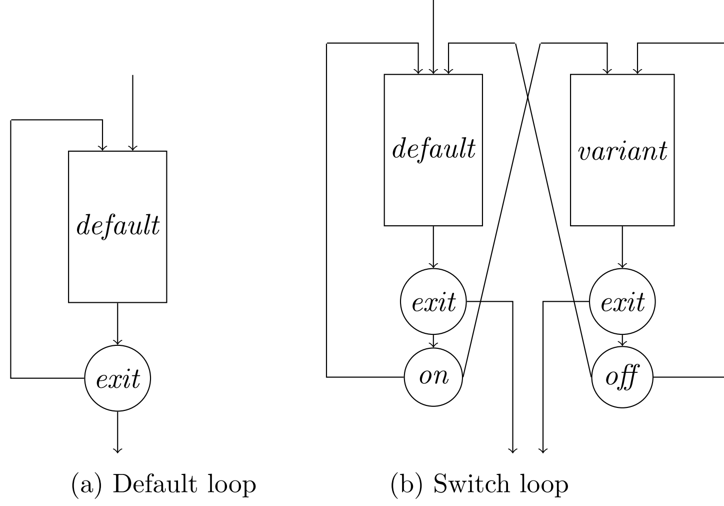
```
limit = 1 << n

for (..) {
  if (limit % 4 != 0)
    goto default_loop

  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id + 3 < limit) {
      ... // optimized code
    }
  }
  limit = limit / 2
}

default_loop:
for (..) {
  if (limit % 4 == 0)
    goto specialized_loop:

  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id < limit) {
      ... // default code
    }
  }

  limit = limit / 2
}
```

Figure 6.4: Kernel with specialized loops. As `op` halves `limit` in every iteration.

If we applied the model for *lispec* naively in our example, we get the result of Figure 6.4. Note, that we achieved to hoist the test out of the thread loop. However, the code is still fairly complex and we can in fact optimize out the `goto` to either version in some cases.

6.3.2 Optimizing the Loop Order

We can generalize the observation made in the example of Figure 6.4. In Figure 6.5, *on* stands for a condition that enables the specialized loop. *off* is



(a) Default loop (b) Switch loop

Figure 6.5: Specialized Loop

a condition that checks whether execution must return to the default loop. *exit* is the original exit condition that is preserved in both loops.

We want to eliminate either the *off* or *on* branch. To this end, we are only interested in finding solutions were either *off* or *on* are always false.

This optimization is only possible, if *lispec* is in fact also a subset of the possible program states.

Given that this holds, we can achieve this through synthesizing the following specifications

The *on*-condition holds, if execution can continue in the specialized loop, *without ever leaving it.*

$$spec_{on}(\sigma_u) = lispec(\sigma_u) \implies lispec(op(\sigma_u))$$

Conversely, the *off*-condition holds, if execution must take the default loop, *without ever leaving it.*

$$spec_{off}(\sigma_u) = lispec(op^{-1}(\sigma_u)) \implies lispec(\sigma_u)$$

Scheduling Decision We synthesize either condition and decide to move the default loop in front, if we find a $model_{off}$ and

$$\forall_{\sigma_u \in \Sigma_u} \overline{model_{off}(\sigma_u)} \implies lispec(\sigma_u)$$

Conversely, we can always safely move the specialized condition in front and dispatch to the default loop.

Inverse Loop Operator In the definition of $spec_{off}$ op^{-1} is the inverse loop operator. As op may not be injective, the inverse may not be functional.

Still, for the `DwtHaar1D` example of Figure 6.4, we define

$$op^{-1}(\texttt{activeThreads}) = 2 * \texttt{activeThreads}$$

Note, that this inverse does not correctly model the fact that `activeThreads` = 0 could result from either prior loop iteration `activeThreads` = 0 OR `activeThreads` = 1. We account for that missing transition by adding it explicitly to the specification of *off*.

Synthesizing on/off tests Note, the our Synthesis Procedure finds under-approximations. Therefore, it is safe to apply it directly to $spec_{on}$ because we can safely stay in the default loop. To get the appropriate over-approximation for $spec_{off}$ however, we have to invert the specification and invert again the resulting model.

Example For the example of Figure 6.4 where $op(limit) = limit/2$, we get

$$limodel(limit) = limit \bmod 4 = 0$$

We get the *off* model, assuming that *limit* is always a power of two value

$$model_{off}(limit) = limit < 4$$

which satisfies the constraint given above. This means that we move the *default* loop in front and obtain the result of Figure 6.6.

```
limit = 1 << n

for (..) {
  if (limit < 4)
    goto default_loop

  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id + 3 < limit) {
      ... // optimized code
    }
  }
  limit = limit / 2
}

default_loop:
for (..) {
  // if (limit % 4 == 0) // optimized out
  //   goto specialized_loop:

  for (thread_id = 0; thread_id < num_threads; ++thread_id) {
    if (thread_id < limit) {
      ... // default code
    }
  }

  limit = limit / 2
}
```

Figure 6.6: Kernel with specialized loops and the *off*-branch removed. As `op` halves `limit` in every iteration, the default code path

6.4 Synthesizing the Variant Test

We synthesize an optimized variant test by applying the synthesis procedure directly to the vectorized expression trees. For the common case that the `thread_id` is tested against a threshold, we obtain the complete mask test seen in Figure 6.7.

