Automatic SIMD Vectorization of SSA-based Control Flow Graphs

Dissertation zur Erlangung des Grades des

"Doktors der Ingenieurwissenschaften"

der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Ralf Karrenberg, M.Sc.

Saarbrücken, Juli 2014

Dekan / Dean:
Prof. Dr. Markus Bläser, Universität des Saarlandes,
Saarbrücken, Germany

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

Prüfungsausschuss / Committee:

Prof. Dr. Christoph Weidenbach (Vorsitzender / Chairman)

Prof. Dr. Sebastian Hack (Gutachter / Reviewer)

Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm (Gutachter / Reviewer)

Dr. Jörg Herter (Akad. Mitarbeiter / Research Assistant)

Eingereicht am / Thesis submitted:
17. Juli 2014

Tag des Kolloquiums / Date of the colloquium:
30. September 2014

UNIVERSITÄT
DES
SAARLANDES

Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Im Stadtwald - Building E1.1
66123 Saarbrücken

Foreword

I first met Ralf four years ago on stage at the magnificent "Le Majestic" congress center in Chamonix, France. This was in April 2011, and we switched connections at the podium as Ralf completed his talk on *Whole-Function Vectorization* at CGO 2011 and I was about to start my talk. Four years later, it is a pleasure to write this foreword to Ralf's PhD thesis on this subject, which significantly extends and builds upon his earlier works. Ralf's works have already attracted attention and inspired research and development in both academia and industry, and this thesis will undoubtedly serve as a major reference for developers and researches involved in the field.

Ralf's original *Automatic Packetization* Master's thesis from July 2009 has inspired early OpenCL tools such as the *Intel OpenCL SDK Vectorizer* presented by Nadav Rotem at LLVM's November 2011 Developer's Meeting. Ralf Karrenberg and Sebastian Hack's CC 2012 paper on *Improving Performance of OpenCL on CPUs* further extended their CGO 2011 paper. Recently, Hee-Seok Kim et al. from UIUC refer to the aforementioned papers in their CGO 2015 paper on *Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures*, which effectively maximizes the vectorization factors for kernel functions where possible and profitable. In addition, Yunsup Lee et al. from UC Berkeley and NVIDIA also refer to the aforementioned papers in their recent Micro-47 paper on *Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures*, which compares software and hardware techniques for supporting divergence on GPU architectures. These recent publications exemplify the contributions Whole-Function Vectorization already has in spurring additional research and development, and will surely continue to have, harnessing new potentials of future SIMD architectures.

Among the new vectorization challenges and opportunities presented in this thesis, the use of Whole-Function Vectorization to handle explicit and partially vectorizable loops, as explained in Section 8.4, are particularly noteworthy. It effectively complements the classical approach of distributing loops to isolate their vectorizable parts. Such opportunities are expected to become applicable to more general-purpose language extensions such as OpenMP starting from its recent 4.0 release. This is in addition to the data-parallel and shading languages such as OpenCL which it originally targets effectively, as Sections 8.2 and 8.3 demonstrate. Another direction forward aims to leverage the techniques of this thesis within the standard C language, as we argue in our upcoming PLC 2015 paper on *Streamlining Whole Function Vectorization in C using Higher Order Vector Semantics*.

Heuristics and possibly machine-learning techniques are among the suggested future work directions listed in Chapter 10, striving to navigate the tradeoffs involved in linearization and specialization transformations efficiently. Indeed, the innovative foundations of Rewire Target Analysis and Partial CFG Linearization laid clearly in Chapters 5 and 6 pose new interesting optimization challenges potentially involving code duplication. Moving forward, extending the scope of vectorization analyses and transformations across control flow and inter-procedural boundaries as presented in this thesis, will most likely continue to have direct impact on compilation tools and programming environments.

Finally, it is worth noting the deliberate and constructive use of state-of-the-art open source software in demonstrating and disseminating the techniques of Whole-Function Vectorization. Namely, using the LLVM Compiler Infrastructure with its SSA vector intermediate representation, as presented at LLVM's April 2012 Developer's Meeting. Doing so provides a valuable and practical implementation that complements this research thesis. It also facilitates artifact evaluation, something which the Compiler Design Lab at Saarland University has excelled at, and is yet to become standard practice in our community.

Haifa, Israel Ayal Zaks

Acknowledgement

First and foremost, I want to express my gratitude towards my advisor Prof. Dr. Sebastian Hack. Not only did he give me the opportunity to pursue a PhD in his group, he also gave me all the necessary freedom, boundaries, and advice to finally succeed.

Furthermore, I would like to thank Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm for reviewing this thesis, and Prof. Dr. Christoph Weidenbach and Dr. Jörg Herter for serving on my committee.

I also want to offer my special thanks to Prof. Dr. Philipp Slusallek. He first introduced me to working in research during my undergraduate studies and since then has accompanied—with his typical, contagious enthusiasm—most of the projects that I have been involved in.

Another special thanks go to Dr. Ayal Zaks for a lot of valuable feedback and for the honor of writing the foreword.

It is a pleasure to thank the following people that I met during my time at university: Marek Košta, Thomas Sturm, Roland Leißa, Marcel Köster, Dmitri Rubinstein, Thomas Schaub, and Simon Moll for great team work; Christoph Mallon, for his help with too many problems to write down anywhere; Sven Woop and Heiko Friedrich, my first advisors at university during my time at the computer graphics chair; Fred, Peter, Wadim, Helge, Michel, and Billy (in order of appearance) for the countless hours of beachvolleyball during lunch break; Flo for all the non-beachvolleyball lunch breaks.

I would also like to extend my thanks to Thomas Schaub, Kevin Streit, Roland Leißa, Viola Bianco, and Christoph Mallon, who helped to improve this thesis by proof-reading and giving feedback.

Finally, I am indebted to all my friends, in Saarbrücken as much as in Oberhöchstadt, San Francisco, Paris, or anywhere else; and to my parents Ilse and Bernhard and my sister Sonja. I could never have gotten to the point of even starting a PhD without you providing friendship, support, and distraction from work.

Most importantly, however, I owe my deepest gratitude to my wife Inge: For your support, your unlimited source of confidence in me, and all the time I was and will be fortunate to spend with you.

Berlin, Germany Ralf Karrenberg

Abstract

Applications that require the same computation to be performed on huge amounts of data play an important role in today's computing landscape: Particle simulations, stock option price estimation, and video decoding are some examples for such data-parallel programs. Languages like OpenCL and CUDA facilitate their development. They allow to write a scalar function that is executed on different input values in parallel by a runtime system. To unleash the full parallel potential of modern processors, the compiler and runtime system of these languages then have to make use of all available cores as well as *SIMD instructions*: These allow to efficiently execute a single operation on multiple input values at once per core.

This thesis presents *Whole-Function Vectorization* (WFV), an approach that allows a compiler to automatically exploit SIMD instructions in data-parallel settings. Without WFV, one processor core executes a single instance of a data-parallel function. WFV transforms the function to execute multiple instances at once using SIMD instructions.

For simple, straight-line code, the transformation is easily applied and delivers drastically improved performance. However, problems such as particle simulations or shading in computer graphics exhibit more complex code. We show that in such scenarios, a naïve WFV approach will often not improve performance or even slow down execution. The focus of this thesis is to present an advanced WFV algorithm that includes a variety of analyses and code generation techniques that help to improve the generated code.

An implementation of WFV has been evaluated in different settings: First, in a stand-alone OpenCL runtime system. Second, in a compiler for domain-specific languages used in real-time ray tracing. Third, in a compiler that performs classic loop vectorization upon request by the user. In all scenarios, WFV improves performance compared to state-of-the-art approaches. The performance of the OpenCL implementation is on par with the proprietary Intel driver, and faster than any other available CPU driver.

Kurzfassung

Anwendungen wie Partikelsimulationen oder die Optionspreisbewertung an Aktienmärkten, die gleiche Berechnungen auf eine Vielzahl von Daten anwenden, sind ein wichtiger Aspekt der heutigen Informatik. Um das Erstellen solcher Anwendungen zu vereinfachen wurden datenparallele Programmiersprachen wie OpenCL und CUDA entwickelt. Diese ermöglichen es, ein skalares Programm zu schreiben, das von einem Laufzeitsystem parallel für verschiedene Eingabewerte ausgeführt wird. Um das vollständige Potenzial heutiger Prozessoren auszunutzen, müssen der Übersetzer und das Laufzeitsystem sowohl alle verfügbaren Kerne als auch *SIMD Befehle* verwenden: Letztere führen dieselbe Operation effizient für mehrere Eingabewerte gleichzeitig aus.

Die vorliegende Arbeit beschreibt *Whole-Function Vectorization* (WFV), einen Ansatz, der es dem Übersetzer erlaubt, in einem datenparallelen Kontext automatisch SIMD Instruktionen zu verwenden. Ohne WFV wertet jeder Prozessorkern eine einzelne Instanz einer Funktion aus. WFV transformiert die Funktion so, dass sie mit Hilfe von SIMD Befehlen mehrere Instanzen auf einmal auswertet. Diese Transformation ist für einfachen, verzweigungsfreien Programmcode leicht durchzuführen und bringt drastische Laufzeitverbesserungen. Probleme wie Partikelsimulationen jedoch verwenden komplexeren Code. Häufig verbessert dann ein naiver WFV Ansatz die Laufzeit nicht oder verschlechtert sie sogar. Die vorliegende Arbeit beschreibt einen Algorithmus für WFV, der neue Analysen und Techniken zur Codeerzeugung verwendet, die die Performanz des Programms verbessern.

WFV wurde in unterschiedlichen Anwendungsgebieten evaluiert: Erstens in einem OpenCL Laufzeitsystem. Zweitens in einem Übersetzer für domänenspezifische Sprachen für Echtzeit-Ray-Tracing. Drittens in einem Übersetzer, der klassische Schleifenvektorisierung durchführt. In allen drei Szenarien zeigt die Auswertung Verbesserungen der Laufzeit bei Verwendung von WFV im Vergleich mit dem neuesten Stand der Technik. Das OpenCL Laufzeitsystem steht auf einer Stufe mit dem äquivalenten Produkt von Intel und ist effizienter als jeder andere CPU-basierte Ansatz.

Contents

1 Introduction

Data-parallel applications play an important role in today's computing landscape, especially in the High-Performance Computing (HPC) area. Particle simulations, stock option prediction, medical imaging, or video encoding and decoding are just a few problems that can be formulated in a data-parallel way: They each require to do similar computations for large amounts of input data, with no or only limited dependencies between the computations of different inputs.

Due to the amount of data that needs to be processed, the performance of the application code is critical. To reach maximum performance, the available data-parallelism needs to be exploited. The best possible results can always be achieved by manually tuning an application to a specific target architecture. However, this usually comes at the cost of development time, error-freedom, maintainability, and portability.

To balance these aspects without sacrificing too much performance, domain-specific languages (DSLs) are used. Well-known examples are shading languages like RenderMan in computer graphics or data-parallel languages like OpenCL and CUDA. In particular, the latter two aim to provide portability alongside an easy-to-use programming model by abstracting from the concrete hardware: The user writes a scalar function, a so-called *kernel*, and the runtime system executes it many times with different input data. The choice of how to best execute these *instances* of the function on a given machine is left to the runtime system and compiler. The data-parallel semantics explicitly allow to run instances in parallel, with only few restrictions to allow for synchronization when required. Listing 1.1 shows an example for a kernel that computes an image of the Mandelbrot set. [1]

To exploit the full parallel potential of modern processors, a runtime system has to use both multi-threading and *vector instructions*: instructions that allow to execute an operation on multiple sets of input values at once. Because of this property, they are also called *SIMD* instructions, for "Single Instruction, Multiple Data." Such an operation requires the same amount of time as its scalar counterpart would require for a single set of input values.

[1]Implementation from the ATI Stream SDK (now AMD APP SDK) version 2.1.

Listing 1.1 Kernel function for a Mandelbrot application (OpenCL). The loop exit conditions were moved to the body to show the connection with the control flow graph of Figure 1.1. Letters in comments denote basic blocks of this graph.

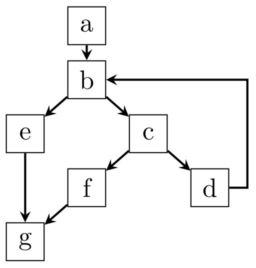
```
__kernel void
Mandelbrot(__global int*  image,
           const    float scale,
           const    uint  maxIter,
           const    int   width)
{
    /* a */
    int tid = get_global_id(0);

    int i = tid%width;
    int j = tid/width;

    float x0 = ((i*scale) - ((scale/2)*width))/width;
    float y0 = ((j*scale) - ((scale/2)*width))/width;

    float x = x0;
    float y = y0;

    float x2 = x*x;
    float y2 = y*y;

    float scaleSq = scale * scale;

    uint iter=0;
    for (;;)
    {
        /* b */
        if (iter >= maxIter)
        {
            /* e */
            break;
        }
        /* c */
        if (x2+y2 > scaleSq)
        {
            /* f */
            break;
        }
        /* d */
        y  = 2 * x * y + y0;
        x  = x2 - y2   + x0;
        x2 = x*x;
        y2 = y*y;
        ++iter;
    }
    /* g */
    image[tid] = 255*iter/maxIter;
}
```

The number of inputs that can be processed in parallel is described by the *SIMD width* S. The value for S is derived from the number of single-precision (32 bit) floating point values that can be stored in one vector register of the specific SIMD instruction set. We say that such a SIMD register has S *SIMD lanes*. S is currently 4, 8, or 16 for most CPUs and 32 or 64 for GPUs. For example, Intel's SSE instruction set provides vector registers of 16 bytes, yielding a SIMD width of 4. For applications that use only single-precision operations, exploiting SIMD instructions can provide a speed up factor of up to S. This huge performance potential has traditionally been used mostly in manual vectorization of compact, performance-critical parts of code and automatic loop vectorization of simple loops.

This thesis presents *Whole-Function Vectorization* (WFV), a compiler-based, automatic approach to exploit SIMD instructions in data-parallel settings. Without WFV, one processor core would execute a single instance of a data-parallel function, relying solely on multi-threading to exploit inter-core parallelism. WFV transforms the function to execute a *SIMD group* of W instances at once. W is the *vectorization factor*, and is usually a multiple of the SIMD width S. The values of each of the W instances are kept in one SIMD lane. This way, each processor core executes W instances, exploiting the intra-core parallelism provided by SIMD instruction sets in addition to multi-threading. If W is larger than S, or if a value is larger than 32 bit, multiple SIMD registers are required for the grouped instances.

To illustrate how the algorithm operates, consider the example in Figure 1.1. The control flow graph (CFG) on the left represents the `Mandelbrot` kernel shown in Listing 1.1. It is applied to four different inputs in parallel. This results in four different execution traces as shown in the table on the right. For example, instance 1 executes the blocks `a b c d b c d b e g` while instance 3 executes the blocks `a b c f g`.

Divergent control flow makes SIMD execution more complicated: Instances 1, 2, and 4 iterate the loop again while instance 3 leaves it directly at block `b`. Also, instances 1 and 4 leave the loop from block `b`, whereas the other two instances take the exit at block `c`. Hence, the instances also execute different blocks after they left the loop. SIMD execution of all instances grouped together is not possible because the loop is either left or iterated again. The usual solution is that the SIMD program iterates loops as long as any of the instances that entered would, executes *both* exit blocks, and compensates for the unwanted effects: Instance 3 ignores all iterations of the loop, instance 2 ignores the last iteration, instances 1 and 4 ignore the computations of block `f`, and so on. This is called *predicated execution* [Park & Schlansker 1991].



Instance	Trace
1	`a b c d b c d b e f g`
2	`a b c d b c d b e f g`
3	`a b c d b c d b e f g`
4	`a b c d b c d b e f g`

Figure 1.1: The control flow graph of the `Mandelbrot` kernel from Listing 1.1 and four execution traces. A greyed out block denotes that the corresponding instance is inactive if all instances are executed together in SIMD fashion.

In practice, this behavior is implemented in different ways. GPUs perform predicated execution implicitly in hardware: Each instance of the data-parallel program is mapped to a scalar core in a multi-core processor. Groups of W instances are executed together, i.e., the entire group executes one instruction before the control jumps to the next one. This so-called *lock step* execution is a key aspect of data-parallel execution in general: Compared to sequential or fully concurrent execution of the program, it improves memory access and cache behavior and is more energy efficient due to requiring only one instruction dispatch for the entire group. A central control unit stalls a processor on code regions where its instance is inactive.

The Whole-Function Vectorization algorithm is designed to map data-parallel languages onto processors that require *explicit* vectorization via SIMD instructions. To use SIMD instructions in the presence of divergent control flow as shown above, the code has to be transformed to allow for predicated execution. This is achieved by replacing all control flow by data flow [Allen et al. 1983]. The information which instances are active or inactive in which part of the code is obtained through *predicates* (also called *masks*). These hold one bit of information for each instance. If the bit is set during execution of some code, the corresponding instance is active and results are valid. If the bit is not set, all results for this instance have to be discarded and side effects prevented.[2] For code without loops, this is commonly called *if conversion*. For example, consider the function `f` in Listing 1.2 and its version without control flow `f_df`. The function `select`

[2]Note that in the setting of data-parallel execution, elapsed time is not considered a side effect.

Listing 1.2 A scalar function with control flow (`f`). The same function with control flow replaced by data flow and optimized predication (`f_df`). A vector implementation of the latter using SSE intrinsics (`f_sse`).

```
float f(float a, float b) {
  float r;
  if (a > b) r = a + 1;
  else       r = a - 1;
  return r;
}
```

```
float f_df(float a, float b) {
  bool mask = a > b;
  float s = a + 1;
  float t = a - 1;
  float r = select(mask, s, t);
  return r;
}
```

```
__m128 f_sse(__m128 a, __m128 b) {
  __m128 m = _mm_cmpgt_ps(a, b);
  __m128 s = _mm_add_ps(a, _mm_one);
  __m128 t = _mm_sub_ps(a, _mm_one);
  __m128 r = _mm_blendv_ps(mask, s, t);
  return r;
}
```

chooses, dependent on the mask value, either `s` or `t`. We also call the `select` a *blend* operation, since in the vector program it blends two vectors to form a new one. The function `f_sse` in Listing 1.2 shows how a C/C++ programmer would implement a vectorized variant of the same code using intrinsics for Intel's SSE instruction set.

Operations that may have side effects deserve additional attention during vectorization: The SIMD function is required to produce the same side effects as W applications of the scalar function. However, the order of occurrence of these side effects may be different: WFV preserves the order of side effects per operation, but not between operations. This means that the side effect of instruction a for instance 0 will be observed before the side effect of a for instance 1. However, due to lock step execution, the side effects of instruction a of *all* instances will be observed before the side effects of any subsequent instruction. An implication of this is that race conditions may appear between different instances of the same function. Data-parallel languages such as OpenCL and CUDA explicitly allow such behavior to enable the compiler to be more aggressive. If desired, synchronization thus has to be enforced on a higher level of abstraction (more on this in Section 8.2).

There exist several approaches that convert control flow into data flow to perform vectorization (Chapter 4 discusses related work in further detail).

Those approaches typically originate from the parallel computing community where parallelization is performed early in the compilation process; often already at the source level. Because this has two major disadvantages, we argue that vectorization should be performed late in the compilation process:

1. By transforming control flow to data flow too early, all analyses and optimizations in the compiler that make use of control flow information are rendered useless. These analyses (for example conditional constant propagation or loop invariant code motion) would need to be rewritten to be able to analyze the predicated vector code.

2. Modern compiler infrastructures use virtual instruction sets as a code exchange format (for example LLVM bitcode [Lattner & Adve 2004] or Nvidia's PTX[3]) to decouple front ends from code generators. Those representations commonly use control flow graphs of instructions to represent the code. Performing control-flow to data-flow conversion already in the front end destroys the portability of the generated code for two reasons: First, the transformation would need to be undone to use the code for architectures that do not require control-flow to data-flow conversion such as GPUs. Second, even if the target architecture requires control-flow conversion, the front end would need to address architectural parameters like the SIMD width and details about the implementation of predicated execution.

Whole-Function Vectorization performs control-flow to data-flow conversion on code represented in modern intermediate representations such as LLVM bitcode. More specifically, WFV is a program transformation on control flow graphs in Static Single Assignment (SSA) form.

Coming back to Figure 1.1, one can observe that SIMD execution requires overhead to maintain correct behavior of the data-parallel program. As long as any instance is active, the loop has to keep iterating, and all inactive instances do nothing. We say that such a loop is *divergent* because not all instances leave at the same time and/or over the same exit.[4] In this example with $W = 4$, only 33 out of 44 block executions do work that contributes to the final results of the instances. Compared to the sequential execution, this still translates to a significant performance improvement: Assume, for simplicity, that execution of each block takes equally long. Then, the SIMD code is 3 times faster, since it exhibits a cost of 11 instead of 33.

[3] `docs.nvidia.com/cuda/parallel-thread-execution`

[4] Note that a *divergent loop* in the SIMD context refers to a loop in which different instances may choose different exits and/or iteration counts, not a loop that never terminates.

However, it is possible to improve this situation. For diverging loops, the reduced efficiency due to inactive instances cannot be avoided unless some mechanism reorders the instances that are grouped together (discussed further in Section 4.6 and Chapter 7). Nonetheless, looking at the loop exit conditions of the `Mandelbrot` kernel, we can see that if `iter` reaches a threshold, the loop will be left in block `b`. This threshold is the same for all instances, so we know that *if* this exit is taken, it is taken by *all* instances that are still active. For the traces in Figure 1.1, this does not make a difference, since there are instances that leave over both edges, so both blocks have to be executed. However, if the loop is left from the other exit in block `c`, block `e` does not have to be executed. This is because no instance can have left from block `b`, otherwise all instances would have left there. In Chapter 8, we show that this has indeed a noticeable impact on performance.

In general, such cases occur if the condition of a branch only depends on values which are the same for all grouped instances. We call such a value *uniform*, whereas a value that holds different values for different instances is *varying*. One way to exploit this is to inject code into the vector program that tests *dynamically* whether all instances evaluate a condition to the same value [Shin 2007]. If so, the corresponding code part can be bypassed by a branch, trading a reduction of the amount of executed code for the overhead of the dynamic test. Unfortunately, this does not solve all problems related to divergent control flow, as is detailed in Chapter 5. Also, the dynamic test introduces additional overhead, so a carefully crafted heuristic is required for this approach to work well.

This thesis introduces the notion of *divergence-causing blocks* and *rewire target blocks* that capture the behavior of control flow in a SIMD context. We further present a static analysis that determines the rewire targets of each divergence-causing block in a CFG, and an improved, *partial* control-flow to data-flow conversion algorithm that makes use of this information. The presented approach is an overapproximation, i.e., it cannot cover as many cases as a dynamic approach because some occurrences of non-diverging control flow can be caused by input data. However, since the algorithm only uses static information, it never impacts performance negatively in contrast to the dynamic test.

1.1 Contributions

In summary, this thesis makes the following contributions:

- We present Whole-Function Vectorization (WFV), a transformation of control flow graphs in SSA form for processors with SIMD instructions. Existing vectorization approaches are tied to the source language and often also to the target platform. A CFG in SSA form is an intermediate representation that abstracts from *source language* and *target machine*. Thus, WFV improves portability and maintainability.

- WFV generates predicated code for *arbitrary* control flow on architectures without hardware support for predicated execution by carefully placing `select` operations. This includes nested loops, loops with multiple exits, and exits leaving multiple loops. Therefore, WFV allows to vectorize a larger class of applications than state-of-the-art approaches, which are usually restricted to structured or non-cyclic control flow.

- We present a static analysis that identifies certain constraints on the values of variables in different parallel instances: This *Vectorization Analysis* determines which values are the same for all parallel instances, which values require sequential execution (e.g. due to side effects), which program points are always executed by all instances, which loops may have instances leaving in different iterations or over different exits, and classify memory access patterns.

- We introduce an extension to this analysis that improves the precision of the determined memory access patterns. The extension formulates the problem whether a memory operation accesses consecutive memory locations as a set of equations in Presburger Arithmetic. An SMT solver is used to solve the equations.

- We present a static analysis that identifies certain constraints on the behavior of control flow in a SIMD context: This *Rewire Target Analysis* determines which parts of the control flow graph must never be skipped to ensure correct results during SIMD execution. Implicitly, this describes parts of the CFG which can be left untouched.

- The results of the Rewire Target Analysis are leveraged by a novel, *partial control-flow to data-flow conversion* algorithm. This algorithm keeps regions of the control flow graph intact in which instances never diverge. Existing approaches either linearize the entire CFG, or at most employ simple optimizations such as retaining diamond-shaped `if` statements if the branch condition is uniform. Our approach is able to determine on a per-edge basis which parts of the CFG do not have to be linearized, even for unstructured control flow. For example, it can retain uniform exits of divergent loops and parts of nested and overlapping regions. This reduces the amount of code that has to be executed by every call to the generated function.

- We present a set of extensions to WFV that exploit dynamic properties of a function. This is achieved using dynamic tests and separate, optimized code paths where the tested property is used to generate better code.
- We implemented WFV using the LLVM compiler infrastructure and evaluated it in three practically relevant scenarios. First, we integrated it into a compiler for the RenderMan shading language, a computer graphics DSL that is widely used for visual effects in the movie industry, and performed experiments with a real-time ray tracer. Second, we developed a prototypical OpenCL runtime system that uses WFV to execute multiple work items at once per core. The implementation also includes a novel, software-based scheme for barrier synchronization. Third, we implemented a classic loop vectorization phase on top of WFV. We evaluated it in a compiler that allows the user to define which optimizations to run on what parts of the code. The loop vectorization employs a novel method for handling arbitrary loop-carried dependencies, which includes schemes commonly called *reductions*. We show an average speedup of 3.8 for the ray tracer, speedups between 0.8 and 3.8 for different OpenCL applications, and speedups of 1.2 and 1.45 for applications tuned with the custom compiler on a machine with $S = 4$.

1.2 Publications

The analyses and transformations presented in this thesis build on the following publications:

- **Whole-Function Vectorization**
  Ralf Karrenberg and Sebastian Hack
  In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society.
  [Karrenberg & Hack 2011]

- **Improving Performance of OpenCL on CPUs**
  Ralf Karrenberg and Sebastian Hack
  In *Proceedings of the 21st International Conference on Compiler Construction*, pages 1–20, Berlin, Heidelberg, 2012. Springer-Verlag.
  [Karrenberg & Hack 2012]

- **Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages**
  Ralf Karrenberg, Marek Košta, and Thomas Sturm
  In *Frontiers of Combining Systems*, Volume 8152 of *Lecture Notes in Computer Science*, pages 56–70, Berlin, Heidelberg, 2013. Springer-Verlag.
  [Karrenberg et al. 2013]

- **AnySL: Efficient and Portable Shading for Ray Tracing**
  Ralf Karrenberg, Dmitri Rubinstein, Philipp Slusallek, and Sebastian Hack
  In *Proceedings of the Conference on High Performance Graphics*, pages 97–105, Aire-la-Ville, Switzerland, 2010. Eurographics Association.
  [Karrenberg et al. 2010]

2 Foundations & Terminology

This thesis builds on a few key developments in the field of compiler construction that we briefly introduce in this chapter. More detailed explanations are given, for example, by Aho et al. [2006] and Kennedy & Allen [2002].

2.1 Basic Block

A *basic block* consists of a list of instructions that have to be executed in order. This implies that any operation that can influence which instruction is executed next, i.e., any branch instruction, may only occur at the end of a block.

2.2 Control Flow Graph (CFG)

A CFG is a directed, possibly cyclic graph with a dedicated entry node. It represents the structure of a function by expressing instructions or basic blocks as nodes and their successor relations as edges. We consider CFGs with basic blocks as nodes. If the function code at one point allows multiple different code paths to be executed next, the corresponding block in the CFG has multiple outgoing edges. This is the case for explicit control flow constructs like `if` statements or loops, but can also occur in other circumstances, e.g. for operations that can throw an exception. If previously disjoint paths merge again, the corresponding block has multiple incoming edges, e.g. behind a source-level `if` statement. Figure 2.1 shows the CFG that corresponds to the `Mandelbrot` example in the introduction.

Critical Edges. An edge of a CFG is said to be *critical* if its source block has multiple outgoing edges and its target block has multiple incoming edges. For example, the edge $a \rightarrow c$ in Figure 2.2 is critical. Such edges are usually not desired, and can be easily removed by splitting the edge and placing an additional block at the split point. All algorithms presented in this thesis can handle critical edges. The Partial CFG Linearization phase (Section 6.3) in some cases performs implicit breaking of critical edges.

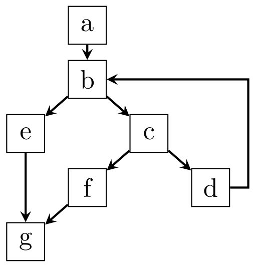


Figure 2.1: The CFG that corresponds to the `Mandelbrot` function in Listing 1.1. Special loop blocks according to Section 2.4: Block a is the entry block and at the same time the preheader of the loop that consists of blocks b, c, and d. Block b is the header of this loop, d is a loop latch. The blocks b and c are loop exiting blocks, e and f the corresponding exit blocks. Block g is the final block that holds a `return` statement.

2.3 Dominance and Postdominance

A basic block a *dominates* another block b if a is part of every possible path from the entry block of the function to b [Lowry & Medlock 1969]. In particular, the entry block of a function dominates all reachable blocks. Also, according to the definition in Section 2.4, a header of a loop dominates all blocks in the loop. A basic block a *strictly* dominates a block b if a dominates b and a and b are distinct blocks. A basic block a is the unique *immediate dominator* of a block b if a strictly dominates b and does not strictly dominate any other strict dominator of b.

The immediate dominance relation of basic blocks induces a tree structure, the *dominator tree*. The *dominance frontier* of a block a is the set of nodes B of which a predecessor of each block in B is dominated by a but each block in B is not strictly dominated by a. The *iterated dominance frontier* of blocks $a_1, \dots, a_n$ is the limit of the increasing sequence of sets of nodes D, where D is initialized with the dominance frontiers of $a_1, \dots, a_n$, and each iteration step increases the set with the dominance frontiers of all nodes of D [Cytron et al. 1991].

A basic block b *postdominates* another block a if b is part of every possible path from b to the function exit block.[1] As an example, if there is exactly

[1]If the function has multiple exit blocks and/or non-terminating loops, we assume a *virtual*, unique exit block that all these blocks jump to instead of returning.

one edge in a loop that jumps back to the header, the source block of that edge is a postdominator of all blocks in the loop (see Section 2.4). The terms *strict postdomination*, *immediate postdominator*, *postdominator tree*, *postdominance frontier*, and *iterated postdominance frontier* are defined analogously.

Various algorithmic approaches exist to efficiently compute dominance, dominance frontiers, and dominator trees [Cytron et al. 1991, Lengauer & Tarjan 1979, Ramalingam 2002].

Note that these definitions are straightforward to extend from basic blocks to SSA values: If two values are defined in the same basic block, the upper one in the list of instructions strictly dominates the lower one. Otherwise, the dominator relation of their parent blocks is the dominator relation of the values.

2.4 Loops

A loop is a strongly connected component of blocks in a CFG. The blocks that are enclosed in such a region are called the *body* of the loop. In addition, we use the following definitions [Ramalingam 2002]:

- A *header* is a block of a loop that is not dominated by any other block of the loop.
- A *preheader* is a block outside the loop that has an unconditional branch to a loop header.
- A *back edge* is an edge that goes from a block of a loop to a loop header.
- A *latch* is a block of a loop that has a back edge as one of its outgoing edges.
- An *exiting* block of a loop is a block inside the loop that has an outgoing edge to a block that is outside the loop.
- An *exit* block of a loop is a block outside the loop that has an incoming edge from a block that is inside the loop.

Note that, by definition, an exiting block has to end with a conditional branch, since otherwise it would not be part of the loop itself. Figure 2.1 exemplifies these terms at the example of the `Mandelbrot` CFG.

If not mentioned otherwise, we consider only *reducible* loops, which are the vast majority of loops found in real-world programs. In a reducible loop, the header dominates (Section 2.3) its source, the latch [Hecht & Ullman 1972]. This implies that a reducible loop may not have more than one header, i.e., it is a loop with a single entry. The single header then also dominates all blocks of the loop. Note that, if required, irreducible control flow can be transformed into reducible control flow using node splitting [Janssen & Corporaal 1997]. However, this can result in an exponential blowup of the code size [Carter et al. 2003].

In addition, we consider all loops to be *simplified*, i.e., they have exactly one preheader and one latch, and all exits have a unique successor block that has no other incoming edges. This does not impose any restrictions on the source code. This simplified definition allows to determine a tree-shaped hierarchy of *nested* loops for every CFG:[2] A loop L_n is *nested* in another loop L_p if the header of L_n is part of the loop body of L_p. In such a case, we call L_p the *outer loop* or *parent loop*, and L_n the *inner loop* or *child loop*. We also say that L_n is at a *nesting level* that is one level *deeper* than L_p. Multiple loops may be nested in one loop, but a loop may not be nested in multiple loops. Note that this definition allows exit edges to leave multiple loops at once, going up multiple nesting levels. However, an entry edge may not enter a loop of a nesting level deeper than the one below its own nesting level.

2.5 Static Single Assignment (SSA) Form

Static Single Assignment form [Alpern et al. 1988], short *SSA* form, is a program representation in which each variable has exactly one static definition. To transform a given program into SSA form, an SSA variable is created for *each* definition of a variable. Figure 2.2 shows an example for the transformation of a normal CFG into a CFG in SSA form. To represent situations where multiple definitions of the same original variable can reach a use, so-called ϕ-functions have to be inserted at dominance frontiers [Cytron et al. 1991]. A ϕ-function has an incoming value for every incoming edge. The operation returns the value that corresponds to the edge over which its basic block was entered. This is required to ensure that the original relations between definitions and uses are preserved while maintaining the SSA property that every use has exactly one definition.

[2]Note that this definition requires the detection of nested SCCs, which can be achieved by ignoring back edges.

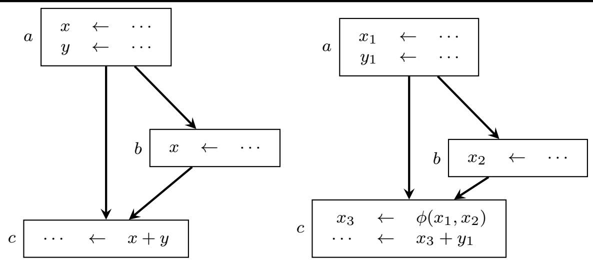


Figure 2.2: A CFG and its SSA counterpart

2.5.1 LCSSA Form (LCSSA)

We consider all CFGs to be in LCSSA form, an extension to SSA form that was first introduced by Zdenek Dvorak[3] in the GCC compiler.[4] LCSSA guarantees that for a value that is defined in a loop, every use that is outside the loop is a ϕ-function in the corresponding loop exit block. This simplifies handling of loop result values during the different phases of the WFV algorithm.

2.6 Control Dependence

A statement y is said to be *control dependent* on another statement x if (1) there exists a nontrivial path from x to y such that every statement $z \neq x$ in the path is postdominated by y, and (2) x is not postdominated by y [Kennedy & Allen 2002].

This definition implies that there can be no control dependencies within a basic block. Thus, it makes sense to talk about control dependence relations of basic blocks. For example, in Figure 2.1, blocks e and c are control dependent on b, and blocks b, f, and d are control dependent on c. Note that blocks can have multiple control dependencies, e.g. in nested conditionals.

2.7 Live Values

The classic definition of a *live variable* in a non-SSA context is the following [Kennedy & Allen 2002]: A variable x is said to be *live* at a given point s

[3]`http://gcc.gnu.org/ml/gcc-patches/2004-03/msg02212.html`
[4]`gcc.gnu.org`

in a program if there is a control flow path from s to a use of x that contains no definition of x prior to the use. Most importantly, this includes program points where the variable is not yet defined. In our context, we are only interested in the live values that are already defined at program point s: An SSA value v is said to be *live* at a given point s if there is a control flow path from v to s and there is a control flow path from s to a use of v that contains no definition of v. For example, in the CFG on the right of Figure 2.2, y_1 is live in block b.

2.8 Register Pressure

Every processor has a fixed set of registers where values are stored. For convenience, programming languages usually abstract from this by providing an unlimited set of registers to the user. The compiler then has to determine how to best map the used registers to those available in hardware. This is called *register allocation*.

Register pressure describes the number of registers that are required for a given program. If there are not enough registers, values have to be *spilled*, i.e., stored to memory, and reloaded later when they are used again. This has implications for the performance of the program: register access is much faster than accessing memory. Thus, it is desirable for a compiler to optimize the usage of registers and minimize the number of spill and reload operations and their impact on performance. Especially in a GPU environment, register pressure is a critical variable: Since the available registers are shared between threads, the number of registers used by a function determines how many threads can execute it in parallel.

In consequence, if a code transformation increases register pressure, this may result in decreased performance even if the resulting code itself is more efficient. As will be discussed in Section 3.3, this is an important issue for SIMD vectorization.

2.9 LLVM

The compiler infrastructure *LLVM* [Lattner & Adve 2004] forms the basis for our implementation of the Whole-Function Vectorization algorithm. Amongst other tools and libraries, the framework includes:

- a language and target independent, typed, intermediate representation in the form of a control flow graph in SSA form,

- a front end (Clang) for the C language family (C/C++/ObjC),[5]
- various back ends, e.g. for x86, ARM, or PTX, and
- a just-in-time compiler.

Our implementation works entirely on LLVM's intermediate representation (IR), which allows to use it independently of the source-language and the target-architecture.

2.9.1 Intermediate Representation (IR)

Throughout the next chapters, we show examples in the human-readable representation of the LLVM IR, which we briefly describe in the following.

The LLVM IR is a typed, low-level, assembler-like language. There are neither advanced language features like overloading or inheritance nor control flow statements such as `if-then-else` statements or explicit loops. Instead, the control flow of a function is represented by a CFG whose edges are induced by low-level instructions such as conditional or unconditional branches. The nodes of the CFG are basic blocks with lists of instructions. Each instruction corresponds to exactly one operation and usually also represents an SSA value. The only exception to this are values with return type `void`, such as `branch`, `call`, and `store` operations. Every other value has a name that starts with "`%`" which is referenced everywhere that value is used. A typical instruction looks like this:

```
%r = fadd <4 x float> %a, %b
```

The example shows the LLVM IR equivalent for the SSE2 vector addition intrinsic ADDPS (`_mm_add_ps(__m128 a, __m128 b)` in C/C++). The left-hand side of the expression is the name of the value. On the right hand side, the operation identifier is followed by its operands, each with its type preceding the name. If types are equal, they can be omitted for all operands after the first.

The following code defines a function `foo` with return type `float` and argument types `int` and `float`:

```
define float @foo(int %a, float %b) {
entry:
  %x = fsub float %b, 1.000000e+01
  ret float %x
}
```

The label `entry` marks the only basic block which is followed by the instructions of that block.

[5] `clang.llvm.org`

2.9.2 Data Types

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

Table 2.1 Examples for the most important LLVM data types and their C/C++ counterparts. Note that this list only shows *typical* C/C++ types, since types like `int` are machine dependent.

LLVM Type	C/C++ Type	Explanation
`i1`	`bool`	truth value (`true (1)` or `false (0)`)
`i8`	`char`	single character
`i32`	`int`	32bit integer value
`i64`	`long`	64bit integer value
`float`	`float`	32bit floating-point value
`type*`	`type *`	pointer to type `type`
`i8*`	`void *`	void pointer
`<4 x float>`	`__m128`	vector of 4 32bit floating-point values
`<4 x i32>`	`__m128i`	vector of 4 32bit integer values
`<8 x float>`	`__m256`	vector of 8 32bit floating-point values
`<8 x i32>`	`__m256i`	vector of 8 32bit integer values
`<4 x i1>`	(`__m128`)	mask vector
{ `types` }	`struct` { `types` }	structure of types `types`
[N × `type`]	`type t`[N]	array of size N of type `type`

2.9.3 Important Instructions

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

- *Phi*
  The `phi` instruction is the equivalent to the ϕ-function described in Section 2.5. It chooses a value depending on which predecessor block was executed:

  ```
  %r = phi float [ %a, %blockA ], [ %b, %blockB ]
  ```

  The value of `r` is set to `a` if the previously executed block was `blockA` or to `b` if the previously executed block was `blockB`.

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

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

  The value of `r` is set to `a` if condition `c` is `true` and to `b` otherwise. If the select statement has operands of vector type, a new vector is created by a *blend* operation that merges the two input vectors on the basis of a per-element evaluation of the condition vector (see Section 6.2). The terms “select” and “blend” are thus used interchangeably for the same operation.

- *GetElementPointer (GEP)*
  The `GetElementPointer` instruction returns a pointer to a member of a possibly nested data structure. It receives the data structure and a list of indices as inputs. The indices denote the position of the requested member on each nesting level of the structure. In the following example, the `GEP` instruction (`%r`) creates a pointer to the `float` element of the struct of type `struct.B` that is nested in the struct `%A` and stores 3.14 to that location:

  ```
  %struct.A = type { i8*, i32, %struct.B }
  %struct.B = type { i64, float, i32 }
  ...
  %r = getelementptr %struct.A* %A, i32 0, i32 2, i32 1
  store float 0x40091EB860000000, float* %r, align 4
  ```

  The first index is required to step through the pointer, the second index references the third element of the struct (which is the nested struct) and the third index references the second element of that nested struct. It is important to note that a `GEP` only performs an address calculation and does not access memory itself.

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

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

  The first instruction inserts the `float` value `elem` at position `1` into vector `p`, yielding a new vector SSA value. The second instruction extracts the same `float` from that vector `p2`, yielding a scalar value again.

- *Comparison Operations*
  The `ICmp` and `FCmp` instructions allow to compare values of the same integer or floating point types and return a truth value. If the values are of vector type, the result of the comparison is stored component-wise as a vector of truth values `<W x i1>`.

- *"All-`false`" Vector Comparison*
  In vector programs, one often needs to take a branch if *all* elements of a mask vector (e.g. `<4 x i1>`) are `false`. Since that requires a single truth value to base the decision upon, one cannot employ a vector comparison operation, which returns a vector of truth values. As of LLVM 3.3, the most efficient solution to this is the following IR pattern: Given a vector condition of type `<W x i1>`, sign extend it to `<W x i32>`, bitcast to `i(W*32)`, and compare to zero. For example, when targeting SSE with a target SIMD width of 4, the following IR is recognized by the x86 back end:

  ```
  %ext = sext <4 x i1> %cond to <4 x i32>
  %bc  = bitcast <4 x i32> %ext to i128
  %test = icmp ne, i128 %bc, 0
  br i1 %test, %blockA, %blockB
  ```

  The resulting assembly is an efficient `PTEST` followed by a conditional jump.