Scalar mask	$m_i = C < U$
Specification test	$\bigwedge_{0 \leq i < width} (C + i < U)$
Synthesized condition	$C + width - 1 < U$

Figure 6.7: Frequently occurring mask in a complete mask test (U, C are not aligned).

Chapter 7

Evaluation

Our implementation is semi-automatic. The implemented Variant Analysis pass operates directly on the code of the kernel. The implementation is based on the SMT solver *Z3* ([De Moura and Bjørner, 2008]) which is an efficient solver for Quantified Linear Integer Arithmetic. Variant Analysis checks various properties of the control-flow masks (Complete mask, Scalar mask, Empty mask) and array offsets (Uniform access, Consecutive access). We refer the reader to Chapter 2 for definitions of these properties. If it succeeds, the Variant Analysis outputs a synthesized pre-condition over program variables. We manually crafted the optimized code path and inserted the synthesized pre-condition.

We report runtime performance of OpenCL kernels with optimized memory access and control-flow patterns.

Limitations. The Variant Analysis was able to autonomously find the pre-conditions for the control-flow examples. Our experiments show that no solver was able to synthesize the loop-invariant pre-condition for specialized loops (introduced in Section 6.3). These include the pre-conditions used in [Karrenberg et al., 2013].

7.1 OpenCL Performance

We report the measured runtimes for control-flow optimizations in Figure 7.1 and Figure 7.3. Results for memory access optimizations are shown in Figure 7.2. The machine was Intel®Core™2 Quad CPU Q9550 with 2.83GHz equipped with 4 GB RAM. The CPU has SSE vector registers of width 4.

Reported results are the median of 1000 kernel runs. The baseline is a version of WFV-OpenCL (column *default*) that executes the kernel with the LLVM MCJIT. A complete listing of the optimized kernels can be found in the Appendix A.

7.1.1 Complete Mask

A mask is complete, if all lanes in a SIMD-group are active. We optimize the `DwtHaar1D` and `FastWalshTransform` OpenCL benchmarks from the `AMD APP SDK`[1].

Our measurements are shown in Figures 7.1 and 7.3.

Optimizations We used the synthesized pre-conditions from our Variant Analysis pass. Loop specialization was done as an additional step by hand.

7.1.2 Case: `DwtHaar1D` Benchmark

Benchmark	Work size	Version	Result	Speed-up
Complete mask				
DwtHaar1D	65536	original	8.10	-
		unroll	8.96	-0.10
		opt-naive-unroll	7.08	+14.40
		opt-unroll	6.91	+17.13

Table 7.1: Compared are the original kernel (*original*) and various optimized versions. *opt* refers to a variant with a specialized code path, *unroll* means the last two iterations were unrolled and *naive* means the specification was tested directly. Timing results are given in milliseconds.

Original kernel The code loop of the original `DwtHaar1D` is shown in Figure 7.1.

Synthesized Pre-condition The Variant Analysis found out that the body of the `if`-statement in the loop is full, if

$$\texttt{first_local_id} < \texttt{activeThreads}$$

where `first_local_id` is the thread ID of the first thread in the SIMD-group. The Variant Analysis pass needed 33 milliseconds to synthesize the pre-condition (median of 100 runs).

Loop Specialization We manually optimized the loop using the synthesized *off* condition

$$\mathit{off}(\texttt{activeThreads}) = \texttt{activeThreads} < 4$$

The conditions exploits that `activeThreads` is always a power of two. The specialized loops are shown in Figure 7.2.