In addition, we define two special instructions that do not actually exist in LLVM IR, but help us keep code in listings more compact:

- *Merge*
  The `merge` instruction creates a new vector from a set of input values:

  ```
  %v = merge float %a, %b, %c, %d
  ```

  This is equivalent to the following IR:

  ```
  %v0 = insertelement <4 x float> undef, float %a, i32 0
  %v1 = insertelement <4 x float> %v0,   float %b, i32 1
  %v2 = insertelement <4 x float> %v1,   float %c, i32 2
  %v  = insertelement <4 x float> %v2,   float %d, i32 3
  ```

- *Broadcast*
  The `broadcast` instruction creates a new vector of the given type where each element is a copy of the single operand:

  ```
  %v = broadcast float %a to <4 x float>
  ```

This is equivalent to a special case of `merge`:

```
%v = merge float %a, %a, %a, %a
```

Note that the broadcast in LLVM IR can also be done with an `insert-element` operation followed by a `shufflevector` instead:

```
%av = insertelement <4 x float> undef, float %a, i32 0
%v  = shufflevector <4 x float> %av, <4 x float> undef,
                    <4 x i32> zeroinitializer
```

2.10 Single Instruction, Multiple Data (SIMD)

Single Instruction, Multiple Data (SIMD) describes an execution model that executes a single operation on *vectors* of input values, yielding a result vector, whereas traditional operations only receive single, *scalar* values and produce a scalar result. Most of today's CPUs—even processors of mobile devices such as smartphones—have special SIMD instruction sets. Common examples for such instruction sets are, in order of consumer market appearance at the time of writing:

- Intel MMX (64 bit registers, since 1996)
- AMD 3DNow! (128 bit registers, since 1998)
- Intel Streaming SIMD Extensions (SSE, 128 bit, since 1999)
- Freescale/IBM/Apple AltiVec (also VMX, 128 bit, since 1999)
- ARM NEON (128 bit, since 2005)
- Intel Advanced Vector Extensions (AVX, 256 bit, since 2011)
- Intel Larrabee New Instructions (LRBni, 512 bit, since 2013).
- Intel AVX-512 (512 bit, expected 2015).

These instruction sets operate on SIMD registers that are two to sixteen times larger than a standard, single-precision value of 32 bit. However, the vector instructions only require approximately the same time for execution as their scalar counterparts. Thus, if used efficiently, they can increase performance of an application significantly. In addition, they are more energy efficient since only one instruction fetch and decode has to be performed.

We use the following terms to refer to properties of SIMD instructions and vectorization: The *SIMD width* S of an architecture is the number of 32 bit values that can be stored in a single SIMD register. Such an architecture thus has *S SIMD lanes*, where the *i*-th lane refers to the element at position *i* of the vector. Data-parallel vectorization is achieved by executing an instance of the program in a single SIMD lane and combining *S* instances into a *SIMD group* that is executed together.

3 Overview

In this chapter, we present an overview of the basic *Whole-Function Vectorization* algorithm, the challenges faced when vectorizing real-world code, and the most important performance implications of the transformation. In Chapter 5, we give a detailed description of the analyses involved, followed by the actual presentation of the WFV algorithm in Chapter 6.

3.1 Whole-Function Vectorization (WFV)

The algorithm transforms a given function f into its SIMD equivalent f_W that performs W executions of f in parallel. It consists of five main phases:

1. An analysis phase determines a variety of properties of the function under data-parallel semantics (Chapter 5). For example, it has to be determined which instructions cannot be vectorized and thus have to be replaced by multiple scalar operations instead of a vectorial one. Also, the subsequent phases can produce much more efficient code if operations could be proven to always produce the same result for all grouped instances or whether a memory operation accesses consecutive locations.
2. Masks are computed for every edge of the control flow graph (Section 6.1). They store information about the flow of control in the function.
3. Select instructions that discard results of inactive instances are introduced where necessary (Section 6.2). These operations blend together values from disjoint paths that were executed by different instances.
4. Those parts of the control flow graph where the grouped instances may take different paths are linearized (Section 6.3). This means that, in such regions, all branches except for loop back edges are removed and code of originally disjoint paths is merged into one path.
5. Now that control flow has been transformed to data flow, every instruction is replaced by its vector counterpart or is split into W sequential operations (Section 6.4).

3.2 Algorithmic Challenges

The WFV algorithm is designed to vectorize code for a large class of language features. This includes arbitrary control flow, nested data structures, operations with side effects, and operations without vector equivalent. In general, the algorithm is able to transform any function, but possibly with significant overhead.

Arbitrary Control Flow. The control-flow to data-flow conversion that will be presented in the following sections can convert any kind of control flow, including loops with multiple exits, nested loops, and exits that leave multiple nesting levels. Section 6.5 also presents an extension for irreducible control flow. Vectorization of arbitrary control flow requires to generate mask code that tracks which instances are active at any point of the function. Furthermore, we do not want to discard results of inactive instances after each operation, since that would introduce far too much overhead. Placing only as many `blend` operations as required, however, is a non-trivial task in the presence of complex loop structures. To our knowledge, WFV is the only vectorization approach that can vectorize arbitrary control flow.

Partial Linearization of Control Flow. The Partial CFG Linearization phase is able to retain some of the original structure of the CFG using static information. This includes the preservation of arbitrarily shaped, nested regions for which the instances that are executed in SIMD fashion do not diverge into different paths of the CFG. Also, some loop exits can be retained even if instances may diverge inside the loop and leave it over different exit edges and in different iterations. To our knowledge, WFV is the only vectorization approach that can do partial CFG linearization without hard-coded pattern matching.

Operations with Side Effects. The WFV algorithm handles any kind of side effect by duplicating the scalar operation W times and guarding each operation by conditional execution (see Section 6.4.4).[1] This guard tests the mask value of the corresponding instance and only allows execution of the operation if the instance is active. This way, for a given operation, the vectorized function also preserves the order of occuring side effects as

[1]By *duplication* we do not refer to creating W exact duplicates but to creating W scalar instructions that operate on those elements of their input values that correspond to their instance. We also call this *splitting*.

compared to sequential execution. However, note that this does not hold for multiple operations with side effects within a function: Because execution happens in lock step, the vectorized function executes the first operation with all active instances before executing the second operation with all active instances. During sequential execution of the instances, on the other hand, each instance would execute both operations before the next instance would execute both. This exemplifies how WFV relies on data-parallel language semantics that specify that instances are considered independent and thus allow such a change of observable behavior.

Operations without Vector Equivalent. The algorithm can also handle operations without vector equivalent. For example, an operation may return a value whose return type is not vectorizable, or the operation itself does not have a vector equivalent. Similar to operations with side effects, the vectorized function uses W duplicates of the scalar operation. In general, however, no guards have to be introduced for these operations, since there is no effect observable from the outside. Thus, values of inactive instances will be discarded by later select operations just like vector elements.

3.3 Performance Issues of Vectorization

There are a number of issues that can potentially reduce the performance of the vector code.

Overhead of Mask and Select Operations. If the target architecture does not support predicated execution (such as most of today's SIMD instruction sets including SSE, AVX, and NEON), overhead has to be introduced in the form of instructions that compute and update masks and blend vectors together. The mask and select generation phases of WFV are designed to minimize the number of mask and blend operations that are introduced (see Sections 6.1 and 6.2).

Critical Path Extension. Full linearization of control flow by definition results in longer critical paths: Instead of taking either the `then` or the `else` branch of an `if` statement, both paths are executed, thus increasing the amount of executed code. As described in the introduction, this results in inactive instances that do not contribute to the final result. In the worst case, some code may be executed without *any* active instance, i.e., all its results will be thrown away. As Figure 3.1 shows, this inefficiency becomes

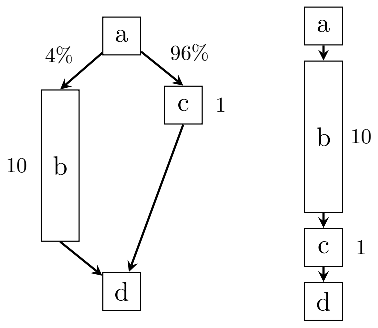


Figure 3.1: Pathological example for how linearization increases the amount of executed code. Percentages on edges represent execution frequencies, numbers next to blocks represent the cost of the block. Assuming 100 executions, the vectorized CFG on the right at $W = 4$ would yield a slowdown of $2\times$ compared to a scalar execution of the CFG on the left ($(100/4)*11 = 275$ vs. $96*1+4*10 = 136$). If the branch can be retained, the vector variant is improved by a factor of 8.1 ($(96/4)*1 + (4/4)*10 = 34$, $4\times$ faster than scalar).

even more problematic in cases where one of the two paths is much more expensive than the other but only rarely executed. If the linearized paths are in a loop the negative impact on performance is amplified even more. The Partial CFG Linearization phase of WFV reduces this problem by statically retaining parts of the control flow graph (see Section 6.3).

Register Pressure Increase. Linearization of control flow may also increase *register pressure*. Consider the example CFGs in Figure 3.2, where the variables that are live from block a to b are disjoint from those live from a to c. The vectorized code on the right has a higher register pressure than the sequential variant on the left. This is because all values that are live on edge $a \rightarrow c$ are now live throughout b, increasing its register pressure. Likewise, all values that are live on edge $b \rightarrow d$ are now live throughout c, increasing the register pressure in this block as well. The increased register pressure in turn can lead to additional spilling, which can severely impact performance. By preventing linearization of parts of the control flow graph, WFV does not increase register pressure to the same extent in the first place.

Uniform Operations. Often, instructions will use values that are the same for different instances. For example, a constant function argument is always

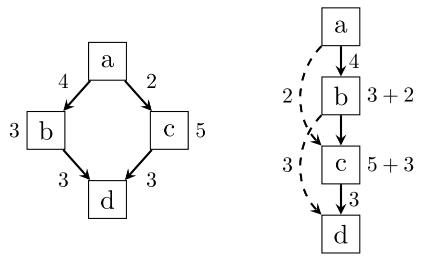


Figure 3.2: Linearization can increase register pressure. Numbers on edges represent live values on that edge, numbers next to blocks represent the maximum number of values live at any point in that block. Live value sets of $a \to b$ and $a \to c$ are disjoint. The register pressure of block b increases from 3 to 5, the pressure of c increases from 5 to 8.

the same for all instances. Such a *uniform* value has to be *broadcast* into vector form as soon as it is used as the input of a vector operation. This means that a vector is created that has a copy of the value at every position. It is generally desirable to do this as late as possible: Listing 3.1 shows a case where broadcasting constants eagerly results in slower code due to more `broadcast` and `extractelement` operations. Note that the eager broadcast code still exploits the fact that the call to `purefn` is *uniform* and does not produce side effects: only the first element of the vector is extracted, and only one call is issued. A more conservative approach would have to create W calls and merge the results.

In general, it can be beneficial to have more operations use the processor's scalar unit which otherwise is largely inactive due to most code using the vector unit. Thus, an intermixing of scalar and vector operations can result in better performance even if there would be no additional instructions introduced by early broadcasting. The Vectorization Analysis (Section 5.6) determines which values can be kept scalar. This implicitly defines the program points where `broadcast` operations have to be placed.

It should be noted that if a single value is reused often, broadcasting values eagerly requires less operations. Listing 3.2 shows an example for this. However, the code that results from late broadcasting does not introduce significant overhead in comparison. Also, the code is transparent to later optimization phases, which may decide to broadcast earlier and use vector instructions. This is not possible when broadcasting eagerly, since that has more influence on the rest of the code (see Listing 3.1).

Listing 3.1 Late broadcasting may require less operations. The function `purefn` is assumed to not produce any side effects, otherwise it would be split in both cases.

```
; scalar code. a0/a1/a2 are uniform, v is varying
%x = fadd float %a0, %a1
%y = call float @purefn(float %x, float %a2)
%z = fsub float %y, %v

; vector code (eager broadcast)
%v0 = broadcast float %a0 to <4 x float>
%v1 = broadcast float %a1 to <4 x float>
%v2 = broadcast float %a2 to <4 x float>
%vx = fadd <4 x float> %v0, %v1
%vx0 = extractelement <4 x float> %vx, i32 0
%vy0 = call float @purefn(float %vx0, %a2)
%vy = broadcast float %vy0 to <4 x float>
%vz = fsub <4 x float> %vy, %v

; vector code (late broadcast)
%x = fadd float %a0, %a1
%y = call float @purefn(float %x, float %a2)
%vy = broadcast float %y to <4 x float>
%z = fsub <4 x float> %vy, %v
```

Non-Divergent Control Flow. If a *uniform* value is used as the condition of a control-flow statement, all instances will always go into the same direction—control flow does not diverge. In such cases, the increased amount of executed code and increased register pressure as described above can be prevented. In Sections 5.9 and 6.3, we describe in detail how to exploit this using static techniques. In addition, Section 7.6 shows dynamic optimizations applicable to code that is expected to have uniform branching behavior.

Random Memory Access Operations. Vector `load` and `store` instructions that may operate on a vector of non-consecutive addresses have to be executed as scalar, sequential operations, unless the instruction set supports so-called *gather* and *scatter* operations. Most of today's instruction sets such as SSE, AVX, or NEON only allow vector loads from and vector stores to consecutive addresses. The Vectorization Analysis (Section 5) is often able to statically prove that a memory operation always accesses consecutive locations. This allows later phases to emit more efficient vector memory operations. Unfortunately, there are memory address calculations that

Listing 3.2 Late broadcasting may require more operations.

```
; scalar code. a is uniform, v is varying
%x = fadd float %a, %a
%y = fmul float %a, %a
%s = fsub float %x, %v
%t = fsub float %y, %v

; vector code (eager broadcast)
%va = broadcast float %a to <4 x float>
%vx = fadd <4 x float> %va, %va
%vy = fmul <4 x float> %va, %va
%vs = fsub <4 x float> %vx, %v
%vt = fsub <4 x float> %vy, %v

; vector code (late broadcast)
%x = fadd float %a, %a
%y = fmul float %a, %a
%vx = broadcast float %x to <4 x float>
%vy = broadcast float %y to <4 x float>
%s = fsub <4 x float> %vx, %v
%t = fsub <4 x float> %vy, %v
```

cannot be analyzed statically, e.g. if they involve dynamic input values. Sections 5.8 and 7.2 describe techniques that can improve the situation even in such cases.

Operations with Side Effects & Nested Data Structures. Operations with side effects introduce additional overhead due to being executed as guarded, sequential, scalar operations. To prevent execution of the operation for inactive instances, a *guard* is required for every instance: a test of the corresponding mask element, followed by a conditional branch that jumps to the operation or skips it.

The following problem occurs if such an operation has an operand that is a nested data structure: If this data structure is not uniform, we have to generate code that extracts the sequential values from that data structure and creates values of the corresponding scalar data structure for each of the sequential operations. If the operation is allowed to modify the value, we have to introduce additional write-back operations afterwards. Listing 3.3 shows an example for this.

Listing 3.3 Structures of data passed to unknown functions yield significant overhead due to creation of temporary scalar structures. The code exemplifies what operations are necessary to produce valid code if such a call appears in the function to be vectorized.

```
; scalar source code
define void @foo({ i32 }* %scalarStrPtr) {
  call void @bar({ i32 }* %scalarStrPtr)
  ret void
}

; "vectorized" code
define void @foo_W({ <4 x i32> }* %vectorStrPtr) {
  ; allocate memory for scalar struct
  %scalarStrPtr0 = alloca { i32 }, align 4

  ; write content of lane 0 of vector struct to scalar struct
  %ex0 = load { <4 x i32> }* %vectorStrPtr, align 16
  %ex1 = extractvalue { <4 x i32> } %ex0, 0
  %ex2 = extractelement <4 x i32> %ex1, i32 0
  %scalarStr0 = insertvalue { i32 } undef, i32 %ex2, 0
  store { i32 } %scalarStr0, { i32 }* %scalarStrPtr0, align 4

  ; call scalar function with temporary struct
  call void @bar({ i32 }* %scalarStrPtr0)

  ; write back scalar struct to vector struct lane 0
  %25 = load { i32 }* %scalarStrPtr0, align 4
  %26 = load { <4 x i32> }* %vectorStrPtr, align 16
  %27 = extractvalue { i32 } %25, 0
  %28 = extractvalue { <4 x i32> } %26, 0
  %29 = insertelement <4 x i32> %28, i32 %27, i32 0
  %30 = insertvalue { <4 x i32> } %26, <4 x i32> %29, 0

  ; repeat W-1 times for lanes (1, ..., W-1)
  ret void
}
```

4 Related Work

In this chapter, we discuss other work that is related to this thesis. We give an overview of the different kinds of approaches for vectorization and an overview of languages that offer automatic vectorization. Furthermore, we summarize work on static and dynamic analyses for SIMD execution and work on dynamic code variants for data-parallel programs.

4.1 Classic Loop Vectorization

Generating code for parallel hardware architectures is being studied since the emergence of vector computers and array processors. Various research programs were aimed at parallelizing scientific (Fortran) programs. Especially the analysis and automatic transformation of loop nests has been studied thoroughly [Allen & Kennedy 1987, Darte et al. 2000]. Allen et al. [1983] pioneered control-flow to data-flow conversion to help the dependence analyses to cope with more complex control structures. In our setting, we do not have to perform dependence analysis or find any parallelism; it is implicit in the programming model we consider. We use control-flow to data-flow conversion as a technique to *implement* data-parallel programs on SIMD processors. Furthermore, Allen et al. perform their transformation on the abstract syntax tree (AST) level. In this thesis, however, we consider control-flow to data-flow conversion on arbitrary control flow graphs in SSA form. In addition, our approach allows to retain certain control flow structures such that not all code is always executed after conversion.

In general, the control-flow conversion of Allen et al. is very similar to our Mask Generation phase, but it only targets vector machines that support *predicated execution* [Park & Schlansker 1991]. Predicated execution is a hardware feature that performs implicit blending of results of operations. For machines without predication, we are the first to show how masking of arbitrary control flow can be implemented using blend operations.

Another strain of work bases on uncovering explicit *instruction-level parallelism (ILP)* for *automatic vectorization*: Inner loops are unrolled several times such that multiple instances of the same instruction are generated.

These can be combined to vector instructions (“unroll-and-jam”). While this has first been introduced for classic vector machines, several authors also discuss this approach for SIMD instruction sets [Cheong & Lam 1997, Krall & Lelait 2000, Nuzman & Henderson 2006, Sreraman & Govindarajan 2000]. Since those techniques only consider inner loops, they only vectorize acyclic code regions. Also, target loops are often restricted, e.g. to static iteration counts, specific data-dependency schemes, or straight-line code in the loop body.

4.2 Superword Level Parallelism (SLP)

Superword Level Parallelism (SLP) [Larsen & Amarasinghe 2000] describes the occurrence of independent isomorphic statements, i.e., statements performing the same operations in the same order, inside a basic block, independent of loops. Such statements can be combined to SIMD instructions similar to instructions unrolled inside loops. Shin et al. [2005] extended the approach to also work in the presence of control flow by using predicates. Barik et al. [2010] introduced a similar approach based on dynamic programming. They exploit shuffle and horizontal vector operations and algebraic reassociation to uncover more potential for vectorization. Unfortunately, this technique introduces overhead for the packing and unpacking of vectors that makes the approach unusable for smaller fractions of code. Also, it is restricted to code-regions without loops. *Subgraph Level Parallelism* [Park et al. 2012] is another variant of SLP that minimizes packing and unpacking overhead.

4.3 Outer Loop Vectorization (OLV)

Our approach can be seen as a generalization of *outer loop vectorization* (OLV) [Ngo 1994, Nuzman & Zaks 2008, Scarborough & Kolsky 1986, Wolfe 1995]. In OLV, outer loops are unrolled to improve vectorization, e.g. due to longer trip counts of outer loops or better memory access schemes. However, OLV does not allow for divergent control flow inside the outer loop in contrast to our algorithm.

Thread merging [Yang et al. 2012] or *thread coarsening* [Magni et al. 2013], is a code transformation technique for data-parallel languages that is similar to WFV, but is aimed at GPUs. Although it does not perform any vectorization, thread merging is similar to OLV: It describes a technique where all code of a kernel function that depends on the instance identifier is duplicated to do more work per GPU thread. This can be seen as unrolling the implicit loop in which the kernel is executed.

4.4 Auto-Vectorizing Languages

On the language side, there are many different *data-parallel languages* and language extensions that automatically compile to parallel and/or vector code. Examples include NESL [Blelloch et al. 1993], CGiS [Fritz et al. 2004], MacroSS [Hormati et al. 2010], ArBB [Newburn et al. 2011], Cilk+[1], or Vc [Kretz & Lindenstruth 2012]. Modern, GPGPU-oriented languages like CUDA[2] or OpenCL[3] execute code in SIMT (Single Instruction, Multiple Threads) fashion. On a GPU, a thread roughly corresponds to an element in a vector register. As detailed below, some CPU implementations of CUDA and OpenCL employ techniques similar to WFV. VecImp [Leißa et al. 2012], Sierra [Leißa et al. 2014] and ispc [Pharr & Mark 2012] are languages that employ automatic SIMD vectorization specifically targeted at CPUs with SIMD instruction sets. Lately, the `#pragma simd` extension [Tian et al. 2012] is becoming more widespread in CPU compilers: it forces loop vectorization and can also vectorize functions. These vectorization approaches generally work in a similar fashion as WFV: they produce SIMD code from scalar source code in a data-parallel way.

For simple cases, all these languages will produce code similar to Whole-Function Vectorization. However, there are distinct differences in our approach: In contrast to WFV, no approach can handle arbitrarily diverging control flow, and no approach employs partial CFG linearization except in trivial cases. They also do not include analyses to determine static properties of the code that go as far as the analyses presented here. Sierra and ispc take a conceptually different approach here: they rely on the programmer to write the desired code, e.g. by offering a “non-divergent” `if` statement and `uniform` and `varying` keywords. Sierra, in addition, allows to break out of a vector context if desired, which is not possible in OpenCL or CUDA. PTX, the low-level instruction set architecture of Nvidia GPUs, includes a special, `uniform` branch instruction to allow a programmer or compiler to optimize control flow behavior. The `#pragma simd` extension also supports a modifier called `linear`, which is similar to our `consecutive` mark (Chapter 5). It

[1] `cilk.com`
[2] `developer.nvidia.com/cuda`
[3] `khronos.org/opencl`

helps the compiler to identify and optimize memory operations that access consecutive memory locations. However, it has to be used manually by the programmer.

WFV is entirely based on control flow graphs in SSA form, whereas to our knowledge all other approaches operate on the level of the AST. The reason for this is that performing this kind of vectorization on an abstract syntax tree in the front end is less complex than on arbitrary control flow of low-level code such as LLVM bitcode. However, this approach loses optimization potential: classic compiler optimizations do not work well on vectorized code. The fact that control flow is no longer explicit disturbs many compiler optimizations such as common-subexpression elimination. Hence, it is better to first apply optimizations on the scalar code and then perform vectorization.

RTSL [Parker et al. 2007] is a domain-specific language (DSL) in computer graphics. It allows the user to write *shaders*, i.e., functions that describe visual properties of materials, in scalar code that is automatically transformed to vector code usable by packet ray tracing systems. RTSL vectorizes code on the syntax level and generates C code that is forwarded to a standard compiler.

Our model of computation is inspired by the GPGPU-style languages. However, our pass comes so late in the compilation phase (immediately before machine-dependent code generation tasks) that the source language's influence is negligible. We argue that all languages mentioned here can be mapped to the language core presented in Section 5.1 and thus can profit from our algorithm. In the evaluation in Chapter 8, we demonstrate this for three languages: RenderMan, OpenCL, and a subset of C/C++ that is suitable for loop vectorization.

4.4.1 OpenCL and CUDA

An increasing number of OpenCL drivers is being developed by different software vendors for all kinds of platforms from GPUs to mobile devices. For comparison purposes, the x86 CPU drivers by Intel[4] and AMD[5] are most interesting. However, most details about the underlying implementation are not disclosed. Both drivers have in common that they build on LLVM and exploit all available cores with some multi-threading scheme. The Intel

[4]`software.intel.com/en-us/vcsource/tools/opencl`, OpenCL SDK XE 2013 R2
[5]`developer.amd.com/sdks/AMDAPPSDK`, version 2.8.1

driver also performs SIMD vectorization similar to our implementation[6]. However, to our knowledge, it lacks analyses to retain uniform computations and control flow, an important source of performance (see Chapter 8).

The Portland Group implemented an x86 CPU driver for CUDA which also makes use of both multi-threading and SIMD vectorization.[7] Again, no details are publicly available. MCUDA [Stratton et al. 2008] and Ocelot [Diamos et al. 2010] are other x86 CPU implementations of CUDA that do not use WFV. MCUDA first introduced a “thread loop”-based synchronization scheme that is similar to the barrier synchronization described in Section 8.2. This approach uses loop fission of the thread loop to remove barriers, which results in similar code as our approach if no barriers are inside loops. In that scenario however, MCUDA generates additional, artificial synchronization points at the loop header and before the back branch. This can impose significant overhead due to additional loading and storing of live variables. Jääskeläinen et al. [2010] implemented *POCL*, a standalone OpenCL driver that generates customized code for FPGAs and also uses this synchronization scheme. In contrast to our driver, they rely on the FPGA design to use the instruction-level parallelism exposed by duplicating the kernel code W times instead of performing explicit SIMD vectorization. We compare the performance of POCL to our driver in Chapter 8. Additional OpenCL implementations that do not make use of SIMD vectorization are TwinPeaks [Gummaraju et al. 2010] and Clover[8] as part of GalliumCompute[9].

4.5 SIMD Property Analyses

Implementations of data-parallel languages often include an analysis that marks program points as `uniform` or `varying`, similar to a subset of our Vectorization Analysis (Section 5.6). Examples include the *invariant analysis* of CGiS [Fritz et al. 2004] and the *variance analysis* by Stratton et al. [2010]. The *divergence analysis* presented by Coutinho et al. [2011] and Sampaio et al. [2013] also classifies values as `uniform` or `varying`. In addition to our approach, they can also analyze affine constraints, yielding more precise results in some situations. Since the divergence analysis also marks branches

[6]We have no information about the AMD driver but suspect that no Whole-Function Vectorization is used due to the inferior performance.
[7]`pgroup.com/resources/cuda-x86.htm`, PGI CUDA-x86
[8]`people.freedesktop.org/~{}steckdenis/clover`
[9]`http://dri.freedesktop.org/wiki/GalliumCompute/`

as `uniform` or `varying`, Coutinho et al. employ an optimization that replaces PTX `branch` instructions by the `uniform` equivalent `bra.uni`. This covers a trivial subset of what our Rewire Target Analysis (Section 5.9) and Partial CFG Linearization (Section 6.3) do: As we show in Section 6.3, the approach does not produce correct results for unstructured control flow. Lee et al. [2013] introduced a scalarizing compiler for GPU kernels that identifies code regions where no threads are inactive. Their *convergence analysis* employs Stratton's variance analysis and yields similar results to the part of our analysis that marks program points that are always executed by all instances (Section 5.6.11). This is used to modify the program to use only one instruction or register per warp instead of per thread, similar to what the scalarization phase of Fritz et al. [2004] does in CGiS.

To our knowledge, our work is the first to classify instructions that have to be executed sequentially or guarded, the first to determine consecutive memory access operations, and the first to determine necessary blocks for partial CFG linearization.

The SMT-based improvement for our analysis shares some similarities with techniques developed for verification and performance analysis of GPU code. Various approaches exist that analyze memory access patterns for GPUs. However, none of the static approaches can handle integer division, modulo by constants, or non-constant inputs. CuMAPz [Kim & Shrivastava 2011] and the official CUDA Visual Profiler perform dynamic analyses of memory access patterns and report non-coalesced operations to the programmer. Yang et al. [2010] implemented a static compiler optimization to improve non-coalesced accesses using shared memory. Li & Gopalakrishnan [2010] proposed an SMT-based approach for verification of GPU kernels. This was extended by Lv et al. [2011] to also profile coalescing. Tripakis et al. [2010] use an SMT solver to prove non-interference of SPMD programs. GPUVerify [Betts et al. 2012] is a tool that uses Z3 to prove OpenCL and CUDA kernels to be free from race-conditions and barrier divergence. None of these SMT-based techniques is concerned with automatic code optimization but only with verification.

4.6 Dynamic Variants

The dynamic approach of Shin [2007] employs "branches-on-superword-condition-code" (BOSCC) to exploit non-diverging control flow. This technique reintroduces control flow into vectorized code to exploit situations where the predicates (masks) for certain parts of the code are entirely `true` or `false` at runtime. While this prevents unnecessary execution of code, it suffers from some overhead for the dynamic test and does not solve the problem of increased register pressure as described in Section 3.3 without code duplication: the code still has to account for situations where the predicate is not entirely `true` or `false`, unless a completely disjoint path is introduced where all code is duplicated. Our analysis can do better by providing the necessary information statically, which allows to retain the original control flow. However, it is possible to benefit from both our optimizations *and* BOSCCs.

The compiler of RTSL [Parker et al. 2007] includes a similar technique that automatically introduces multiple code paths for a conditional branch where instances may diverge: one for the general case of a mixed mask, one for the "all-`false`" case, one for the "all-`true`" case. Timnat et al. [2014] describe a more generic variant of the same technique. In addition to the separate paths for entirely `true` or `false` masks they can also switch between the paths. For example, when a loop starts iterating with all instances active, it executes a code path that does not blend values or track masks. When the first divergence occurs, i.e., some but not all instances leave the loop, execution continues on a different code path that does include blend operations. These techniques are summarized in Section 7.6.

Manual implementations of dynamic variants for specific applications exist, e.g. for a ray tracer for the Intel MIC architecture by Benthin et al. [2012]. They trace packets of rays in a SIMD fashion, but switch to scalar execution once the rays diverge. With an appropriate heuristic, this could be achieved automatically with a dynamic variant as described in Section 7.3.

The *Instance Reorganization* variant described in Section 7.5 shares some similarities with approaches like *dynamic warp formation* [Fung et al. 2007], *branch and data herding* [Sartori & Kumar 2012], and *SIMD lane permutation* [Rhu & Erez 2013]. A variety of manual implementations of similar schemes exist in ray tracing, where rays are reordered for packet tracing [Benthin et al. 2012, Boulos et al. 2008, Moon et al. 2010, Wald 2011]. However, all of these approaches are concerned about *upfront* reordering of instances that execute together in a SIMD group. This way, branching and data access behavior can be improved without modifying the code. In contrast, Instance Reorganization is performed *within* a fixed group of instances and relies on code transformation.

4.7 Summary

Whole-Function Vectorization is a generic vectorization approach that focuses on control-flow to data-flow conversion and instruction vectorization. It can be used in a variety of scenarios such as loop vectorization, outer loop vectorization, and in the back ends of data-parallel languages. In contrast to most other approaches, WFV works on the control flow graph rather than on source or syntax tree level. This allows to aggressively optimize the code before vectorization. The state-of-the art in vectorization is usually limited to straight-line code or structured control flow, and complete linearization of the control flow structure is performed except in trivial cases. WFV on the other hand can handle arbitrary control flow, and can retain some structure of the CFG. The analyses introduced in this thesis describe a more complete picture of the behavior of SIMD execution than previous divergence or variance analyses. The WFV-based OpenCL driver employs some of the most advanced code generation and vectorization techniques that have been developed. The prototype offers, together with the Intel driver, the currently best performance of any commonly used OpenCL CPU driver.

5 SIMD Property Analyses

In this chapter, we describe the heart of the WFV algorithm: a set of analyses that determine properties of a function for a SIMD execution model.

The analyses presented in this chapter determine a variety of properties for the instructions, basic blocks, and loops of a scalar source function. The properties related to values are listed in Table 5.1, those related to blocks and loops in Table 5.2. They describe the behavior of the function in data-parallel execution.

In general, the SIMD properties are dynamic because they describe values that may depend on input arguments. The Vectorization Analysis (Section 5.6) describes a safe, static approximation of them.

Many of the analysis results are interdependent. For example, values may be non-`uniform` because of divergent control flow, and control flow may diverge because of non-`uniform` values (see Section 5.2). Thus, the Vectorization Analysis consists of several parts that interact.

The analysis results are used throughout all phases of vectorization and in many cases allow to generate more efficient code.

Table 5.1 Instruction properties derived by our analyses. Note that `varying` is defined for presentation purposes only: it subsumes `consecutive` and `unknown`.

Property	Symbol	Description
`uniform`	`u`	result is the same for all instances
`varying`	`v`	result is not provably `uniform`
`consecutive`	`c`	result consists of consecutive values
`unknown`	`k`	result values follow no usable pattern
`aligned`	`a`	result value is aligned to multiple of S
`nonvectorizable`	`n`	result type has no vector counterpart
`sequential`	`s`	execute W times sequentially
`guarded`	`g`	execute sequentially for active instances only

Table 5.2 List of block and loop properties derived by our analyses.

Property	Description
`by_all`	block is always executed by all instances
`div_causing`	block is a divergence-causing block
`blend`$_v$	block is join point of instances that diverged at block v
`rewire`$_v$	block is a rewire target of a `div_causing` block v
`divergent`	loop that instances may leave at different points (time & space)

5.1 Program Representation

We consider the scalar function f to be given in a typed, low-level representation. A function is represented as a control flow graph of instructions. Furthermore, we require that f is in SSA form, i.e., every variable has a single static assignment and every use of a variable is dominated by its definition. A prominent example of such a program representation is LLVM bitcode [Lattner & Adve 2004] which we also use in our evaluation (Chapter 8). We will restrict ourselves to a subset of a language that contains only the relevant elements for this thesis. Figure 5.1 shows its types and instructions. Other instructions, such as arithmetic and comparison operators are straightforward and omitted for the sake of brevity.

This program representation reflects today's consensus of instruction set architectures well. `alloca` allocates local memory and returns the corresponding address as a pointer of the requested type. The `gep` instruction ("get element pointer") performs address arithmetic. `load` (`store`) takes a base address and reads (writes) the elements of the vector consecutively from (to) this address. The `bool` type is special in that we do not allow creating pointers of it. This is because the purpose of boolean values is solely to encode control flow behavior.

The function `tid` returns the instance identifier of the running instance (see above). A `phi` represents the usual ϕ-function from SSA. An `lcssaphi` represents the ϕ-function with only one entry that is found in loop exit blocks of functions in LCSSA form. The operation `arg`(i) accesses the i-th argument to f. We assume that all pointer arguments to f are aligned to the SIMD register size.

Function calls are represented by `call` instructions that receive a function identifier and a set of parameters. Branches are represented by `branch` instructions which take a boolean condition and return one of their target program point identifiers.

Types			Instructions		
σ	=	`unit` $\vert\, \tau$	`val`	:	τ
τ	=	$\beta \,\vert\, \pi$	`tid`	:	`unit` $\rightarrow$ `int`
β	=	`bool`	`arg`	:	`int` $\rightarrow \tau$
π	=	$\nu \,\vert\, \pi*$	`phi`	:	$(\tau, \gamma) \times (\tau, \gamma) \rightarrow \tau$
ν	=	`int` $\vert$ `float`	`lcssaphi`	:	$(\tau, \gamma) \rightarrow \tau$
γ	=	`program point id`	`alloca`	:	`int` $\rightarrow \pi*$
δ	=	`function id`	`gep`	:	$\pi* \times$ `int` $\rightarrow \pi*$
			`load`	:	$\pi* \rightarrow \pi$
			`store`	:	$\pi* \times \pi \rightarrow$ `unit`
			`call`	:	$\delta \times (\tau \times \cdots \times \tau) \rightarrow \sigma$
			`branch`	:	$\beta \times \gamma \times \gamma \rightarrow \gamma$

Figure 5.1: Program representation: types and instructions.

5.2 SIMD Properties

We now define the SIMD properties relevant for our analyses and discuss their meaning. The following definitions describe *dynamic* properties, i.e., they may depend on values that are not known statically. Our analyses derive conservative approximations of them.

5.2.1 Uniform & Varying Values

Definition 1 (Uniformity) *An operation is* `uniform` *iff it does not produce any side effects and a single, scalar operation is sufficient to produce all values required for all instances of the executing SIMD group. Otherwise, it is* `varying`*.*

In general, this is a dynamic property that may depend on input values. The Vectorization Analysis (Section 5.6) computes a safe, static underapproximation of *uniformity.*

The `uniform` property also depends on control flow: If the condition that controls a conditional branch is `varying`, control flow may diverge, and the program points where control flow joins again are join points (`blend`). ϕ-functions at such join points and LCSSA ϕ-functions at exit points of `divergent` loops become `varying`: Consider a ϕ-function with constants as incoming values on all edges. If that ϕ-function resides at a join point, i.e., it is `blend`, the ϕ is *not* `uniform` but `varying`. This is because not all instances enter the ϕ's program point from the same predecessor in every execution, so the result has to be a vector of possibly different constants for each instance.

However, if we can prove that the ϕ is not `blend`, then the ϕ is `uniform`, too. Similar behavior can be observed for LCSSA ϕ-functions at exit points of `divergent` loops and for operations with side effects at points that are not provably always executed by all instances (`by_all`).

The `varying` property is a general property that states that a program point is not `uniform`, i.e., it holds different values for different instances. Properties like `consecutive`, `unknown`, or `guarded` describe a `varying` program point in more detail.

If the Rewire Target Analysis (Section 5.9) is disabled, all conditional branches and non-void return statements are conservatively marked `varying`. This effectively means that all values that depend on any control flow will be considered `varying` as well.

5.2.2 Consecutivity & Alignment

Definition 2 (Consecutivity) *An operation is* `consecutive` *iff its results for a SIMD group of instances are natural numbers where each number is by one larger than the number of its predecessor instance, except for the first instance of the group.*

It is common for vector elements to hold *consecutive* values. Assume we have a base address and know that the values of the offsets are consecutive for consecutive instance identifiers. Putting the offsets in a vector yields $\langle\ n, n+1, \ldots, n+W-1\ \rangle$. Using such an offset vector in a `gep` gives a vector of consecutive addresses. Thus, optimized vector `load` and `store` instructions that only operate on consecutive addresses can be used.

Definition 3 (Aligned) *An operation is* `aligned` *iff its results for a SIMD group of instances are natural numbers and the result of the first instance is a multiple of the SIMD width S.*

Current SIMD hardware usually provides more efficient vector memory operations to access memory locations that are *aligned.* Thus, it is also important to know whether an offset vector starts exactly at the SIMD alignment boundary. If, for example, a `load` accesses the array element next to the instance identifier, the vectorized variant would access $\langle\ id+1, id+2, \ldots, id+W$. Without further optimizations, this non-`aligned` operation would require two vector `loads` and a `shuffle` operation. In the context of classic loop vectorization, options how to best compute unaligned memory operations have been studied extensively [Eichenberger et al. 2004, Fireman et al. 2007, Shahbahrami et al. 2006].

5.2.3 Sequential & Non-Vectorizable Operations

There are values and operations that must not or cannot be vectorized. These are executed as W sequential, scalar operations, potentially guarded by conditionals.

Definition 4 (Non-Vectorizable Operation) *An operation is* `non-vectorizable` *iff its return type or the type of at least one of its operands has no vector equivalent.*

For example, a call that returns a `void` pointer is `nonvectorizable` since there is no vector of pointers in our program representation. More importantly, however, a `nonvectorizable` value forces operations that use it to be executed sequentially:

Definition 5 (Sequential Operation) *An operation is* `sequential` *iff it has no vector equivalent or if it is* `nonvectorizable` *and not* `uniform`*.*

This is mostly relevant for operations with `nonvectorizable` return type or at least one `nonvectorizable` operand. It also applies to all those operations that do not have a vector counterpart, for example some SIMD instructions sets do not support a `load` from non-consecutive memory locations.

Definition 6 (Guarded Operation) *An operation is* `guarded` *iff it may have side effects and is not executed by all instances.*

Unknown calls have to be expected to produce side effects and thus require guarded execution. Similarly, `store` operations must never be executed for inactive instances.

5.2.4 All-Instances-Active Operations

Many sources of overhead in vectorized programs are caused by conservative code generation because instances may diverge. If it can be proven that *all* instances are active whenever an operation is executed, more efficient code can be generated.

Definition 7 (All-Active Program Point) *A program point is* `by_all` *iff it is not executed by a SIMD group which has an inactive instance.*

For example, operations that may have side effects have to be executed sequentially and guarded if some instances may be inactive. This is because the side effect must not occur for inactive instances. If the operation is `by_all`, it will always be executed by all instances, and no guards are required.

5.2.5 Divergent Loops

Definition 8 (Loop Divergence) *A loop is* `divergent` *iff any two instances of a SIMD group leave the loop over different exit edges and/or in different iterations.*

It is important to note that this definition of the term "divergence" differs from the usual one that describes a loop that never terminates. A divergent loop in the SIMD context can be left at different points in time (iterations) or space (exit blocks) by different instances. Because of this, the loop has to keep track of which instances are active, which left at which exit, and which values they produced. This involves overhead for the required mask and `blend` operations. However, if the loop is *not* `divergent`, i.e., it is always left by all instances that entered it at once and at the same exit, then no additional tracking of the instances is required.

5.2.6 Divergence-Causing Blocks & Rewire Targets

The `div_causing` and `rewire` properties are used to describe the divergence behavior of control flow. Program points marked `div_causing` or `rewire` correspond to exits and entries of basic blocks, so we simply refer to the properties as *block properties*:

Definition 9 (Divergence-Causing Block) *A block b is* `div_causing` *iff not all instances that entered b execute the same successor block.*

A block that ends with a `varying` branch may cause control flow of different instances to diverge, i.e., to execute different program points after the branch. Paths that start at such a block are subject to control-flow to data-flow conversion. The blocks that define how the CFG has to be transformed for this conversion are defined as follows:

Definition 10 (Rewire Target) *A block b is a* `rewire` target *of a* `div_causing` *block d iff it is entered by at least one instance that executed d and either*

1. *it is a successor block of d, or*
2. *it is an end block of a disjoint path due to d.*

End blocks of disjoint paths may be either join points, loop exits, or loop latches. Consider an edge $x \to b$, where b is a `rewire` target of block d, and $x \neq d$. This edge marks the end of a disjoint path p_1 that started at a successor of d. If this edge is reached, this indicates that d was executed earlier and that some instances went into another disjoint path p_2. So, before b is entered, execution continues with p_2. The start block of p_2 is another `rewire` target of d. Section 5.9 will discuss this in more detail and give explanatory examples.

5.3 Analysis Framework

In the following, we give a definition of a framework to describe and derive SIMD properties of a program. The framework is built on the ideas of *abstract interpretation* [Cousot & Cousot 1976; 1977; 1979] and closely follows the notation and definitions of Grund [2012]:

First, an *Operational Semantics (OS)* (Section 5.4) is defined for the program representation introduced in Section 5.1. It describes the exact behavior of a program when being executed as transformations between program states. The effects of each operation of the program are described by a transformation rule that modifies the input state.

Second, OS implicitly defines a *Collecting Semantics (CS)*. It abstracts from the OS by combining states to a set of states, which allows to reason about the effects of an operation for all possible input states, e.g. all possible input values of a function.

Third, an *Abstract Semantics (AS)* (Section 5.6) is defined. It abstracts from the CS by reasoning about abstract properties rather than sets of concrete values. This allows its transformation rules to be implemented as a static program analysis that can be executed at compile time.

To ensure soundness of the abstraction, we sketch a proof for the *local consistency* of CS and AS (Section 5.7). This is done by defining a *concretization function* γ that maps from AS to CS.

5.4 Operational Semantics

We use the notation of Seidl et al. [2010] to define an Operational Semantics (OS) for the program representation from Section 5.1. Let $d = (\rho, \mu, @, \#)$ be a *state* of the execution at a given node, where $\rho : \mathit{Vars} \to \tau$ is a mapping of variables to values, $\mu : \pi* \to \pi$ is a mapping of memory locations to values, $@ : \gamma \to \mathit{bool}$ stores for each program point whether it is *active* or not, and $\# : \texttt{int}$ is an instance identifier. The notation $\rho \oplus \{x \mapsto y\}$ stands for

$$\lambda v. \begin{cases} y & \text{if } x = v \\ \rho(v) & \text{otherwise,} \end{cases}$$

i.e., the value of x in ρ is updated to y.

The evaluation functions (also called the *transformer* of OS) are defined in Figure 5.2. For the sake of brevity, we only show the effects on the updated elements of state d per rule.