[1] AMD APP SDK v2.7 (`http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`)

```
uint activeThreads = (1 << levels) / 2;
uint midOutPos = signalLength / 2;

for(uint i = 0; i < levels; ++i)
{
  if(localId < activeThreads)
  {
    float data0 = sharedArray[2 * localId];
    float data1 = sharedArray[2 * localId + 1];

    sharedArray[localId] = (data0 + data1) / sqrt((float)2);
    uint globalPos = midOutPos + groupId * activeThreads + localId;
    coefsSignal[globalPos] = (data0 - data1) / sqrt((float)2);

    midOutPos >>= 1;
  }
  activeThreads >>= 1;
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

Figure 7.1: Core loop of the original `DwtHaar1D` benchmark

Unrolling We achieved the best results when the last two iterations of the loop were unrolled. The compiler did not apply this transformation automatically. To show that the performance gain is still due to our synthesized pre-condition, we measured unrolling for the original and the kernel with the specialized code path.

The `DwtHaar1D` consists of a single loop with an optimizable pre-condition. We compare the *default* version, a version which checks the pre-condition in every iteration (*partial*) and a version with completely specialized loops *complete*.

7.1.3 Case: `PrefixSum` Benchmark

Original Kernel The `PrefixSum` consists of two loops as shown in Figure 7.4. The second loop is identical to the loop found for `DwtHaar1D`. The first loop doubles an iteration variable in every iteration and can be optimized similarily. The *both* version is optimized using the information that `length` is a power of two value. Different than the `DwtHaar1D` benchmark, this can not be inferred directly from the kernel code. To this end, we decided to measure a version (*single-loop*) that does not depend on this information as well.

Synthesized Pre-condition For both loops, the synthesized pre-condition is

$$\texttt{first_tid} + 3 < \texttt{d}$$

```
uint activeThreads = (1 << levels) / 2;

for(uint i = 0; i < levels; ++i)
{
  if (activeThreads < 4)
    break;

  ... // complete mask code

  activeThreads >>= 1;
  barrier(CLK_LOCAL_MEM_FENCE);
}

for(uint i = 0; i < levels; ++i)
{
  barrier(CLK_LOCAL_MEM_FENCE);
  ... // default code
  activeThreads >>= 1;
}
```

Figure 7.2: Core loop of the original `DwtHaar1D` benchmark

Benchmark	Work size	Version	Result	Speed-up
PrefixSum	512	original	0.12	-
		single-loop	0.10	+0.16
		both	0.08	+0.48

Figure 7.3: We compare the *original* kernel against versions that optimize both loops (*both*) and only the last loop (*single-loop*).

Specialized Loops The second loop can be specialized just like the loop in the `DwtHaar1D` benchmark (shown in Figure 7.2).

```
for(int d = length>>1; d > 0; d >>=1)
{
  barrier(CLK_LOCAL_MEM_FENCE);

  if(tid < d)
  {
    int ai = offset*(2*tid + 1) - 1;
    int bi = offset*(2*tid + 2) - 1;

    block[bi] += block[ai];
  }
  offset *= 2;
}

...

for(int d = 1; d < length ; d *= 2)
{
  offset >>=1;
  barrier(CLK_LOCAL_MEM_FENCE);

  if(tid < d)
  {
    int ai = offset*(2*tid + 1) - 1;
    int bi = offset*(2*tid + 2) - 1;