We assume a non-standard execution model: *predicated execution*. Regardless of the flow of control, every operation in the program is executed in topological order. However, the state is only updated if the value $@(x)$ of the program point x is `true`, otherwise the operation has no effect: $@(x)$ is the *predicate* of x. The value of @ is `true` for the entry program point, and initially `false` for all other program points. A `branch` at program point x updates the mapping of its successor program points: The value $@(s_1)$ for the `true` successor s_1 is set to the result of a disjunction of the current value $@(s_1)$ and a conjunction of $@(x)$ and the condition. The value $@(s_2)$ for the `false` successor s_2 of a conditional `branch` is set to the result of a disjunction of the current value $@(s_2)$ and a conjunction of $@(x)$ and the negated condition. The disjunctions merge the predicates of all incoming edges at control flow join points. The only exception to this are loop headers, for which @ must not be updated with a disjunction since that would reactivate instances that left the loop already. Thus, @ is updated by selecting the value from the preheader in the first iteration and the value from the latch in all subsequent iterations, similar to a ϕ-function. This scheme ensures that all the control flow is encoded in @.

Note that # is never written, which reflects the fact that the instance identifier is an implicit value of every instance that can only be queried, not changed.

$$
\begin{array}{rcl}
[\![x \leftarrow c]\!](d) & = & \rho \oplus \{x \mapsto c\} \\
[\![x \leftarrow v]\!](d) & = & \rho \oplus \{x \mapsto \rho(v)\} \\
[\![x \leftarrow \texttt{tid}]\!](d) & = & \rho \oplus \{x \mapsto \#\} \\
[\![x \leftarrow \texttt{arg}(i)]\!](d) & = & \rho \oplus \{x \mapsto \rho(\texttt{arg}(i))\} \\
[\![x \leftarrow \texttt{phi}((v_1, b_1), (v_2, b_2))]\!](d) & = & \rho \oplus \left\{ x \mapsto \begin{cases} \rho(v_1) & \text{if } @(b_1) = \texttt{true} \\ \rho(v_2) & \text{if } @(b_2) = \texttt{true} \end{cases} \right\} \\
[\![x \leftarrow \texttt{lcssaphi}(v, b)]\!](d) & = & \rho \oplus \{x \mapsto v\} \\
[\![x \leftarrow \texttt{alloca}(n)]\!](d) & = & \rho \oplus \{x \mapsto \texttt{newptr}(\mu, c)\} \\
[\![x \leftarrow \texttt{load}(a)]\!](d) & = & \rho \oplus \{x \mapsto \mu(\rho(a))\} \\
[\![x \leftarrow \texttt{store}(a, v)]\!](d) & = & \mu \oplus \{\rho(a) \mapsto \rho(v)\} \\
[\![x \leftarrow \texttt{call}(\texttt{g}, v_1, \ldots, v_n)]\!](d) & = & \rho \oplus \{x \mapsto \texttt{g}(\rho(v_1), \ldots, \rho(v_n))\}\ , \\
 & & \mu \oplus \{M_g(\mu, \rho(v_1), \ldots, \rho(v_n))\} \\
[\![x \leftarrow \omega(v_1, v_2)]\!](d) & = & \rho \oplus \{x \mapsto \omega(\rho(v_1), \rho(v_2))\} \\
[\![x \leftarrow \texttt{branch}(v, b_1, b_2)]\!](d) & = & @ \oplus \{b_1 \mapsto @(b_1) \vee @(x) \wedge v, b_2 \mapsto @(b_2) \vee @(x) \wedge \neg v\}
\end{array}
$$

Figure 5.2: Evaluation functions for the Operational Semantics. ω is an arithmetic or comparison operator, `newptr` returns a new memory location in μ, and M_g describes the memory effects of an execution of function g. A function has no effect if the program point's mapping in @ (its *predicate*) is `false`. An unconditional branch b implicitly updates the predicate of its successor s with its own predicate: $@(s) \vee @(b)$.

5.4.1 Lifting to Vector Semantics

Finally, in order to reason about SIMD programs, we lift OS to operate on vectors instead of scalar values. This is straightforward: # is a vector of instance identifiers now, and every function is evaluated separately for each of these SIMD instances. Instances that are inactive, i.e., their value in @ at the current program point is `false`, are not updated. Note that this lifting means that `phi` can *blend* two incoming vectors if some values in @ are `true` for either predecessor program point.

In order for this to work, loops iterate until the predicate that describes which instances stay in the loop is entirely `false`. This means that loops iterate as long as *any* of the instances needs to iterate. Execution then continues with the loop exit program points, again in topological order. Their predicates are correct since they have been continually updated while iterating the loop.

To ensure valid SIMD semantics, the predicated execution model employs *postdominator reconvergence*, also known as *stack-based reconvergence* [Fung & Aamodt 2011]. This scheme *aligns* all traces of states of the instances, i.e., the latest *reconvergence point* of all instances that diverged at a conditional branch is the postdominator of that branch. This will allow us to reason about universal properties of the states in the *Collecting Semantics* of OS.

5.5 Collecting Semantics

OS implicitly defines a *Collecting Semantics (CS)* which combines all possible states of a program point to a set of states. This means it operates on a set of states D which contains all states d of the program at a given program point. The sets ρ, μ, @, and # are lifted to sets of sets. This collection of sets of states allows us to reason about universal properties of the states. Most importantly, the alignment of traces that is ensured by postdominator reconvergence prevents cases where we would derive properties from values that belong to different loop iterations.

5.6 Vectorization Analysis

We now define an Abstract Semantics (AS) that abstracts from the Collection Semantics by reasoning over SIMD properties instead of concrete values. In the following, the transfer functions

$$[\![\cdot]\!]^\sharp : (\mathit{Vars} \to (\mathbb{D} \times \mathbb{B} \times \mathbb{A} \times \mathbb{L})) \to (\mathit{Vars} \to (\mathbb{D} \times \mathbb{B} \times \mathbb{A} \times \mathbb{L}))$$

of AS (the *abstract transformer*) are defined. They can be computed efficiently by a data-flow analysis we refer to as *Vectorization Analysis*. The functions D_{Abs}, B, A, and L contain the analysis information. They map variables to elements of the lattices $\mathbb{D}$, $\mathbb{B}$, $\mathbb{A}$, and $\mathbb{L}$ (see Figures 5.3 and 5.4).

Note that, since we consider SSA-form programs, the terms *variables* and *values* both refer to *program points*. A program point also has a concrete operation associated. The presented analyses use join (not meet) lattices and employ the common perspective that instructions reside on edges instead of nodes. Program points thus sit between the instructions (see Figure 5.5). This scheme has the advantage that the join and the update of the flow facts are cleanly separated.

As in the Operational Semantics, the notation $D_{Abs} \oplus \{x \mapsto y\}$ stands for

$$\lambda v. \begin{cases} y & \text{if } x = v \\ D_{Abs}(v) & \text{otherwise,} \end{cases}$$

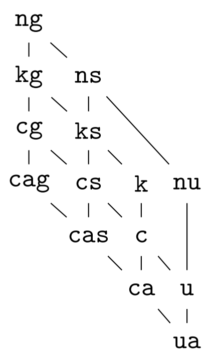
i.e., the value of x in D_{Abs} is updated to y.

The order of properties of D_{Abs} is visualized by the Hasse diagram in Figure 5.3. It describes the *precision relation* of the properties. An element that is lower in the diagram is *more precise* than an element further up. This relation is expressed by the operators $\sqsubseteq$ and $\sqsupseteq$: The notation $a \sqsubseteq b$ describes the fact that a is *at least as precise* as b, $a \sqsupseteq b$ means that a is *at most as precise* as b.

5.6.1 Tracked Information

Our analysis tracks the following information for each program point (see Tables 5.1 and 5.2 and the lattices in Figures 5.3 and 5.4): Is the value the same for all instances that are executed together in a SIMD group (`uniform`), and is it a multiple of the SIMD width (`uniform/aligned`) or does it have a type that is not vectorizable (`uniform/nonvectorizable`)? Such a `uniform` variable can be kept scalar and broadcast into a vector when needed, or used multiple times in `sequential` operations if it is `nonvectorizable`.

Otherwise, a value may hold different values for different instances (`varying`). If possible, these values are merged into a vector in the SIMD function. We track whether a `varying` value contains consecutive values for consecutive instances (`consecutive`) and whether it is aligned to the SIMD width (`consecutive/aligned`). The latter allows to use faster, aligned memory instructions. The former still avoids sequential, scalar execution but needs unaligned memory accesses. If nothing can be said about the shape of a `varying` value, its analysis value is set to `unknown` or, if it has a type that cannot be vectorized, `nonvectorizable`.



Legend

Acronym	Property
n	`nonvectorizable`
g	`guarded`
s	`sequential`
k	`unknown`
c	`consecutive`
a	`aligned`
u	`uniform`

Concrete Value Examples

Element	Shape of Vector	Example
`k`	$\langle\ n_0, n_1, \ldots, n_{W-1}\ \rangle$	$\langle\ 9, 2, 7, 1\ \rangle$
`c`	$\langle\ n, n+1, \ldots, n+W-1\ \rangle$	$\langle\ 3, 4, 5, 6\ \rangle$
`ca`	$\langle\ m, m+1, \ldots, m+W-1\ \rangle$	$\langle\ 0, 1, 2, 3\ \rangle$
`u`	$\langle\ m+c, m+c, \ldots, m+c\ \rangle$	$\langle\ 7, 7, 7, 7\ \rangle$
`ua`	$\langle\ m, m, \ldots, m\ \rangle$	$\langle\ 4, 4, 4, 4\ \rangle$
`nu`	type not vectorizable	`void*`,`void*`,`void*`,`void*`
		$n \in$ `int`, `float`, $m = n \cdot W$

Figure 5.3: Hasse diagram of the lattice $\mathbb{D}$, legend, and value examples. The lattice is lifted to a function space whose elements map variables to elements of $\mathbb{D}$. Note that our notation uses the join, not meet style, i.e., $\top$ (`ng`) is least informative. The properties map to those listed in Table 5.1.

$$\begin{tikzcd}[row sep=small]
\texttt{blend} \arrow[d, dash] & \top \arrow[d, dash] & \texttt{divergent} \arrow[d, dash] \\
\bot & \texttt{by\_all} & \bot
\end{tikzcd}$$

Figure 5.4: Hasse diagrams of the lattices $\mathbb{B}$, $\mathbb{A}$, and $\mathbb{L}$. The properties map to those listed in Table 5.2.

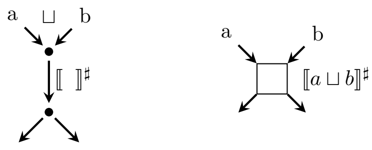


Figure 5.5: Left: Analysis setup with separated join and update of flow facts. Right: Classic setup with mixed join/update.

Furthermore, we track the information whether a program point is `sequential`, and whether it is `guarded`. A `sequential` operation cannot be executed as a vector operation, but has to be split into W sequential operations. If the program point is `guarded`, it also requires conditional execution. This means that each sequential operation is guarded by an `if` statement that evaluates to `true` only if the mask element of the corresponding instance is set.

Finally, control-flow related information is tracked: Is the program point always executed without inactive instances (`by_all`)? Is it a join point of diverged instances of some program point v ($\texttt{blend}_v$)? Does it have to be executed if some program point v was executed ($\texttt{rewire}_v$) or can it be skipped under certain circumstances (`optional`)? Finally, we track whether a program point is part of a loop that some instances may leave at different exits or at different points in time (`divergent`).

5.6.2 Initial State

The initial information passed to the analysis is twofold: the signatures of the scalar source function f and the vectorized target declaration of f_W on one hand, user-defined marks on the other. All other program points are initialized with the bottom element of the lattice, `ua` (`uniform`/`aligned`).

Function Arguments

$$
[\![x \leftarrow \texttt{arg}(i)]\!]^\sharp(D_{Abs}, B, A, L) =
$$

$$
\left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{ua} & \text{if } \texttt{type(}f\texttt{,i)} = \texttt{type(}f_W\texttt{,i)} \wedge \texttt{type(}f\texttt{,i)} \text{ is pointer} \\ \texttt{u} & \text{if } \texttt{type(}f\texttt{,i)} = \texttt{type(}f_W\texttt{,i)} \\ \texttt{nu} & \text{if } \texttt{type(}f\texttt{,i)} = \texttt{type(}f_W\texttt{,i)} \wedge \\ & \texttt{type(}f\texttt{,i)} \text{ not vectorizable} \\ \texttt{ca} & \text{if } \texttt{type(}f_W\texttt{,i)} = \texttt{vect}_{\texttt{W}}\texttt{(type(}f\texttt{,i))} \wedge \\ & \texttt{type(}f\texttt{,i)} \text{ is pointer} \\ \texttt{k} & \text{if } \texttt{type(}f_W\texttt{,i)} = \texttt{vect}_{\texttt{W}}\texttt{(type(}f\texttt{,i))} \end{cases} \right\}, \right.
$$

$$
\left. B, A, L \right)
$$

The argument and return types of the signatures are tested for equality.

Those arguments for which the types match are `uniform`, meaning they will have the same value for all instances that are executed together in the SIMD function. If the argument is a pointer, it is also `aligned`, since we assume all pointer arguments to be aligned to the SIMD register size. If the argument has a type that is not vectorizable, it is `nonvectorizable`/`uniform`. This is the case for `void` pointer types, for example.

All other arguments may have different values for each instance, hence they are considered `varying`. To be more precise, `varying` arguments are either `unknown` or `consecutive`/`aligned`: Normal values are marked as `unknown` to reflect that no information about the different values of the different instances is available. Pointers are `consecutive` and `aligned`, again because we assume that all pointers supplied to the function are aligned.

Note that the algorithm requires the argument types to either match exactly, or the SIMD type has to be a "struct of array" version of the scalar type. For example, a vectorized struct type is not a vector of structs with scalar elements but a struct of vectorized elements. Table 5.3 shows the type vectorization rules that apply.

User-Defined Marks

$$
[\![x \leftarrow \texttt{user mark}]\!]^\sharp(D_{Abs}, B, A, L) = (D_{Abs}, B, A, L) \oplus \{x \mapsto \texttt{user mark}\}
$$

It is possible to add SIMD semantics to arbitrary program points to force certain behavior. Most commonly, this is used to add information about other functions that are called to allow for more efficient code generation. For example, the WFV implementation includes optimized vector variants

Table 5.3 "Struct-of-array" type vectorization rules of the most important types (LLVM notation), applied by the function `vect`$_{\mathtt{W}}$, where W is the chosen vectorization factor. Note that we do not allow vectors of pointers, and thus do not allow vectorization of the void-pointer type `i8*`.

Scalar Type	Vector Type
vect$_{\mathtt{W}}$(i1)	<W x i1>
vect$_{\mathtt{W}}$(i8)	<W x i8>
vect$_{\mathtt{W}}$(i32)	<W x i32>
vect$_{\mathtt{W}}$(i64)	<W x i64>
vect$_{\mathtt{W}}$(float)	<W x float>
vect$_{\mathtt{W}}$(type*)	<W x type>*
vect$_{\mathtt{W}}$(i8*)	N/A
vect$_{\mathtt{W}}$([N $\times$ type])	[N $\times$ vect$_{\mathtt{W}}$(type)]
vect$_{\mathtt{W}}$({ type0, type1, ... })	{ vect$_{\mathtt{W}}$(type0), vect$_{\mathtt{W}}$(type1), ... }

for library functions like `sin`, `cos`, `exp`, etc., that efficiently compute the corresponding function for multiple input values at once. Via user-defined marks, the algorithm is made aware that a scalar call to one of these functions can be vectorized by replacing it with a call to the corresponding vector variant.

These user-defined marks are always obeyed by the analyses. In case a mark is inconsistent with other results of the analysis, a concrete implementation has to return a compilation error. For example, if the analysis marks a value `varying` that is used by an operation that the user marked as `uniform`, code generation would not be possible.

5.6.3 Instance Identifier

$$[\![x \leftarrow \mathtt{tid}]\!]^{\sharp}(D_{Abs}, B, A, L) = (D_{Abs} \oplus \{x \mapsto \mathtt{ca}\}, B, A, L)$$

As defined in Section 5.1, the function `tid` returns the instance identifier of the running instance. By definition, this value is `consecutive` and `aligned` because we consider instances to always be grouped together in order of their instance identifiers. Thus, the identifiers of a group of W instances start with a value that is divisible by W, and each subsequent identifier is incremented by one.

As an example, languages like OpenCL or CUDA that have specific code constructs to query the instance identifier. For example, in OpenCL, the function `get_global_id` corresponds to `tid`.

5.6.4 Constants

$$
[\![x \leftarrow c]\!]^{\sharp}(D_{Abs}, B, A, L) =
$$
$$
\left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{ua} & \text{if } c = mS,\ m \in \mathbb{N} \\ \texttt{nu} & \text{if } \texttt{type(c)} \text{ not vectorizable} \\ \texttt{u} & \text{otherwise} \end{cases} \right\}, B, A, L\right)
$$

Constants are `uniform` by definition. In addition, if a constant is divisible by the SIMD width S, it is `uniform/aligned`. If the constant is of a type that is not vectorizable, it is `nonvectorizable/uniform`.

5.6.5 Phi Functions

$$
[\![x \leftarrow \texttt{phi}((v_1, b_1), (v_2, b_2))]\!]^{\sharp}(D_{Abs}, B, A, L) =
$$
$$
\left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} D_{Abs}(v_1) \sqcup_{\texttt{phi}} D_{Abs}(v_2) & \text{if } x \notin B \\ \texttt{k} & \text{if } x \in B \wedge D_{Abs}(v_1) \sqsubseteq \texttt{kg} \\ & \qquad \wedge D_{Abs}(v_2) \sqsubseteq \texttt{kg} \\ \texttt{ns} & \text{otherwise} \end{cases} \right\},\right.
$$
$$
\forall d \in D_{Abs}.(d \text{ is } \texttt{branch} \wedge d \sqsupseteq \texttt{ca}) \implies
$$
$$
B \oplus \left\{ x \mapsto \begin{cases} \texttt{blend}_d & \text{if } \texttt{disjointPaths}(d,x) \\ B(x) & \text{otherwise} \end{cases} \right\},
$$
$$
\left. A, L\right)
$$

The properties of a ϕ-function depend both on the incoming values and the incoming control flow paths. If instances diverged at a `varying` branch d, they may join again at the program point of this ϕ-function. In such a case, control-flow to data-flow conversion is required: It is not sufficient to select one of the values of the incoming paths, they have to be blended. Thus, such a *join point* has to be marked $\texttt{blend}_d$, and the map B has to be updated.

Definition 11 (Join Point) *Let b be a program point with multiple incoming edges $p_1 \to b, \ldots, p_n \to b$. Let d be another,* `varying` *program point with outgoing edges $d \to s_1, \ldots, d \to s_n$. b is* $\texttt{blend}_d$ *iff there exist two* disjoint *paths $s_a \to^* p_x$ and $s_b \to^* p_y$ with $x \neq y$.*

Informally, this describes the fact that in a given SIMD execution, b can be reached by some instances from *both* edges that leave d. Hence, b is subject to control-flow to data-flow conversion. Figure 5.6 illustrates the disjoint path criterion.

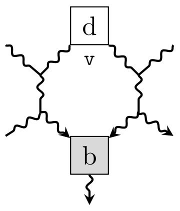


Figure 5.6: Illustration of the blend criterion (Definition 11). Note that, for b to be `blend`$_d$, b is neither required to post-dominate d, nor is d required to dominate b.

The property is determined with the function `disjointPaths`, which is defined as follows:

$$\mathit{disjointPaths}(d, x) = \exists s_1, s_2 \in \texttt{succ}(d).\texttt{disjointPaths}(s_1, s_2, x),$$

where `succ`(d) is the set of successor program points of d. The question whether there are disjoint paths from two definitions s_1, s_2 to a program point x is the same that is asked during construction of SSA form. Its answer is given easiest by testing whether x is in the *iterated dominance frontier* of the two start blocks. During SSA construction, these join points are the locations where ϕ-functions have to be placed.

If the program point of the ϕ-function is `blend`, it cannot be `uniform`. If its type is vectorizable, it is `unknown`, otherwise it is `nonvectorizable`.

If the program point is not `blend`, no diverged control flow can join again at this point. This means that the ϕ-function does not have to be transformed into a `blend` operation. The operator $\sqcup_{\texttt{phi}}$ that determines the mark of the ϕ-function in this case is defined as follows (symmetric values are omitted):

$\sqcup_{\texttt{phi}} =$

$D_{Abs}(v_1), D_{Abs}(v_2)$	ua	u	nu	ca$_-$	c$_-$	k$_-$	n$_-$
ua	ua	u	nu	k	k	k	ns
u		u	nu	k	k	k	ns
nu			nu	ns	ns	ns	ns
ca$_-$				ca	c	k	ns
c$_-$					c	k	ns
k$_-$						k	ns
n$_-$							ns

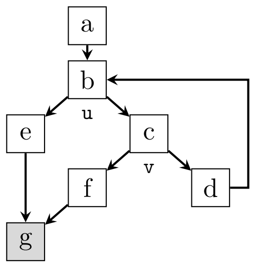


Figure 5.7: The CFG of the `Mandelbrot` kernel from the introduction. Uniformity of exit conditions is shown with a lowercase letter below the block. The block g is a `blend`$_c$ block because instances may diverge at the `varying` branch in c and join again in g. The corresponding disjoint paths are $c \to f \to g$ and $c \to d \to b \to e \to g$.

Note that the ϕ-function can still be `varying`, e.g. if the incoming value from either path is `consecutive` or `unknown`. In such a case, although it returns either of the incoming values and does not have to be replaced by a `blend` operation, it can't be `uniform` either. In a case where only one incoming value is `uniform`, the transfer function has to be conservative and mark the ϕ-function at least `consecutive`, sacrificing some precision.

Along with the `blend` mark, the program points that caused the mark are also stored. We denote this by referring to `blend`$_d$ if the branch at program point d caused the `blend` mark. This information is used by the Rewire Target Analysis (Section 5.9). This property can later be used to determine paths that have to be linearized (Section 6.3).

Example. Figure 5.7 shows the CFG of the `Mandelbrot` kernel from the introduction with annotated uniformity of the branches. The CFG contains a loop with two exits, of which the one in block c is controlled by a `varying` branch. Instances of a SIMD group may diverge at this branch, and the instances that remain in the loop may at some point leave it over the other exit in b. The instances join again in block g. This makes it necessary to blend the results of the different instances together at this point, so corresponding ϕ-functions in g have to be transformed to `blend` operations during vectorization. Thus, the program points that correspond to ϕ-functions in block g have to be marked as `blend`. We say that g is a `blend` block.

LCSSA Phis

$$\llbracket x \leftarrow \texttt{lcssaphi}(v, b) \rrbracket^\sharp(D_{Abs}, B, A, L) =$$

$$\left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{ns} & \text{if } x \in L \wedge D_{Abs}(v) \sqsupseteq \texttt{nu} \\ \texttt{ns} & \text{if } x \notin L \wedge D_{Abs}(v) \sqsupseteq \texttt{ns} \\ \texttt{k} & \text{if } x \notin L \wedge \texttt{kg} \sqsupseteq D_{Abs}(v) \sqsupseteq \texttt{k} \\ \texttt{c} & \text{if } x \notin L \wedge \texttt{cg} \sqsupseteq D_{Abs}(v) \sqsupseteq \texttt{c} \\ \texttt{ca} & \text{if } x \notin L \wedge \texttt{cag} \sqsupseteq D_{Abs}(v) \sqsupseteq \texttt{ca} \\ \texttt{k} & \text{otherwise,} \end{cases} \right\}, B, A, L\right)$$

The `consecutive`, `aligned`, and `unknown` properties of an LCSSA phi are transferred from the incoming value if no loop that is left over the corresponding edge is `divergent`. This does not count for the `sequential` and `guarded` properties, since they only describe the program point of the incoming value and do not require the phi itself to also be `sequential` or `guarded`. If one of the loops that is left is `divergent`, then the phi is `nonvectorizable/sequential` if the incoming value is `nonvectorizable`, and `unknown` otherwise. This is because values that are live across exits of `divergent` loops may be different for different instances, so to be conservative they must not be `uniform`.

5.6.6 Memory Operations

While SIMD instruction sets provide memory operations that operate on vectors, a scalar `load` or `store` cannot blindly be replaced by a vector `load` or `store`, and an `alloca` cannot simply be changed to allocate W times as much memory. This has two reasons: First, `store` operations produce side effects that must not occur for inactive instances. Second, the pointer operands of the instructions may not point to adjacent memory locations, but vector memory instructions typically only allow to operate on consecutive addresses.

Alloca

$$\llbracket x \leftarrow \texttt{alloca}(n) \rrbracket^\sharp(D_{Abs}, B, A, L) =$$

$$\left(D_{Abs} \oplus \left\{ x \mapsto \bigsqcup_{\texttt{alloca}} \{D_{Abs}(u) | u \in \texttt{uses}(x)\} \right\}, B, A, L\right)$$

The operator $\sqcup_{\texttt{alloca}}$ is defined as follows:

$\sqcup_{\texttt{alloca}} =$

$D_{Abs}(u_1), D_{Abs}(u_2)$	u_	nu	n_	_s	_g
u_	ua	nu	ns	cas	cas
nu		nu	ns	ns	ns
n_			ns	ns	ns
_s				cas	cas
_g					cas

else `ca`

`Alloca` operations only have to be vectorized if any use is not `uniform`. The operation is always aligned since the memory is allocated locally. If the type is not vectorizable, the `alloca` is `nonvectorizable`, and `sequential` or `uniform`. If the type of the `alloca` is a compound type that contains a nested pointer, and it has at least one `sequential` use, it has to be `sequential`.[1] This is required to correctly handle nested pointers. Consider the following scalar code, where the function `some_fn` initializes the pointer in the given structure:

```
%struct.A = type { i32* }
...
%A = alloca %struct.A
call @some_fn(%struct.A* %A)
...
```

If the call does not have a vector equivalent, it has to be executed sequentially (assume it does not have side effects so we don't need guards). If the `alloca` is not split into W scalar operations, temporary structs of the scalar type `%struct.A` need to be allocated for the call (similar to Listing 3.3 in Section 3.3). This is because the call is executed sequentially and thus requires scalar arguments. After the calls, the temporaries have to be written back to the actual struct (`%A`). However, for the nested pointer this is not possible, since the two calls may return two different pointers. Thus, we have to prevent this writeback operation by splitting the `alloca`, i.e., it has to be marked `sequential`. Note that this is often the better choice for performance reasons as well because no temporary structs with extract and write back operations are required (recall the example from Listing 3.3). This, however, depends on how many other uses of the `alloca` are not `sequential` because these require the vectorized struct as input.

[1] For the sake of a more concise presentation, the given definition of $\sqcup_{\texttt{alloca}}$ is more conservative in that it ignores the type of the `alloca`.

Load

$$[\![x \leftarrow \texttt{load}(a)]\!]^\sharp(D_{Abs}, B, A, L) =$$

$$(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{u} & \text{if } D_{Abs}(a) \sqsubseteq \texttt{nu} \\ \texttt{k} & \text{if } \texttt{cg} \sqsupseteq D_{Abs}(a) \sqsupseteq \texttt{ca} \land x \in A \\ \texttt{ks} & \text{if } D_{Abs}(a) \sqsupseteq \texttt{k} \\ \texttt{ns} & \text{if } D_{Abs}(a) \sqsupseteq \texttt{ns} \end{cases} \right\}, B, A, L)$$

A `load` instruction can be replaced by a vector `load` if its address operand is `consecutive`/`aligned`. If it is not `aligned`, a slightly less efficient, unaligned vector `load` can be used. If the address operand is `unknown`, the `load` is a so-called *gather* operation: It may load from non-contiguous memory locations and thus has to be executed sequentially.

If the `load` is not `by_all`, there are two options: First, the `load` of each instance can be executed sequentially and guarded. Second, the `load` can be performed for all instances regardless of whether they are active or not. Since the value is blended when the instances join again, the loaded value is discarded anyway. In some cases, this way of blindly executing all W values may be more efficient than executing W branch statements followed by the `loads`. However, this option may not always be safe: it loads from an address that belongs to an inactive instance, so it may not point to valid data. This may be explicitly allowed by the source languages semantics, for instance.

The result of the `load` is never `consecutive`, since we do not reason about contents of memory. A straightforward way to improve precision would be to employ alias analysis to track properties of values in memory.

Store

$$[\![x \leftarrow \texttt{store}(a, v)]\!]^\sharp(D_{Abs}, B, A, L) =$$

$$(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{u} & \text{if } D_{Abs}(a) \sqsubseteq \texttt{nu} \land D_{Abs}(v) \sqsubseteq \texttt{nu} \\ \texttt{k} & \text{if } x \in A \land \texttt{cg} \sqsupseteq D_{Abs}(a) \sqsupseteq \texttt{ca} \land D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{ks} & \text{if } x \in A \land D_{Abs}(a) \sqsupseteq \texttt{k} \land D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{kg} & \text{if } x \notin A \land D_{Abs}(a) \sqsupseteq \texttt{k} \land D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{ns} & \text{otherwise, if } x \in A \\ \texttt{ng} & \text{otherwise, if } x \notin A \end{cases} \right\},$$
$$B, A, L)$$

A vector `store` can be used if the address operand is `consecutive` and the program point is `by_all`. If the address is not `aligned`, an unaligned memory access has to be used. This is slower than the aligned variant, but still avoids sequential, scalar execution.

If the address operand is `unknown`, the `store` conservatively has to be expected to be a so-called *scatter* operation: It may write to non-contiguous memory locations and thus the `store` operation of each instance has to be executed separately. If the `store` is not `by_all` or its address or value operand are not vectorizable, it also has to be executed sequentially. Also, if it is not `by_all`, it has to be guarded by conditionals.

Finally, a `uniform` address operand of a `store` effectively describes a race condition: All instances write to the same memory location. However, if the stored value is also `uniform`, the same value is written to the same address multiple times, so the effect may not be visible.

5.6.7 Calls

$$
\begin{aligned}
&[\![x \leftarrow \texttt{call}(g, v_1, \ldots, v_n)]\!]^\sharp(D_{Abs}, B, A, L) = \\
&\quad \left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases}
\texttt{u} & \text{if } g \in F_{pure} \wedge \forall v. D_{Abs}(v) \sqsubseteq \texttt{nu} \wedge g \in R_v \\
\texttt{nu} & \text{if } g \in F_{pure} \wedge \forall v. D_{Abs}(v) \sqsubseteq \texttt{nu} \wedge g \notin R_v \\
\texttt{k} & \text{if } g \in F_{mapped} \wedge \exists v. D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge \nexists v. D_{Abs}(v) \sqsupseteq \texttt{ns} \\
\texttt{ks} & \text{if } (x \in A \vee f \in F_{pure}) \wedge g \in R_v \wedge \\
& \quad ((\exists v. D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge g \notin F_{mapped}) \vee (\exists v. D_{Abs}(v) \sqsupseteq \texttt{ns})) \\
\texttt{ns} & \text{if } (x \in A \vee f \in F_{pure}) \wedge g \notin R_v \wedge \\
& \quad ((\exists v. D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge g \notin F_{mapped}) \vee (\exists v. D_{Abs}(v) \sqsupseteq \texttt{ns})) \\
\texttt{kg} & \text{if } (x \notin A \wedge f \notin F_{pure}) \wedge g \in R_v \wedge \\
& \quad ((\exists v. D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge g \notin F_{mapped}) \vee (\exists v. D_{Abs}(v) \sqsupseteq \texttt{ns})) \\
\texttt{ng} & \text{if } (x \notin A \wedge f \notin F_{pure}) \wedge g \notin R_v \wedge \\
& \quad ((\exists v. D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge g \notin F_{mapped}) \vee (\exists v. D_{Abs}(v) \sqsupseteq \texttt{ns}))
\end{cases} \right\}, \right. \\
&\quad B, A, L)
\end{aligned}
$$

Here, $g \in R_v$ means that the return type of function g is vectorizable. If $g \in F_{pure}$, g is a *pure* function.[2] If $g \in F_{mapped}$, there exists a mapping of g to a vector implementation. Note that the existence of such a mapping implies that the function has a vectorizable return type.

There are three kinds of function calls that do *not* have to be executed sequentially. First, pure functions with only `uniform` operands are `uniform`. Second, "known" functions such as `sin`, `sqrt`, or `ceil` are mapped directly to specialized vector implementations for the target architecture if available. They are never marked `sequential` or `guarded` since they do not produce side effects. Third, if the callee's code is available, WFV can be applied recursively. The resulting vectorized function receives an optional mask

[2] *Pure* functions are guaranteed not to produce side effects and always evaluate to the same result when called with the same parameters.

argument if the call is in a block that is not `by_all`. Finally, the user can specify custom mappings, e.g. if a hand-optimized vector implementation of a function exists. Examples are calls to `get_global_id` in OpenCL or calls to `traceRay` in RenderMan.

The target function of this mapping can also specify a mask argument. Without mask argument, user-defined mappings are ignored if the parent block of the call is not `by_all`, yielding a `guarded` mark. This behavior was omitted from the transfer function to not complicate it further.

Note that vectorization of recursive function calls is possible via specification of a custom mapping.

5.6.8 Cast Operations

$$
\begin{aligned}
&[\![x \leftarrow \texttt{castop}(v)]\!]^\sharp(D_{Abs}, B, A, L) = \\
&\quad \left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases}
\texttt{u} & \text{if } \texttt{type(x)} \text{ vectorizable} \wedge D_{Abs}(v) \sqsubseteq \texttt{nu} \\
\texttt{nu} & \text{if } \texttt{type(x)} \text{ not vectorizable} \wedge D_{Abs}(v) \sqsubseteq \texttt{nu} \\
\texttt{ns} & \text{if } \texttt{type(x)} \text{ not vectorizable} \wedge D_{Abs}(v) \sqsupseteq \texttt{ca} \\
\texttt{ks} & \text{if } \texttt{type(x)} \text{ vectorizable} \wedge D_{Abs}(v) \sqsupseteq \texttt{ns} \\
\texttt{k} & \text{if } \texttt{type(x)} \text{ vectorizable} \wedge (\texttt{k} \sqsupseteq D_{Abs}(v) \sqsubseteq \texttt{kg} \vee \\
& \texttt{type(x)} \text{ is pointer} \wedge \\
& \texttt{elemtysize(x)} \neq \texttt{elemtysize(v)}) \\
\texttt{c} & \text{if } \texttt{c} \sqsupseteq D_{Abs}(v) \sqsubseteq \texttt{cg} \wedge (\texttt{type(x)} \in \{\texttt{int}, \texttt{float}\} \vee \\
& \texttt{type(x)} \text{ is pointer} \wedge \\
& \texttt{elemtysize(x)} = \texttt{elemtysize(v)}) \\
\texttt{ca} & \text{if } \texttt{ca} \sqsupseteq D_{Abs}(v) \sqsubseteq \texttt{cag} \wedge (\texttt{type(x)} \in \{\texttt{int}, \texttt{float}\} \vee \\
& \texttt{type(x)} \text{ is pointer} \wedge \\
& \texttt{elemtysize(x)} = \texttt{elemtysize(v)})
\end{cases} \right\}, \right. \\
&\quad B, A, L)
\end{aligned}
$$

The function `type(x)` returns the target type and `elemtysize(v)` returns the size of the type of the value that the pointer v points to.

Precise information of the shape of the source operand (such as `consecutive`) can only be transferred to the program point of the cast if the target type is either `int` or `float`. Otherwise, the mark of the program point depends on whether the target type is `nonvectorizable`, and whether the source operand is `uniform`. For example, if the source operand is `nonvectorizable`, the cast has to be `sequential` since it cannot be represented by a vector operation.

Casting a value from one pointer type to another can change the access pattern if the pointer is used by a `load`. For example, if casting a `consecutive/aligned` pointer of type `i32*` to `i16*`, the access becomes `strided`.

As another example, if casting from type `i32*` to `i64*`, the access may become unaligned, depending on the original alignment. For simplicity reasons, we therefore resort to `unknown` for cases where the size of the target element type does not match the size of the source type.

5.6.9 Arithmetic and Other Instructions

All of the operators below by definition have vectorizable types, so cases with `nu`, `ns`, and `ng` are omitted. For the sake of simplicity, we skip function values with $\texttt{k}_{\_}$ arguments. They all yield `k`.

Additive operator $\omega \in \{\texttt{add}, \texttt{gep}\}$

$$[\![x \leftarrow \omega(v_1, v_2)]\!]^{\sharp} D_{Abs} = D_{Abs} \oplus \left\{ x \mapsto \begin{array}{|c|cccc|} \hline D_{Abs}(v_1), D_{Abs}(v_2) & \texttt{ua} & \texttt{u} & \texttt{ca}_{\_} & \texttt{c}_{\_} \\ \hline \texttt{ua} & \texttt{ua} & \texttt{u} & \texttt{ca} & \texttt{c} \\ \texttt{u} & & \texttt{u} & \texttt{c} & \texttt{c} \\ \texttt{ca}_{\_} & & & \texttt{k} & \texttt{k} \\ \texttt{c}_{\_} & & & & \texttt{k} \\ \hline \end{array} \text{ else } \texttt{k} \right\}$$

Subtraction

$$[\![x \leftarrow \texttt{sub}(v_1, v_2)]\!]^{\sharp} D_{Abs} = D_{Abs} \oplus \left\{ x \mapsto \begin{array}{|c|cccc|} \hline D_{Abs}(v_1), D_{Abs}(v_2) & \texttt{ua} & \texttt{u} & \texttt{ca}_{\_} & \texttt{c}_{\_} \\ \hline \texttt{ua} & \texttt{ua} & \texttt{u} & \texttt{k} & \texttt{k} \\ \texttt{u} & \texttt{u} & \texttt{u} & \texttt{k} & \texttt{k} \\ \texttt{ca}_{\_} & \texttt{ca} & \texttt{c} & \texttt{ua} & \texttt{u} \\ \texttt{c}_{\_} & \texttt{c} & \texttt{c} & \texttt{u} & \texttt{u} \\ \hline \end{array} \text{ else } \texttt{k} \right\}$$

Multiplication

$$[\![x \leftarrow \texttt{mul}(v_1, v_2)]\!]^{\sharp} D_{Abs} = D_{Abs} \oplus \left\{ x \mapsto \begin{array}{|c|cc|} \hline D_{Abs}(v_1), D_{Abs}(v_2) & \texttt{ua} & \texttt{u} \\ \hline \texttt{ua} & \texttt{ua} & \texttt{ua} \\ \texttt{u} & & \texttt{u} \\ \hline \end{array} \text{ else } \texttt{k} \right\}$$

Other arithmetic and comparison operators:

$$[\![x \leftarrow \texttt{op}(v_1, v_2)]\!]^{\sharp} D_{Abs} = D_{Abs} \oplus \left\{ x \mapsto \begin{array}{|c|cc|} \hline D_{Abs}(v_1), D_{Abs}(v_2) & \texttt{ua} & \texttt{u} \\ \hline \texttt{ua} & \texttt{u} & \texttt{u} \\ \texttt{u} & & \texttt{u} \\ \hline \end{array} \text{ else } \texttt{k} \right\}$$

Other operations without side effects:

$$[\![x \leftarrow \mathtt{op}(v_1, \ldots, v_n)]\!]^\sharp D_{Abs} = D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \mathtt{u} & \text{if } \forall v. D_{Abs}(v) \sqsubseteq \mathtt{nu} \\ \mathtt{k} & \text{otherwise} \end{cases} \right\}$$

Integer Division and Modulo

For integer division and modulo, there are two special cases when the second operand is a constant: Integer division of a `consecutive/aligned` value by a multiple of W produces a `uniform` value. For example,

$$\langle\ 12, 13, 14, 15\ \rangle\ \mathit{div}\ \langle\ 8, 8, 8, 8\ \rangle = \langle\ 1, 1, 1, 1\ \rangle.$$

The result of a `consecutive/aligned` value modulo a multiple of W produces a `consecutive/aligned` value again. For example,

$$\langle\ 12, 13, 14, 15\ \rangle\ \mathit{mod}\ \langle\ 8, 8, 8, 8\ \rangle = \langle\ 4, 5, 6, 7\ \rangle.$$

Neutral Elements

Operations which include their neutral elements as one operand (such as 1 for multiplication or 0 for additive operations) have not been denoted explicitly. If any operand is the neutral element of the operation, the operation receives the mark of the other operand (without `sequential` or `guarded` properties, similar to ω).

Inverted Consecutive Values

Subtraction of a `consecutive` value from a `uniform` value produces an *inverted* `consecutive` value. This information could still be beneficial during analysis and code generation: If such an operation is followed by a subtraction with another inverted `consecutive` value, this again produces a `uniform` value. During code generation, a `load` from an inverted `consecutive` value can be implemented with a shuffle and vector `load`, which is much more efficient than sequential operations. For presentation reasons, we resort to `unknown` here. However, the corresponding modification of the lattices and transfer functions is straightforward.

Strided Values

For addition, multiplication, and pointer cast operations, it may make sense to define a `strided` value. This is because multiplication of a `consecutive` value with a `uniform` value produces a vector with constant stride. For example,

$$\langle\ 5, 6, 7, 8\ \rangle \times \langle\ 2, 2, 2, 2\ \rangle = \langle\ 10, 12, 14, 16\ \rangle.$$

Similarily, an addition of two `consecutive` values produces constant stride values. For example,

$$\langle\ 5, 6, 7, 8\ \rangle + \langle\ 3, 4, 5, 6\ \rangle = \langle\ 8, 10, 12, 14\ \rangle.$$

Again, this would improve precision of the analysis: For multiplication, a division of the result by the same `uniform` value produces a `consecutive` value again. For addition, a subtraction of any `consecutive` value from the result or a division by the constant 2 produces a `consecutive` value again. If deemed beneficial, the corresponding modification of the lattices and transfer functions is straightforward.

Note that `strided` can be exploited during code generation [Nuzman et al. 2006], but this requires tracking the stride interval: For an interval of 2, two vector `loads` and a `shuffle` operation can be used instead of sequential, scalar `loads`. A more involved technique could scan for multiple subsequent `strided load` or `store` operations that together access all elements in a certain range. This can be exploited by vector `loads` and `shuffles` as well and is much more efficient than a series of sequential operations.

Optimization of Pack/Unpack Operations

There is an optimization problem to solve in cases where multiple subsequent operations require packing and unpacking of values: It might be more profitable to execute some operations sequentially although they are not `sequential`, `guarded`, or `nonvectorizable`. This is because the cost for packing operands and unpacking results can be higher than the gain due to the vector instruction. This is subject to a heuristic, e.g. as presented by Kim & Han [2012]. The algorithm would then treat some program points as `sequential` that do not necessarily require to be executed sequentially.

5.6.10 Branch Operation

$$[\![x \leftarrow \texttt{branch}(v, b_1, b_2)]\!]^\sharp(D_{Abs}, B, A, L) =$$
$$(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{u} & \text{if } D_{Abs}(v) \sqsubseteq \texttt{nu} \\ \texttt{k} & \text{otherwise} \end{cases} \right\}, B, A, L)$$

By definition, the type of the branch condition is vectorizable, and a branch has no side effects, so it is never `nonvectorizable`, `sequential`, or `guarded`. Since a branch returns a block identifier rather than a value, it cannot be `consecutive` or `aligned`, either. Thus, the branch is `uniform` if its condition is `uniform`, and `unknown` (equivalent to ¬`uniform` and `varying`) otherwise.