    float t = block[ai];
    block[ai] = block[bi];
    block[bi] += t;
  }
}

barrier(CLK_LOCAL_MEM_FENCE);
```

Figure 7.4: Core loops of the `PrefixSum` benchmark.

7.1.4 Optimized Memory Access

In Table 7.2, we reproduce the results from [Karrenberg et al., 2013].

Consecutiveness Condition We use the consecutiveness condition by [Karrenberg et al., 2013] to re-produce their results. The modified specification applies for `FastWalshTransform` and `BitonicSort`. We asked for which state of the uniform variables is a memory access consecutive for all threads. This yields an under-approximation of the program states with consecutive memory access.

Using the notation from Chapter 6, we write the uniform variable states as σ_u and the thread dependent states as σ_t. The checked specification was

$$spec(\sigma) = \forall_{\sigma_u \in \Sigma_u, \sigma_t \in \Sigma_t} \left(consecutive(\sigma) \right)$$

Configuration The *Z3* SMT-solver failed for `FastWalshTransform` and `BitonicSort` in the Literal Discovery stage. This is due to the Non-linear

Benchmark	Work size	Default	Optimized	Speed-up
FastWalshTranform	2^{24}	79.18	21.07	2.76
BitonicSort	2^{20}	576.00	567.573	0.01

Table 7.2: OpenCL runtime results for optimized access patterns.

Integer Arithmetic used in the address computations. To this end, we could apply our Variant Analysis pipeline directly to the problem. To emulate what our Synthesis Procedure would do, we used the known solution

$$spec(s) = s \bmod 4 = 0$$

as the specification. We show the results arguing that these limitations can be overcome by advances in SMT Solving or the synthesis procedure. The solver needed 1513.14 milliseconds to model the emulated specification (median of 100 runs).

Driver Limitations The Whole-Function Vectorizer can not process already vectorized code. This rules out many of the OpenCL kernels in the `AMD APP SDK`,`Parboil`,`Rodinia` benchmark suites. For other kernels the OpenCL driver failed for no comprehensible reason. We attribute this to the missing support for unaligned vector loads/stores on our test machine. This again removes benchmarks from the list of scalar OpenCL kernels. None of the kernels evaluated in [Karrenberg, 2014] feature control-flow patterns complex enough for our synthesis procedure.

Chapter 8

Conclusions

We apply a novel synthesis procedure that synthesizes pre-conditions for optimizable memory access patterns and control-flow patterns. The results show, that the specialized code paths make the generated CPU code more efficient. Our approach improves over prior work both in terms of scope,runtime performance and automation. For linear integer problems, our implementation is automated up to the final code transformation. The procedure consistently finds solutions within fractions of a second making it applicable in JIT-compilation. We applied our synthesis procedure to a variety of optimizable pre-conditions in Whole-Function Vectorization. These include memory access patterns (consecutive) and the *Complete Mask* control-flow pattern.

8.1 Synthesis Procedure

Our procedure synthesizes a boolean predicate over integer variables that under-approximates a specification. We treat the specification as a black box that is queried using SMT solvers.

Depending on the structure of the specification, these queries may be undecidable or have a high theoretic complexity. In fact, with current SMT solvers our synthesis procedure can not automatically synthesize preconditions found in [Karrenberg et al., 2013]. The tested SMT solvers (Z3, MiniSMT) can either not handle quantifiers (MiniSmt) or Non-linear Integer Arithmetic (Z3). This could be achieved by algorithmic improvements in the synthesis procedure that deviate the need for quantifiers.

However, the solver found a solution for synthesizing the pre-conditions in the control-flow examples. The Linear Integer Arithmetic involved in those benchmarks could be handled by *Z3*. The small number of variables in our SMT queries further helps to keep the solver runtime low.

Theoretic Limitations. The constraint model is restricted. Even within the constraint model, the approximations during Literal Discovery make the synthesis procedure incomplete. Our experiments show, that the theoretical incompleteness is ineffective in our application.

8.2 Implementation

We implemented our synthesis procedure in a prototype. Even tough we automate the process of pre-condition synthesis, we still need to apply the code optimizations by hand. This step was left out because of time constraints. A complete implementation will automate the entire optimization pipeline.

Chapter 9

Future Work

9.1 Algorithmic Improvements

Quantifier-free Boolean Structure Synthesis We argue, that the quantifier in Boolean Structure Synthesis can be eliminated. For example, a repeated query could find a counter example and a new clause that excludes it while adhering to all samples found so far. This will be also be necessary

Improved Literal Discovery We used the loop-independent consecutiveness specification for the `FastWalshTransform` and `BitonicSort` benchmarks. The criterion uses a quantifier over all `thread_ids`. We reckon that Literal Discovery can be improved to eliminate this quantifier. In Boolean Structure Synthesis, the quantifier is already implied by the quantifier that checks that the specification holds for all values in the clause.

Precise Edges We apply approximations during Literal Discovery to make the synthesis procedure more efficient. However, the theoretical incompleteness may show in other applications. The procedure can be adapted to synthesize predicates for edges in all variables.

Arbitrary Monomial Factors Our current implementation will only find linear constraints where the variables have factors in $\{-1, 0, 1\}$. This restriction could be lifted by letting Clause Construction fill in the factors. This extends on the interpretation that Literal Discovery constructs a set of building blocks. Boolean Structure Synthesis composes the predicate drawing from that library and filling in parameters.