5.6.11 Update Function for All-Active Program Points

The functions use the set A which captures information about `by_all` nodes: nodes that are always executed with *all instances active*, e.g. at the entry point of the function. Definition 7 is a dynamic property that may depend on input values. The following definition describes a safe underapproximation of *`by_all`* program points which can be statically proven:

Definition 12 (Static All-Active Program Point) *If the function has an initial mask parameter, no program point is* `by_all`*. Otherwise, the function entry point e by definition is* `by_all`*. Let b be a program point that is* control dependent *on a set of program points C. b is* `by_all` *iff all blocks in C are* `by_all` *and end with a* `uniform` *branch.*

If, on a path that starts at a `by_all` program point, instances diverge at `varying` program point v, but the postdominator of v is reached on all paths to b, b is not control-dependent on v, and thus may still be `by_all`. This is because all paths that have diverged at v always reconverge before b is reached. Notice that this definition implicitly prevents that any program point inside a `divergent` loop is marked `by_all`.

To determine whether a program point is `by_all`, every program point holds an additional transfer function that may add the corresponding node to A. Let b be a program point, and let e be the program point of the function entry node. The update function for A at node b is

$$A' = A \cup \begin{cases} b & \text{if } \forall c \in \texttt{controlDeps}(b). c \in A \wedge D_{Abs}(c) \sqsubseteq \texttt{nu} \\ \emptyset & \text{otherwise,} \end{cases}$$

where `controlDeps` returns the set of program points that b is control dependent on. Initially, the set A is empty if there is a mask parameter. Otherwise, it contains the entry point.

5.6.12 Update Function for Divergent Loops

The functions use the set L which captures information about nodes that are part of the body of a `divergent` loop: Again, Definition 8 is a dynamic property that may depend on input values. The following definition describes an overapproximation of *loop divergence* which can be statically proven:

Definition 13 (Static Loop Divergence) *Let l be a loop, and let v be a* `varying` *program point inside the body of l. The loop l is* `divergent` *if v is not strictly postdominated by another program point that belongs to l.*

Intuitively, Definition 13 describes that a loop is `divergent` if there is an "escaping path" from a `varying` branch to a loop exit. Over this path, some instances may leave the loop at one exit, while others that diverged at this branch may go to a different one or keep iterating. For example, the program points in the loop of the `Mandelbrot` kernel (Figure 5.7) are `divergent` because of the `varying` branch in block c. At this branch, some instances may leave the loop while others continue iterating.

To determine whether a program point is `divergent`, the program point that corresponds to the loop entry holds an additional transfer function that may add the nodes of the loop body to L.

Let ℓ be the set of all program points within the body of a loop. The update function for L at the node that corresponds to the loop entry point is

$$L' = L \cup \begin{cases} \ell & \text{if } \exists v \in \ell.\text{v is } \texttt{branch} \wedge D_{Abs}(v) \sqsupseteq \texttt{ca} \wedge \\ & \quad \nexists b \in \ell.b \neq v \wedge b \in \texttt{postdom}(v) \\ \emptyset & \text{otherwise.} \end{cases}$$

Initially, the set L is empty. We chose this definition to be coherent with the rest of this section. For practical reasons, a concrete implementation would store the `divergent` property for loop objects instead of program points. The query then first determines whether a program point is part of a loop, and then tests whether that loop is `divergent`.

If the Rewire Target Analysis is disabled, all loops are conservatively marked `divergent`.

$$\begin{tikzcd}
\cdot \arrow[r, "{[\![\ ]\!]^\sharp}"] \arrow[dd, "\gamma"'] & \cdot \arrow[d, "\gamma"] \\
 & \cdot \arrow[d, phantom, "\rotatebox{90}{$\leq$}"] \\
\cdot \arrow[r, "{[\![\ ]\!]^C}"'] & \cdot
\end{tikzcd}$$

Figure 5.8: Visualization of the local consistency property of the abstract transformer $[\![\,]\!]^\sharp$ with the concretization function $\gamma : D_{Abs} \to D$. The transformer $[\![\,]\!]^C$ of CS applies the transformer of OS to every state $d \in D$.

5.7 Soundness

We show the soundness of the abstraction by sketching a proof of the *local consistency* of the Abstract Semantics AS with respect to the Operational Semantics OS. To this end, we define the *concretization function* $\gamma : D_{Abs} \to D$ that maps states of AS to states of the implicitly defined Collecting Semantics of OS. To define γ, we first introduce a set of properties similar to the SIMD properties of AS:

$$\begin{array}{rcl}
\texttt{uni}(d,x) & = & \forall x_i \in \rho(x). x_i = x_{i_0} \\
\texttt{aligned}(d,x) & = & \rho_0(x) \,\%\, S = 0 \\
\texttt{consec}(d,x) & = & \forall i_j \in \#. j \neq 0 \implies \rho_{i_j}(x) = \rho_{i_{j-1}}(x) + 1 \\
\texttt{unknown}(d,x) & = & (\neg\texttt{uni}(d,x) \wedge \neg\texttt{consec}(d,x)) \vee (\texttt{uni}(d,x) \wedge \texttt{consec}(d,x)) \\
\texttt{vtype}(x) & = & \text{operation } x \text{ has vectorizable result type} \\
\texttt{novec}(x) & = & \text{operation } x \text{ has no vector equivalent} \\
\texttt{se}(x) & = & \text{operation } x \text{ may have side effects} \\
\texttt{nvop}(x) & = & \exists y \in \texttt{operands}(x). \neg\texttt{vtype}(y)
\end{array}$$

Note that these properties are defined on the *vector semantics* of OS, i.e., each value in ρ is a vector. Accesses to values of an individual instance i are denoted as $\rho_i(x)$ or x_i.

5.7.1 Local Consistency

The concretization function $\gamma : D_{Abs} \to D$ maps from the Abstract Semantics to the Collecting Semantics:

$$\gamma(D_{Abs}) = \bigcup_{x \in D_{Abs}} \gamma(D_{Abs}, x),$$

$$\gamma(D_{Abs}, x) =$$
$$\begin{cases}
\{d|\texttt{uni}(d,x) \wedge \texttt{aligned}(d,x_{i_0})\} & \text{if } D_{Abs}(x) = \texttt{ua} \\
\{d|\texttt{uni}(d,x) \wedge \neg\texttt{aligned}(d,x_{i_0})\} & \text{if } D_{Abs}(x) = \texttt{u} \\
\{d|\texttt{consec}(d,x) \wedge \texttt{aligned}(d,x_{i_0})\} & \text{if } D_{Abs}(x) = \texttt{ca} \\
\{d|\texttt{consec}(d,x) \wedge \texttt{aligned}(d,x_{i_0}) \wedge (\texttt{novec}(x) \vee \texttt{nvop}(x)\} & \text{if } D_{Abs}(x) = \texttt{cas} \\
\{d|\texttt{consec}(d,x) \wedge \texttt{aligned}(d,x_{i_0}) \wedge \texttt{se}(x)\} & \text{if } D_{Abs}(x) = \texttt{cag} \\
\{d|\texttt{consec}(d,x) \wedge \neg\texttt{aligned}(d,x_{i_0})\} & \text{if } D_{Abs}(x) = \texttt{c} \\
\{d|\texttt{consec}(d,x) \wedge \neg\texttt{aligned}(d,x_{i_0}) \wedge (\texttt{novec}(x) \vee \texttt{nvop}(x))\} & \text{if } D_{Abs}(x) = \texttt{cs} \\
\{d|\texttt{consec}(d,x) \wedge \neg\texttt{aligned}(d,x_{i_0}) \wedge \texttt{se}(x)\} & \text{if } D_{Abs}(x) = \texttt{cg} \\
\{d|\texttt{unknown}(d,x)\} & \text{if } D_{Abs}(x) = \texttt{k} \\
\{d|\texttt{unknown}(d,x) \wedge (\texttt{novec}(x) \vee \texttt{nvop}(x))\} & \text{if } D_{Abs}(x) = \texttt{ks} \\
\{d|\texttt{unknown}(d,x) \wedge \texttt{se}(x)\} & \text{if } D_{Abs}(x) = \texttt{kg} \\
\{d|\neg\texttt{vtype}(x) \wedge \texttt{uni}(d,x)\} & \text{if } D_{Abs}(x) = \texttt{nu} \\
\{d|\neg\texttt{vtype}(x) \wedge \neg\texttt{uni}(d,x)\} & \text{if } D_{Abs}(x) = \texttt{ns} \\
\{d|\neg\texttt{vtype}(x) \wedge \neg\texttt{uni}(d,x) \wedge \texttt{se}(x)\} & \text{if } D_{Abs}(x) = \texttt{ng}
\end{cases}$$

Figure 5.9: The concretization function $\gamma : D_{Abs} \to D$. Additional clauses $\texttt{vtype}(x)$, $\neg\texttt{se}(x)$, and $\neg\texttt{novec}(x)$ are omitted in all rules that do not have the corresponding negated clause.

where $\gamma(D_{Abs}, x)$ is defined as shown in Figure 5.9: The sets μ and @ are not tracked in the Abstract Semantics, and # is never updated since it is only required to refer to the elements of different instances of the SIMD group.

The Abstract Semantics is sound if γ is *locally consistent*, i.e., if it exhibits the property

$$[\![\,]\!]^C \circ \gamma \leqslant \gamma \circ [\![\,]\!]^\sharp.$$

where "$\leqslant$" relates the precision of two abstraction states and $[\![\,]\!]^C$ is the transformer of CS that is implicitly defined as the application of the transformer of OS to every state $d \in D$. This means that for every operation $f(x)$, the concretization from AS to CS via γ yields a result that is at most as precise as the direct application of $f(x)$ in CS. Figure 5.8 depicts the consistency property.

In the following, we take a detailed look at the local consistency of three operations: $x \leftarrow \texttt{tid}$, $x \leftarrow \texttt{phi}((v_1, b_1), (v_2, b_2))$, and $x \leftarrow \texttt{store}(a, v)$.

Analogous manual inspection of the other rules shows that γ and the transformation rules shown in Sections 5.5 and 5.6 are indeed locally consistent. Hence, the information carried by the Abstract Semantics can be computed using fixed-point algorithms. Note that not all rules of the abstract transformer are monotone since `sequential` and `guarded` properties can be stripped in some rules. This reflects the fact that these properties only describe the program point without influencing its uses. Thus, the convergence of the analysis is not affected.

Note that γ ignores `blend`, `by_all`, and `divergent` information. Instead, every possible configuration of these is taken into account when testing local consistency.

Instance Identifier

First, direct application of the abstract transformer (Section 5.6.3) to the initial state D_{Abs}

$$[\![x \leftarrow \texttt{tid}]\!]^{\sharp}(D_{Abs}, B, A, L) = (D_{Abs} \oplus \{x \mapsto \texttt{ca}\}, B, A, L),$$

yields an update of x in D_{Abs} to `ca`. Concretization of this state with $\gamma(D_{Abs}, x)$ produces a set of states D_1 which only contains states in which x is bound to `consecutive/aligned` values:

$$\{d | \texttt{consec}(d, x) \wedge \texttt{aligned}(d, x_{i_0})\}.$$

On the other hand, concretization of the initial state D_{Abs} with $\gamma(D_{Abs}, x)$ produces a set of states D_2. Application of the (vector-lifted) transformer

$$[\![x \leftarrow \texttt{tid}]\!](d) = \rho \oplus \{x \mapsto \{\#_0, \dots, \#_{W-1}\}\}$$

to every state $d = (\rho, \mu, @, \#)$ of D_2 produces a set of states D_3 in which x is bound to the values of the corresponding set of instance identifier vectors $\#$. These values are by definition `consecutive` and `aligned`.

To summarize, the sets D_1 include all those sets in which x is bound to `consecutive/aligned` values. Since this includes all possible `consecutive` and `aligned` values in addition to those represented by the sets $\# \in D_3$, the concretized state is at most equally precise as the state obtained by first applying the abstract transformer. Thus, the consistency property holds.

Phi Functions

First, direct application of the abstract transformer (Section 5.6.5) to the initial state D_{Abs}

$$
\begin{aligned}
&[\![x \leftarrow \texttt{phi}((v_1, b_1), (v_2, b_2))]\!]^\sharp(D_{Abs}, B, A, L) = \\
&\quad \left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} D_{Abs}(v_1) \sqcup_{\texttt{phi}} D_{Abs}(v_2) & \text{if } x \notin B \\ \texttt{k} & \text{if } x \in B \wedge D_{Abs}(v_1) \sqsubseteq \texttt{kg} \\ & \qquad \wedge D_{Abs}(v_2) \sqsubseteq \texttt{kg} \\ \texttt{ns} & \text{otherwise} \end{cases} \right\}, \right. \\
&\quad \forall d \in D_{Abs}.(d \text{ is } \texttt{branch} \wedge d \sqsupseteq \texttt{ca}) \implies \\
&\quad B \oplus \left\{ x \mapsto \begin{cases} \texttt{blend}_d & \text{if } \texttt{disjointPaths}(d,x) \\ B(x) & \text{otherwise} \end{cases} \right\}, \\
&\quad \left. A, L \right)
\end{aligned}
$$

yields an update of D_{Abs} and B. Since the update of B does not have any direct effect on the concretization, we can ignore how it is updated. The following cases have to be distinguished to investigate the effects of the `phi` operation:

- x is `blend` and either v_1 or v_2 is `nu` or less precise.
- x is `blend` and v_1 and v_2 are more precise than `nu`.
- x is not `blend`.

In the first case, x in D_{Abs} is updated to `ns` by the abstract transformer. Concretization with $\gamma(D_{Abs}, x)$ yields a set of states D_1 which includes all states in which x is bound to a `nonvectorizable/sequential` value.

On the other hand, concretization of the initial state D_{Abs} with $\gamma(D_{Abs}, x)$ produces a set of states D_2. Application of the (vector-lifted) transformer

$$
\begin{aligned}
&[\![x \leftarrow \texttt{phi}((v_1, b_1), (v_2, b_2))]\!](d) = \\
&\quad \rho \oplus \left\{ x \mapsto \left\{ \begin{array}{ll} \rho_0(v_1) & \text{if } @_0(b_1) = \texttt{true} \\ \rho_0(v_2) & \text{if } @_0(b_2) = \texttt{true} \end{array}, \ldots, \begin{array}{ll} \rho_{W-1}(v_1) & \text{if } @_{W-1}(b_1) = \texttt{true} \\ \rho_{W-1}(v_2) & \text{if } @_{W-1}(b_2) = \texttt{true} \end{array} \right\} \right\}
\end{aligned}
$$

to every state $d \in D_2$ produces a set of states D_3 in which x is bound to vectors for which the element at index i is selected from either v_1 or v_2 depending on the state of $@_i$ of the corresponding instance. This means the values in the sets can satisfy any property because only one of v_1, v_2 must be `nu` or less precise, and none of its values may be selected. The values of D_1 cannot be less precise than `ns`, since `ng` requires the operation to have side effects, which is never the case for a `phi`. Thus, the consistency property holds.

The second case is similar, with x being updated to `k` by the abstract transformer, and yielding a set of states D_1 in which x is bound to an `unknown` value after concretization. Since the application of the transformer of OS to every state can again yield arbitrary properties as precise as or more precise than `k`, the consistency property holds as above.

In the third case, the abstract value of x after application of the abstract transformer can be any except for `ng`, depending on the values of v_1 and v_2. Because the result of the transfer function $\sqcup_{\texttt{phi}}$ is at most as precise as the least precise operand, D_3 can never be more precise than the sets obtained by application of the transformer of OS before concretization. The result of this transformer can be as precise as the most precise operand and is at least as precise as the least precise operand. Thus, the consistency property holds for this case as well.

Store Operations

First, direct application of the abstract transformer (Section 5.6.6) to the initial state D_{Abs}

$$[\![x \leftarrow \texttt{store}(a, v)]\!]^\sharp(D_{Abs}, B, A, L) =$$

$$\left(D_{Abs} \oplus \left\{ x \mapsto \begin{cases} \texttt{u} & \text{if } D_{Abs}(a) \sqsubseteq \texttt{nu} \wedge D_{Abs}(v) \sqsubseteq \texttt{nu} \\ \texttt{k} & \text{if } x \in A \wedge \texttt{cg} \sqsupseteq D_{Abs}(a) \sqsupseteq \texttt{ca} \wedge D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{ks} & \text{if } x \in A \wedge D_{Abs}(a) \sqsupseteq \texttt{k} \wedge D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{kg} & \text{if } x \notin A \wedge D_{Abs}(a) \sqsupseteq \texttt{k} \wedge D_{Abs}(v) \sqsubseteq \texttt{kg} \\ \texttt{ns} & \text{otherwise, if } x \in A \\ \texttt{ng} & \text{otherwise, if } x \notin A \end{cases} \right\}, B, A, L\right)$$

yields an update of x in D_{Abs}. However, x in this particular case is not an abstraction of actual values because the `store` does not return anything. It is an abstraction that describes properties of the operation only. Concretization with $\gamma(D_{Abs}, x)$ yields a set of states D_1 with values of x corresponding to its abstract value. However, because we do not track the state of values in memory in D_{Abs}, and because x is never used by any operation, the result of the concretization has no influence on the rest of the analysis.

On the other hand, concretization of the initial state D_{Abs} with $\gamma(D_{Abs}, x)$ produces a set of states D_2. Application of the (vector-lifted) transformer

$$[\![x \leftarrow \texttt{store}(a, v)]\!](d) =$$

$$\mu_0 \oplus \{\rho_0(a) \mapsto \rho_0(v)\}, \ldots, \mu_{W-1} \oplus \{\rho_{W-1}(a) \mapsto \rho_{W-1}(v)\}$$

to every state $d \in D_2$ produces a set of states D_3 in which for every $\mu_i \in d$, the address $\rho_i(a)$ is updated to $\rho_i(v)$ for every active instance. This state D_3 is more precise than D_1 which always produces an empty set for μ. Thus, the consistency property holds.

5.8 Improving Precision with an SMT Solver

The transfer functions presented in the previous section allow to classify a large class of memory address computations to result in `consecutive` values. However, the expression tree that corresponds to address computations may consist of arbitrary code. This will, in general, lead to undecidable problems, but is hard already for seemingly simple cases with only a few lines of code. For instance, if the address depends on input values. The analysis conservatively has to mark such addresses as `unknown` because of missing static information. This section describes how the precision of the analysis can be improved using an SMT solver [Karrenberg et al. 2013].[3] We generalize the classes for which consecutivity can be proven to linear arithmetic transformations, in particular including integer division and modulo by constants. The approach can—to a certain degree—also handle non-constant inputs. The key idea is to convert the central question "Do consecutive work items access consecutive memory addresses or not?" to a finite set of satisfiability problems in *Presburger Arithmetic* [Presburger 1929]. These problems can be solved with an off-the-shelf SMT solver. There is a variety of decision procedures and complexity results available for Presburger Arithmetic [Weispfenning 1990; and the references given there]. Our input considered here is limited to the existential fragment, for which SMT solvers, in spite of their possible incompleteness, are an interesting choice. For our practical computations we chose Z3 [De Moura & Bjørner 2008], which has the advantage to directly accept mod_k and div_k in the input.

5.8.1 Expression Trees of Address Computations

Consider the two OpenCL kernels in Listing 5.1. The kernel on the right-hand side, `FastWalshTransform`, is taken from the AMD APP SDK v2.8 (see our evaluation in Chapter 8). In this code, the array accesses depend on the value `tid` obtained from calls to `get_global_id`. It is easy to see that the `simple` kernel always accesses contiguous memory locations due to the direct use of `tid`. In contrast, the access pattern of `FastWalshTransform` is not obvious, since the memory locations are given by a more complex expression tree. Experimentally, one would observe that, depending on `step`, there is a considerable number of accesses that actually are consecutive. However, without additional optimization, the memory operations considered here would be executed sequentially. Listing 5.2 shows two vectorized versions of the `FastWalshTransform` kernel: one that would be produced

[3]This section presents joint work with Marek Košta and Thomas Sturm.

Listing 5.1 OpenCL kernels with simple (left) and complex (right) memory address computations: `tid`, `pair`, and `match`. The function `get_global_id` returns the work item identifier, which allows each work item to access different array positions.

```
__kernel void
simple(float* in,
       float* out)
{
  int tid = get_global_id();
  out[tid] = in[tid];
}
```

```
__kernel void
FastWalshTransform(float* arr,
                   int    step)
{
  int tid     = get_global_id();
  int group   = tid % step;
  int pair    = 2*step*(tid/step)
              + group;
  int match   = pair + step;
  float T1    = arr[pair];
  float T2    = arr[match];
  arr[pair]   = T1 + T2;
  arr[match]  = T1 - T2;
}
```

by conservative vectorization, and one with an optimization equivalent to what is achievable with our improved analysis.

To improve this situation, our compiler translates the expression tree that yields the memory address to a term that depends on `tid` and a possible input parameter. For example, the address of the second `load` operation of the `FastWalshTransform` kernel in Listing 5.1 is given by `arr[match]`, where the term obtained for `match` is

```
2*step * (tid / step) + (tid % step) + step.        (5.1)
```

Notice that `step` is an input value that is constant for all work items during one execution of the kernel.