Boolean Structure DNF formulas can be exponentially bigger than the most compact description of a boolean expression. To this end, we reckon that our formulas should be simplified for example by extracting common subclauses via the distributional law.

Extension to other Domains The procedure could be extended to $\mathbb{R}$ arithmetic which could be used to bound divergence of neighboring threads in scientific computations.

Our prototype will require further attention to be integrated in a full code synthesis pipeline. Again, Minismt could be integrated as Z3 is already in the prototype.

9.2 Super Vectorization

Existing SSE-vector code could be vectorized to operate on AVX registers. Interesting question how well vectorized scalar code to AVX would stand-up against super-vectorized SSE code.

9.3 Improved Modulus Constraint Detection

Our algorithm for proposing modulus constraints is simple yet effective. However, in future work the simple algorithm may not suffice anymore. In particular, the procedure should be extended to operate on repeating edges with arbitrary normals. Furthermore, we suggest that an approach based on the inference of semi-linear sets as a decomposition of the samples should be used instead.

Bibliography

[CUD, 2007] (2007). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide.*

[Barrett et al., 2010] Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., Krstić, S., Moskal, M., Moura, L. D., Sebastiani, R., Cok, T. D., and Hoenicke, J. (2010). C.: The smt-lib standard: Version 2.0. Technical report.

[Barthe et al., 2013] Barthe, G., Crespo, J. M., Gulwani, S., Kunz, C., and Marron, M. (2013). From relational verification to simd loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 123–134, New York, NY, USA. ACM.

[Brauer and King, 2012] Brauer, J. and King, A. (2012). Transfer function synthesis without quantifier elimination. *Logical Methods in Computer Science*, 8(3).

[Brauer et al., 2013] Brauer, J., King, A., and Kowalewsk, S. (2013). Abstract interpretation of microcontroller code: Intervals meet congruences. *Science of Computer Programming. Methods of Software Design: Techniques and Applications*, 78, issue 7:862–883.

[Colón et al., 2003] Colón, M., Sankaranarayanan, S., and Sipma, H. (2003). Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–433. Springer.

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA. ACM.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of*

Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

[Dillig et al., 2013] Dillig, I., Dillig, T., Li, B., and McMillan, K. (2013). Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 443–456, New York, NY, USA. ACM.

[Fischer et al., 1974] Fischer, M. J., Fischer, M. J., Rabin, M. O., and Rabin, M. O. (1974). Super-exponential complexity of presburger arithmetic. pages 27–41.

[Gold, 1967] Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10(5):447–474.

[Gulwani et al., 2011a] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. (2011a). Synthesis of loop-free programs. *SIGPLAN Not.*, 46(6):62–73.

[Gulwani et al., 2011b] Gulwani, S., Korthikanti, V. A., and Tiwari, A. (2011b). Synthesizing geometry constructions. In Hall, M. W. and Padua, D. A., editors, *PLDI*, pages 50–61. ACM.

[Itzhaky et al., 2010] Itzhaky, S., Gulwani, S., Immerman, N., and Sagiv, M. (2010). A simple inductive synthesis methodology and its applications. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *OOPSLA*, pages 36–46. ACM.

[Karrenberg, 2014] Karrenberg, R. (2014). Automatic simd vectorization of ssa-based control flow graphs.

[Karrenberg and Hack, 2011] Karrenberg, R. and Hack, S. (2011). Whole Function Vectorization. In *Code Generation and Optimization.*

[Karrenberg et al., 2013] Karrenberg, R., Kosta, M., and Sturm, T. (2013). Presburger arithmetic in memory access optimization for data-parallel languages. In Fontaine, P., Ringeissen, C., and Schmidt, R. A., editors, *FroCos*, volume 8152 of *Lecture Notes in Computer Science*, pages 56–70. Springer.

[Khronos OpenCL Working Group, 2008] Khronos OpenCL Working Group (2008). *The OpenCL Specification, version 1.0.29.*

[Kuncak, 2007] Kuncak, V. (2007). Quantifier-free boolean algebra with presburger arithmetic is np-complete.

[Kuncak et al., 2010] Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2010). Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA. ACM.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA. IEEE Computer Society.

[Laviron and Logozzo, 2009] Laviron, V. and Logozzo, F. (2009). Subpolyhedra: A (more) scalable approach to infer linear inequalities. In Jones, N. D. and Müller-Olm, M., editors, *Verification, Model Checking, and Abstract Interpretation: Proceedings of the 10th International Conference (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 229–244, Savannah, Georgia, USA. Springer-Verlag, Berlin.

[Miltersen et al., 2003] Miltersen, P. B., Radhakrishnan, J., and Wegener, I. (2003). On converting cnf to dnf. In Rovan, B. and Vojtás, P., editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 612–621. Springer.

[Miné, 2006] Miné, A. (2006). The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100.

[Paige and Koenig, 1982] Paige, R. and Koenig, S. (1982). Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454.

[Shapiro, 1983] Shapiro, E. Y. (1983). *Algorithmic Program DeBugging.* MIT Press, Cambridge, MA, USA.

[Shin et al., 2009] Shin, J., Hall, M. W., and Chame, J. (2009). Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocess. Microsyst.*, 33(4):235–243.

[Simbürger et al., 2013] Simbürger, A., Apel, S., Größlinger, A., and Lengauer, C. (2013). The potential of polyhedral optimization: An empirical study. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society. Acceptance rate (full papers): 16

[Srivastava, 2011] Srivastava, S. (2011). *Satisfiability-Based Program Reasoning and Program Synthesis.* Proquest, Umi Dissertation Publishing.

Appendix A

Benchmark kernels

Listing A.1: AMD APP SDK v2.7 disclaimer

```
Copyright (c) 2009-2010 Advanced Micro Devices, Inc.  All rights reserved.

Redistribution and use of this material is permitted under the following
conditions:

Redistributions must retain the above copyright notice and all terms of this
license.

In no event shall anyone redistributing or accessing or using this material
commence or participate in any arbitration or legal action relating to this
material against Advanced Micro Devices, Inc. or any copyright holders or
contributors. The foregoing shall survive any expiration or termination of
this license or any agreement or access or use related to this material.

ANY BREACH OF ANY TERM OF THIS LICENSE SHALL RESULT IN THE IMMEDIATE REVOCATION
OF ALL RIGHTS TO REDISTRIBUTE, ACCESS OR USE THIS MATERIAL.

THIS MATERIAL IS PROVIDED BY ADVANCED MICRO DEVICES, INC. AND ANY COPYRIGHT
HOLDERS AND CONTRIBUTORS "AS IS" IN ITS CURRENT CONDITION AND WITHOUT ANY
REPRESENTATIONS, GUARANTEE, OR WARRANTY OF ANY KIND OR IN ANY WAY RELATED TO
SUPPORT, INDEMNITY, ERROR FREE OR UNINTERRUPTED OPERA TION, OR THAT IT IS FREE
FROM DEFECTS OR VIRUSES.  ALL OBLIGATIONS ARE HEREBY DISCLAIMED - WHETHER
EXPRESS, IMPLIED, OR STATUTORY - INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED
WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
ACCURACY, COMPLETENESS, OPERABILITY, QUALITY OF SERVICE, OR NON-INFRINGEMENT.
IN NO EVENT SHALL ADVANCED MICRO DEVICES, INC. OR ANY COPYRIGHT HOLDERS OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, REVENUE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED OR BASED ON ANY THEORY OF LIABILITY
ARISING IN ANY WAY RELATED TO THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGE. THE ENTIRE AND AGGREGATE LIABILITY OF ADVANCED MICRO DEVICES,
INC. AND ANY COPYRIGHT HOLDERS AND CONTRIBUTORS SHALL NOT EXCEED TEN DOLLARS
(US $10.00). ANYONE REDISTRIBUTING OR ACCESSING OR USING THIS MATERIAL ACCEPTS
THIS ALLOCATION OF RISK AND AGREES TO RELEASE ADVANCED MICRO DEVICES, INC. AND
```

ANY COPYRIGHT HOLDERS AND CONTRIBUTORS FROM ANY AND ALL LIABILITIES, OBLIGATIONS, CLAIMS, OR DEMANDS IN EXCESS OF TEN DOLLARS (US $10.00). THE FOREGOING ARE ESSENTIAL TERMS OF THIS LICENSE AND, IF ANY OF THESE TERMS ARE CONSTRUED AS UNENFORCEABLE, FAIL IN ESSENTIAL PURPOSE, OR BECOME VOID OR DETRIMENTAL TO ADVANCED MICRO DEVICES, INC. OR ANY COPYRIGHT HOLDERS OR CONTRIBUTORS FOR ANY REASON, THEN ALL RIGHTS TO REDISTRIBUTE, ACCESS OR USE THIS MATERIAL SHALL TERMINATE IMMEDIATELY. MOREOVER, THE FOREGOING SHALL SURVIVE ANY EXPIRATION OR TERMINATION OF THIS LICENSE OR ANY AGREEMENT OR ACCESS OR USE RELATED TO THIS MATERIAL.

NOTICE IS HEREBY PROVIDED, AND BY REDISTRIBUTING OR ACCESSING OR USING THIS MATERIAL SUCH NOTICE IS ACKNOWLEDGED, THAT THIS MATERIAL MAY BE SUBJECT TO RESTRICTIONS UNDER THE LAWS AND REGULATIONS OF THE UNITED STATES OR OTHER COUNTRIES, WHICH INCLUDE BUT ARE NOT LIMITED TO, U.S. EXPORT CONTROL LAWS SUCH AS THE EXPORT ADMINISTRATION REGULATIONS AND NATIONAL SECURITY CONTROLS AS DEFINED THEREUNDER, AS WELL AS STATE DEPARTMENT CONTROLS UNDER THE U.S. MUNITIONS LIST. THIS MATERIAL MAY NOT BE USED, RELEASED, TRANSFERRED, IMPORTED, EXPORTED AND/OR RE-EXPORTED IN ANY MANNER PROHIBITED UNDER ANY APPLICABLE LAWS, INCLUDING U.S. EXPORT CONTROL LAWS REGARDING SPECIFICALLY DESIGNATED PERSONS, COUNTRIES AND NATIONALS OF COUNTRIES SUBJECT TO NATIONAL SECURITY CONTROLS. MOREOVER, THE FOREGOING SHALL SURVIVE ANY EXPIRATION OR TERMINATION OF ANY LICENSE OR AGREEMENT OR ACCESS OR USE RELATED TO THIS MATERIAL.

NOTICE REGARDING THE U.S. GOVERNMENT AND DOD AGENCIES: This material is provided with "RESTRICTED␣RIGHTS" and/or "LIMITED␣RIGHTS" as applicable to computer software and technical data, respectively. Use, duplication, distribution or disclosure by the U.S. Government and/or DOD agencies is subject to the full extent of restrictions in all applicable regulations, including those found at FAR52.227 and DFARS252.227 et seq. and any successor regulations thereof. Use of this material by the U.S. Government and/or DOD agencies is acknowledgment of the proprietary rights of any copyright holders and contributors, including those of Advanced Micro Devices, Inc., as well as the provisions of FAR52.227-14 through 23 regarding privately developed and/or commercial computer software.

This license forms the entire agreement regarding the subject matter hereof and supersedes all proposals and prior discussions and writings between the parties with respect thereto. This license does not affect any ownership, rights, title, or interest in, or relating to, this material. No terms of this license can be modified or waived, and no breach of this license can be excused, unless done so in a writing signed by all affected parties. Each term of this license is separately enforceable. If any term of this license is determined to be or becomes unenforceable or illegal, such term shall be reformed to the minimum extent necessary in order for this license to remain in effect in accordance with its terms as modified by such reformation. This license shall be governed by and construed in accordance with the laws of the State of Texas without regard to rules on conflicts of law of any state or jurisdiction or the United Nations Convention on the International Sale of Goods. All disputes arising out of this license shall be subject to the jurisdiction of the federal and state courts in Austin, Texas, and all defenses are hereby waived concerning personal jurisdiction and venue of these courts.

Listing A.2: `PrefixSum` kernel with both loops specialized optimized for complete mask.

```
__kernel
void prefixSum(__global float * output,
               __global float * input,
               __local  float * block,
               const    uint    length)
{
    int tid = get_local_id(0);
    unsigned int xx = get_global_id(0); // Dummy to prevent assertion in AMD driver.

    int offset = 1;

    /* Cache the computational window in shared memory */
    block[2*tid]     = input[2*tid];
    block[2*tid + 1] = input[2*tid + 1];

    /* build the sum in place up the tree */
    {
        int d = length>>1;
        for(; d >= 4; d >>=1)
        {
            barrier(CLK_LOCAL_MEM_FENCE);

            if(first_local_id(0) + 3 < d)
            {
                int ai = offset*(2*tid + 1) - 1;
                int bi = offset*(2*tid + 2) - 1;

                block[bi] += block[ai];
            }
            offset *= 2;
        }

        for(; d > 0; d >>=1)
        {
            barrier(CLK_LOCAL_MEM_FENCE);

            if(tid < d)
            {
                int ai = offset*(2*tid + 1) - 1;
                int bi = offset*(2*tid + 2) - 1;

                block[bi] += block[ai];
            }
            offset *= 2;
        }
    }

    /* scan back down the tree */

    /* clear the last element */
    if(tid == 0)
```

```
    {
        block[length - 1] = 0;
    }

     {
        /* traverse down the tree building the scan in the place */
        int d = 1;
        for(d = 1; d < 4 && d < length ; d *= 2)
        {
            offset >>=1;
            barrier(CLK_LOCAL_MEM_FENCE);

            if(tid < d)
            {
                int ai = offset*(2*tid + 1) - 1;
                int bi = offset*(2*tid + 2) - 1;

                float t = block[ai];
                block[ai] = block[bi];
                block[bi] += t;
            }
        }

        for(; d < length ; d *= 2)
        {
            offset >>=1;
            barrier(CLK_LOCAL_MEM_FENCE);
            if (first_local_id(0) + 3 < d)
            {
                int ai = offset*(2*tid + 1) - 1;
                int bi = offset*(2*tid + 2) - 1;

                float t = block[ai];
                block[ai] = block[bi];
                block[bi] += t;
            }
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    /*write the results back to global memory */
    output[2*tid]     = block[2*tid];
    output[2*tid + 1] = block[2*tid + 1];
}
```

Listing A.3: `DwtHaar1D` kernel with specialized loop optimized for complete mask.

```
__kernel
void dwtHaar1D(
                __global float *inSignal,
                __global float *coefsSignal,
                __global float *AverageSignal,
```

```
                __local float *sharedArray,
                uint tLevels,
                uint signalLength,
                uint levelsDone,
        uint mLevels)

{
    size_t localId = get_local_id(0);
    size_t groupId = get_group_id(0);
    size_t globalId = get_global_id(0);
    size_t localSize = get_local_size(0);

    /**
     * Read input signal data from global memory
     * to shared memory
     */
    sharedArray[localId * 2] = inSignal[groupId * localSize * 2 + localId * 2];
    sharedArray[localId * 2 + 1] = inSignal[groupId * localSize * 2 + localId * 2 + 1]

    /* Divide with signal length for normalized decomposition */
    if(0 == levelsDone)
    {
       sharedArray[localId * 2] /= sqrt((float)signalLength);
       sharedArray[localId * 2 + 1] /= sqrt((float)signalLength);
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    uint levels = tLevels > mLevels ? mLevels: tLevels;
    uint activeThreads = (1 << levels) / 2;
    uint midOutPos = signalLength / 2;

    for(uint i = 0; i < levels; ++i)
    {
        if (activeThreads < 4)
            break;

        if(localId < activeThreads)
        {
            float data0 = sharedArray[2 * localId];
            float data1 = sharedArray[2 * localId + 1];

            sharedArray[localId] = (data0 + data1) / sqrt((float)2);
            uint globalPos = midOutPos + groupId * activeThreads + localId;
            coefsSignal[globalPos] = (data0 - data1) / sqrt((float)2);

            midOutPos >>= 1;
        }
        activeThreads >>= 1;
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    barrier(CLK_LOCAL_MEM_FENCE);
```

```
    if(localId < activeThreads)
    {
        float data0 = sharedArray[2 * localId];
        float data1 = sharedArray[2 * localId + 1];

        sharedArray[localId] = (data0 + data1) / sqrt((float)2);
        uint globalPos = midOutPos + groupId * activeThreads + localId;
        coefsSignal[globalPos] = (data0 - data1) / sqrt((float)2);

        midOutPos >>= 1;
    }
    activeThreads >>= 1;
    barrier(CLK_LOCAL_MEM_FENCE);

    if(localId < activeThreads)
    {
        float data0 = sharedArray[2 * localId];
        float data1 = sharedArray[2 * localId + 1];

        sharedArray[localId] = (data0 + data1) / sqrt((float)2);
        uint globalPos = midOutPos + groupId * activeThreads + localId;
        coefsSignal[globalPos] = (data0 - data1) / sqrt((float)2);

        midOutPos >>= 1;
    }
    activeThreads >>= 1;
    barrier(CLK_LOCAL_MEM_FENCE);

    /**
     * Write 0th element for the next decomposition
     * steps which are performed on host
     */

     if(0 == localId)
        AverageSignal[groupId] = sharedArray[0];
}
```

Listing A.4: `FastWalshTransform` kernel with optimized memory acccess.

```
__kernel
void fastWalshTransform(__global float * tArray,
                         __const  int    step
                        )
{

        if (step % 4 == 0) {
            int fPair = 2 * step* ( first_global_id(0) / step) + (first_global_id(0) %
            int lane = get_global_id(0) - first_global_id(0);
            float T1           = tArray[fPair + lane];
            float T2           = tArray[fPair + step + lane];

            tArray[fPair + lane]       = T1 + T2;
            tArray[fPair + step + lane]              = T1 - T2;
        } else {
```

```
            unsigned int tid = get_global_id(0);
            const unsigned int group = tid%step;
            const unsigned int pair  = 2*step*(tid/step) + group;
            const unsigned int match = pair + step;
            float T1         = tArray[pair];
            float T2         = tArray[match];

            tArray[pair]             = T1 + T2;
            tArray[match]            = T1 - T2;
        }
}
```

Listing A.5: `BitonicSort` kernel with optimized memory acccess.

```
__kernel
void bitonicSort(__global uint * theArray,
                 const uint stage,
                 const uint passOfStage,
                 const uint width,
                 const uint direction)
{
    uint sortIncreasing = direction;
    uint threadId = get_global_id(0);
    uint firstThreadId = first_global_id(0);

    uint pairDistance = 1 << (stage - passOfStage);
    uint blockWidth   = 2 * pairDistance;

    if (pairDistance % 4 == 0) {
        int lane = get_global_id(0) - first_global_id(0);

        uint leftId = lane
          + (firstThreadId % pairDistance)
          + (first_global_id(0) / pairDistance) * blockWidth;

        uint rightId = leftId + pairDistance;

        uint leftElement = theArray[leftId];
        uint rightElement = theArray[rightId];

        uint sameDirectionBlockWidth = 1 << stage;

        if((threadId/sameDirectionBlockWidth) % 2 == 1)
            sortIncreasing = 1 - sortIncreasing;

        uint greater;
        uint lesser;
        if(leftElement > rightElement)
        {
            greater = leftElement;
            lesser  = rightElement;
        }
        else
        {
```

```
            greater = rightElement;
            lesser  = leftElement;
        }

        // if(sortIncreasing)
        {
            theArray[leftId]  = sortIncreasing ? lesser  : greater;
            theArray[rightId] = sortIncreasing ? greater: lesser;
        }
        //else
        //{
            //theArray[leftId]  = greater;
            //theArray[rightId] = lesser;
        //}

    } else { // default path
        uint leftId = (threadId % pairDistance)
                    + (threadId / pairDistance) * blockWidth;

        uint rightId = leftId + pairDistance;

        uint leftElement = theArray[leftId];
        uint rightElement = theArray[rightId];

        uint sameDirectionBlockWidth = 1 << stage;

        if((threadId/sameDirectionBlockWidth) % 2 == 1)
            sortIncreasing = 1 - sortIncreasing;

        uint greater;
        uint lesser;
        if(leftElement > rightElement)
        {
            greater = leftElement;
            lesser  = rightElement;
        }
        else
        {
            greater = rightElement;
            lesser  = leftElement;
        }

        if(sortIncreasing)
        {
            theArray[leftId]  = lesser;
            theArray[rightId] = greater;
        }
        else
        {
            theArray[leftId]  = greater;
            theArray[rightId] = lesser;
        }
    }
}
```