5.8.2 Translation to Presburger Arithmetic

We are now going to switch to a more mathematical notation: The variable t is going to denote the `tid` and a is going to denote the input. For integer division and modulo, we introduce unary functions div_k and mod_k for $k \in \mathbb{Z} \setminus \{0\}$, which emphasizes the fact that the divisors and moduli are limited to numbers. For our example term (5.1), we obtain

$$e(t, a) = 2a \cdot \mathrm{div}_a(t) + \mathrm{mod}_a(t) + a. \tag{5.2}$$

Listing 5.2 Result of WFV manually applied at source level to the `FastWalshTransform` kernel of Listing 5.1 ($W = 2$). Left: Conservative WFV requires sequential execution. Right: WFV with our approach proves consecutivity of the memory addresses for certain values of `step`, which allows to generate a variant with more efficient code.

```
__kernel void
FastWalshTransform(float* a,
                   int    step)
{
  int tid = get_global_id();
  if (tid % 2 != 0) return;
  int2 tiV = (int2)(tid,tid+1);
  int2 s = (int2)(step,step);
  int2 g = tiV % s;
  int2 p = 2*s*(tiV/s)+g;
  int2 m = p + s;
  float2 T = (float2)(a[p.x],
                      a[p.y]);
  float2 V = (float2)(a[m.x],
                      a[m.y]);
  float2 X = T + V;
  float2 Y = T - V;
  a[p.x]   = X.x;
  a[p.y]   = X.y;
  a[m.x]   = Y.x;
  a[m.y]   = Y.y;
}
```

```
__kernel void
FastWalshTransform(float* a,
                   int    step)
{
  if (step<=0 || step%2!=0) {
    // Omitted code:
    // Execute original kernel.
    return;
  }
  int tid = get_global_id();
  if (tid % 2 != 0) return;
  int g = tid % step;
  int p = 2*step*(tid/step)+g;
  int m = p + step;
  float2 T = *((float2*)(a+p));
  float2 V = *((float2*)(a+m));
  *((float2*)(a+p)) = T + V;
  *((float2*)(a+m)) = T - V;
}
```

At this point, let us give the precise definitions of mod_k and div_k:

$$x = k \cdot \mathrm{div}_k(x) + \mathrm{mod}_k(x), \quad \text{where} \quad |\,\mathrm{mod}_k(x)| < |k|. \tag{5.3}$$

It is well-known that this definition does not uniquely specify $\mathrm{div}_k(x)$ and $\mathrm{mod}_k(x)$. SMT-LIB Version 2 resolves this issue by making the convention that $\mathrm{mod}_k(x) \geq 0$.[4] As long as both k and x are non-negative, common programming languages agree with this convention. However, when negative numbers are involved, OpenCL follows the C99 standard, which in contrast to SMT-LIB requires that $\mathrm{sign}(\mathrm{mod}_k(x)) = \mathrm{sign}(x)$. In our setting, we observe that the arguments of mod_k generally are positive expressions involving the `tid` such that both conventions happen to coincide.

Let us analyze a single memory access with respect to the following *consecutivity question*: "Do W consecutive work items access consecutive

[4] `smtlib.cs.uiowa.edu/theories/Ints.smt2`

memory addresses when doing this memory access or not?" Using the corresponding term $e(t, a)$, the following equation holds if and only if the work items t and $t + 1$ access consecutive memory locations for input a:

$$e(t, a) + 1 = e(t + 1, a).$$

The following conjunction generalizes this equation to W consecutive work items $t, \dots, t + W - 1$:

$$\bigwedge_{i=0}^{W-2} e(t + i, a) + 1 = e(t + i + 1, a).$$

Recall from the previous section that these groups of W work items naturally start at 0 so that only conjunctions are relevant where t is divisible by W. The following Presburger formula formally adds this constraint:

$$\varphi(W, a) = \forall t\Big(t \geq 0 \land t \equiv_W 0 \longrightarrow \bigwedge_{i=0}^{W-2} e(t + i, a) + 1 = e(t + i + 1, a)\Big).$$

For given $W \in \mathbb{N}$ and $\alpha, \beta \in \mathbb{Z}$ with $\alpha \leq \beta - 1$, the answer to our consecutivity question for W and $a \in \{\alpha, \dots, \beta - 1\}$ is given by the set

$$A_{W,\alpha,\beta} = \{\, a \in \mathbb{Z} \mid \mathbb{Z} \models \varphi(W, a) \land \alpha \leq a < \beta \,\}.$$

We essentially compute $A_{W,\alpha,\beta}$ by at most $(W - 1)(\beta - \alpha - 1)$ many applications of an SMT solver to the $W - 1$ disjuncts of $\neg\varphi(W, a)$ for $a \in \{\alpha, \dots, \beta - 1\}$, where

$$\neg\varphi(W, a) = \bigvee_{i=0}^{W-2} \exists t\big(t \geq 0 \land t \equiv_W 0 \land e(t + i, a) + 1 \neq e(t + i + 1, a)\big).$$

Notice that, when obtaining "sat" for some $i \in \{0, \dots, W - 2\}$, the remaining problems of the disjunction need not be computed.

Our answer $A_{W,\alpha,\beta}$ consists of those a for which the SMT solver yields "unsat." Note that besides "sat" or "unsat," the solver can also yield "unknown," which we treat like "sat." This underapproximation does not affect the correctness of our approach. We only miss optimization opportunities when generating code later on. The same holds for possible timeouts when imposing reasonable time limits on the single solver calls. Later in Section 5.8.3, we are going to discuss how compact representations for $A_{W,\alpha,\beta}$ can be obtained.

Table 5.4 `FastWalshTransform`: Running times of Z3 applied to $\neg\varphi(W, a)$ for $e(t, a)$ as in (5.2). In all three cases, $\alpha = 1$ and $\beta = 2^{16}$ so that $a \in \{1, \dots, 2^{16} - 1\}$ with a time limit of one minute per call.

W	Sat	Unsat	Unknown	Timeouts	CPU Time
4	16,383	49,152	0	0	4 min
8	8,191	57,344	0	0	5 min
16	4,095	61,128	0	312	334 min

Table 5.5 `BitonicSort`: Running times of Z3 applied to $\neg\varphi(W, a)$ for $e(t, a)$ as in (5.4). In all three cases, $\alpha = 0$ and $\beta = 63$ so that $a \in \{0, \dots, 62\}$ with a time limit of one minute per call.

W	Sat	Unsat	Unknown	Timeouts	CPU Time
4	61	2	0	0	0.7 s
8	60	3	0	0	1.5 s
16	59	4	0	0	3.7 s

Table 5.4 shows running times and results for the application of Z3 version 4.3.1 [De Moura & Bjørner 2008] to the consecutivity question for our `FastWalshTransform` kernel.[5] Alternatives to Z3 include CVC4 [Barrett et al. 2011] and MathSAT5 [Cimatti et al. 2013]. These SMT solvers, however, do not directly support div_k and mod_k, which makes them less interesting for our application here. The numbers shown already include a novel technique called *modulo elimination* [Karrenberg et al. 2013] that improved running times of Z3.

For another kernel taken from the AMD APP SDK, `BitonicSort`, the interesting address computation expression is

$$e(t, a) = 2^{a+1} \cdot \mathrm{div}_{2^a}(t) + \mathrm{mod}_{2^a}(t) + 2^a. \tag{5.4}$$

The input parameter a occurs exclusively as an exponent. This restricts the reasonable range of values to consider to $\{0, \dots, 62\}$ on a 64 bit architecture. Table 5.5 shows the relevant running times.

5.8.3 From SMT Solving Results to Code

Recall from Section 5.8.2 that the answer obtained there to our consecutivity question "Do W consecutive work items access consecutive memory addresses

[5] All our SMT computations have been performed on a 2.4 GHz Intel Xeon E5-4640 running Debian Linux 64 bit.

Table 5.6 Output from the SMT solving step for all our problem sets. We have $X \subseteq \{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \,\}$ with $|X| = 312$, i.e., timeouts occur only for input a with $a \equiv_{16} 0$.

Problem Set	W	α	β	$A_{W,\alpha,\beta}$
`FastWalshTransform`	4	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_4 0 \,\}$
`FastWalshTransform`	8	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_8 0 \,\}$
`FastWalshTransform`	16	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \,\} \setminus X$
`BitonicSort`	4	0	63	$\{0, \dots, 62\} \setminus \{0, 1\}$
`BitonicSort`	8	0	63	$\{0, \dots, 62\} \setminus \{0, 1, 2\}$
`BitonicSort`	16	0	63	$\{0, \dots, 62\} \setminus \{0, 1, 2, 3\}$

when doing this memory access or not?" is the set $A_{W,\alpha,\beta}$ of inputs $a \in \{\alpha, \dots, \beta - 1\}$ for which the answer is affirmative. The respective sets $A_{W,\alpha,\beta}$ for all our problem sets are collected in Table 5.6.

Our goal is now to produce during code generation a case distinction such that for input contained in $A_{W,\alpha,\beta}$, more efficient code including vector memory operations will be executed. The right-hand side of Listing 5.2 shows the automatically generated code for the `FastWalshTransform` kernel for $W = 2$ without imposing bounds α and β. For readability reasons, we use OpenCL notation instead of the LLVM IR, which we actually use at that stage of compilation. The corresponding condition

```
step <= 0 || step % 2 != 0
```
(5.5)

in the first `if` statement describes the complement of the set $A_{W,\alpha,\beta}$ obtained from our SMT solving step.

Due to our independent runs of the SMT solver for all possible choices of a, the sets $A_{W,\alpha,\beta}$ are obtained explicitly as lists of elements. From these, we have to generate implicit descriptions like (5.5). One approach for this is to represent the characteristic functions of the sets $A_{W,\alpha,\beta}$ as bit strings and to use incremental finite automata minimization to obtain minimal regular expressions. These are finally transformed into quantifier-free Presburger conditions. Alternatively, one could apply automatic synthesis techniques as suggested by Gulwani et al. [2011]. At present, this step is not automated yet.

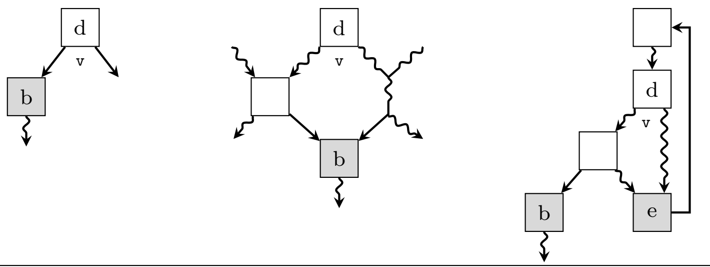


Figure 5.10: Illustration of the four possibilities for blocks to be `rewire` targets (Definition 15) of a `div_causing` block *d* (Definition 14).

5.9 Rewire Target Analysis

The goal of the Rewire Target Analysis is to determine program points that must not be skipped during SIMD execution. Thereby, we lay the foundation for the Partial CFG Linearization algorithm (Section 6.3).

Definitions 9 and 10 imply that, for different groups of instances, the set of `div_causing` blocks and their `rewire` targets may be different. This in turn means that the disjoint paths that have to be executed may also differ from group to group. This non-static property can only be fully exploited by dynamic variants. Such a variant can make it possible to skip blocks or entire paths at runtime (see Chapter 7). However, it is possible to statically derive a conservative overapproximation for the set of `div_causing` blocks and the corresponding `rewire` targets. The following definitions can be used by a partial CFG linearization algorithm:

Definition 14 (Static Divergence-Causing Block) *A block is a* `div_causing` *block if it ends with a* ***`varying`*** *branch.*

Definition 15 (Static Rewire Target) *A block b is a* `rewire` target *of a* `div_causing` *block d (and thus marked* $\texttt{rewire}_d$*) iff any of the following criteria is met:*

1. *b has an incoming edge from d and is no loop header, or*
2. *b is* $\texttt{blend}_d$*, or*
3. *b is a latch of a* `divergent` *loop which includes d, or*
4. *b is an exit block of a loop l with loop latch e. d is also part of l and has n successor blocks $s_1, \ldots, s_n$. There exist at least two* disjoint *paths $s_x \rightarrow^* b$ and $s_y \rightarrow^* e$, where neither path includes the back edge of l or a back edge of an outer loop.*

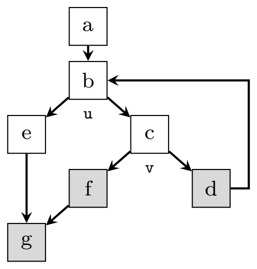


Figure 5.11: The blocks f and d of the `Mandelbrot` kernel are `rewire` targets of c because of criterion 1. f also fulfills criterion 4, d also fullfills criterion 3. g is $\texttt{blend}_c$ and thus $\texttt{rewire}_c$ due to criterion 2. e is not a `rewire` target because there are no disjoint paths from c to e and the latch d.

Figure 5.10 illustrates these conditions, Figures 5.11, 5.12, and 5.13 show a variety of examples (`rewire` target blocks are shaded).

5.9.1 Running Example

Consider again the `Mandelbrot` kernel shown in Figure 5.11. The blocks f and d are `rewire` targets of c due to criterion 1. They are direct successors of c, which has a `varying` branch and thus is a `div_causing` block. This means that none of these blocks must be skipped after c has been executed at least once. Block g is a $\texttt{blend}_c$ block, which means it is a `rewire` target of c due to criterion 2: if c was executed, g has to be executed as well. Since d is the latch of a `divergent` loop, it is $\texttt{rewire}_c$ due to criterion 3 (although this has no effect in this case). Criterion 4 is fulfilled for block f: There are disjoint paths from c to the latch d and to the exit f. This means that whenever all instances have left the loop, f has to be executed, since some instances may have left the loop over that exit in earlier iterations. This is especially important if the `uniform` exit $b \to e$ is taken. The criterion implies that execution must not continue with g but with f instead.

In the left CFG of Figure 5.12, only block b is `div_causing`. Blocks d and e are $\texttt{rewire}_b$ because they are direct successors of b. Blocks g and i are $\texttt{rewire}_b$ because they are join points of disjoint paths starting at b (they are $\texttt{blend}_b$).

5.9.2 Loop Criteria

The third and fourth conditions are required because loops behave differently in terms of control flow divergence.

Criterion 3 models behavior of disjoint paths inside `divergent` loops. The latch e of a loop l by definition is executed in every iteration by all instances that are still active in l. Because of that, e is a `rewire` target of every `div_causing` block d inside l whose postdominator is outside of l.

Criterion 4 is necessary because even loops of which all exit conditions are `uniform` can exhibit divergent behavior: If there are `div_causing` blocks inside the loop, this means that even if a `uniform` exit is taken, there may still be instances in different parts of the loop. This leads to situations where instances are "waiting" in different exit blocks for the loop to finish iterating. These exit blocks are both end and start points of disjoint paths of `div_causing` blocks inside the loop. When the loop finally stops iterating, the paths that start at these `rewire` exit blocks *all* have to be executed one after the other.

Figure 5.13 shows some more involved examples including loops. In the first graph, criterion 4 is exemplified in the inner loop: Although the exit condition in block k is `uniform`, the exit block n is a `rewire` target of i, indicating that some instances may leave the loop at a different exit or in different iterations. This is because some instances that diverge at the `varying` branch in i can take the exit to n while others may reach the loop latch (over the disjoint path $i \to l \to o$) and continue iterating. If, for example, these instances that remain in the loop then take the exit $h \to j$, execution has to continue in block n instead of jumping to m or q. Otherwise, that block would be skipped, and the instances that left over the exit $k \to n$ could produce wrong results.

In the second graph, block e is a `div_causing` block, i is one of its `rewire` targets, and the edge $d \to i$ leaves both loops. If the exit edge is taken, the inner loop has no more active instances because they must all have arrived at that exit. However, there may still be instances active in the outer loop that diverged earlier. Thus, after block i, execution has to continue with a block in the outer loop, not with one outside. Notice that i is only a `rewire` target in terms of the outer loop.

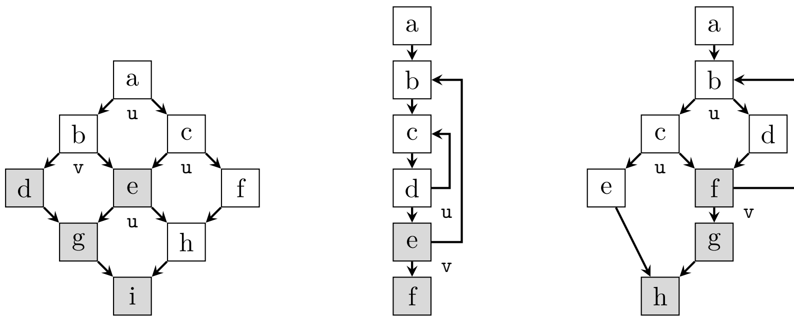


Figure 5.12: Example CFGs showing our analysis results. Uniformity of exit conditions is shown with a lowercase letter below the block, `rewire` target blocks are shaded. Our analysis determines that significant parts of these CFGs are `optional` (no `rewire` targets) and therefore do not have to be linearized (see Figure 6.7).

5.9.3 Formal Definition

The `rewire` property of a program point is described by the simple lattice $\mathbb{O}$ (for *optionality*):

$$\begin{tikzcd}\texttt{rewire} \arrow[d, dash] \\ \texttt{optional}\end{tikzcd}$$

For a program point b, the update function of the `rewire` property o

$$o' = [\![W]\!]^\sharp(o), \qquad [\![\cdot]\!]^\sharp : \mathbb{O} \to \mathbb{O}$$

is defined as follows, which reflects Definition 15:

$$o' = \begin{cases} \texttt{rewire}_d & \text{if } b \neq h_L \wedge \exists d \in \texttt{preds(b)}.\text{d is branch} \wedge D_{Abs}(d) \sqsupseteq \texttt{ca} \\ \texttt{rewire}_d & \text{if } b \in \texttt{blend}_d \\ \texttt{rewire}_d & \text{if } b = \ell_L \wedge d \in L \\ \texttt{rewire}_d & \text{if } b \in E_L \wedge \texttt{disjointPaths}(d, b, \ell_L) \\ \texttt{optional} & \text{otherwise,} \end{cases}$$

where `preds(b)` is the set of predecessor program points of b, E_L is the set of program points behind the exits of a loop L, h_L is the program point at the loop's header, and ℓ_L is the program point at the loop's latch. Furthermore, the mapping D_{Abs} is provided by the Vectorization Analysis. It is used to determine whether a predecessor is a `varying` branch. The information whether two paths are disjoint (non-cyclic and cyclic) is given by a function `disjointPaths` similar to the one in Section 5.6.5. Notice that in this case, however, the *iterated dominance frontier* cannot be used to answer the question whether the paths are disjoint. This is because the paths do not have the same program point as their target.

Since the `rewire` property is a property of a basic block entry point, we often simply refer to *rewire blocks* instead of program points. Loop exit blocks that are `rewire` targets because of criterion 4 will be referred to as *rewire loop exit blocks* or just *rewire exits.*

5.9.4 Application in Partial CFG Linearization

The Partial CFG Linearization phase transforms the CFG in such a way that all disjoint paths that are executed by some instances are always executed (Section 6.3). These paths are described by means of `div_causing` blocks and `rewire` targets. Implicitly, this means that `optional` blocks—blocks that are *no* `rewire` targets—do not always have to be executed: Edges that target such blocks can be retained, and the CFG still exhibits some of the original structure.

A conservative, complete linearization of the CFG thus can be forced by assuming that every conditional branch is `varying`: Each block with multiple outgoing edges then is a `div_causing` block, which in turn makes each of their successor blocks and each block with multiple incoming edges a `rewire` target. CFG linearization then has no choice but to linearize the entire CFG.

Note that this does not only affect linearization, but also the precision of the Vectorization Analysis: Since all branches are considered `varying`, every ϕ-function also has to be considered `varying`, which may result in less efficient code.

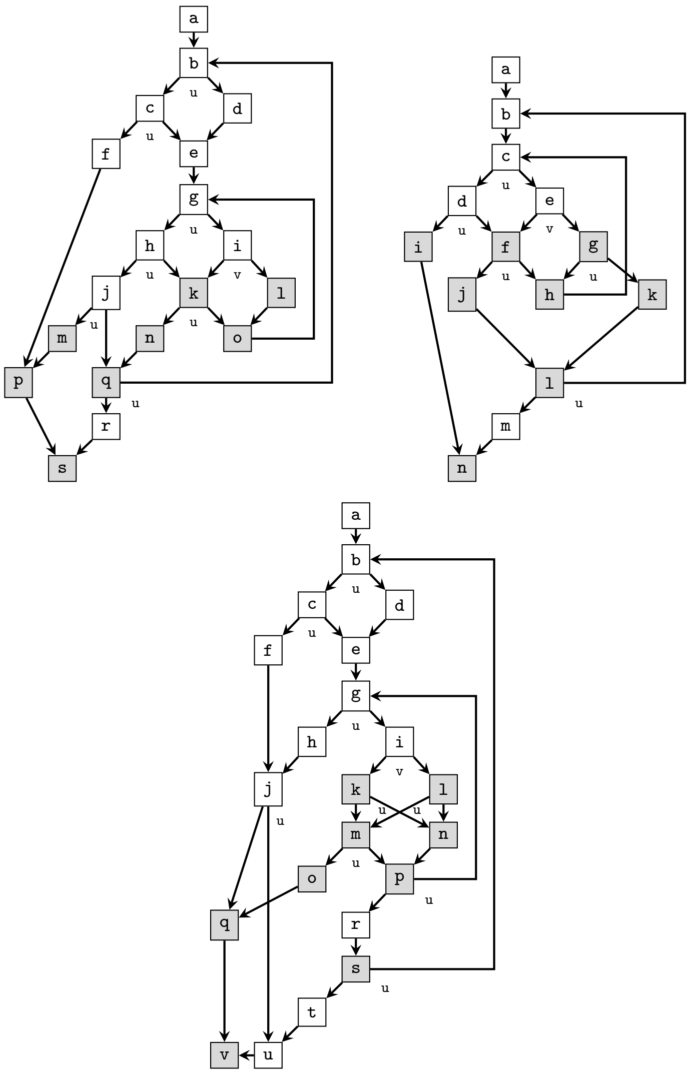


Figure 5.13: Complex examples with nested loops, loops with multiple exits, and exits that leave multiple loops at once. In the second graph, i is `rewire`$_e$ because criterion 4 is fulfilled for the outer loop (disjoint paths $e \to g \to h \to c \to d \to i$ and $e \to f \to j \to l$), albeit not for the inner. Partial linearizations for these examples are shown in Figure 6.14.

6 Whole-Function Vectorization

In this chapter, we present the main transformation phases of the Whole-Function Vectorization algorithm: Mask Generation, Select Generation, Partial CFG Linearization, and Instruction Vectorization.

6.1 Mask Generation

As already mentioned, control flow may diverge because a condition might be `true` for some scalar instances and `false` for others. Consequently, *all* code has to be executed. The explicit transfer of control is modeled by *masks* on control flow edges. A mask is a vector of truth values of size W. If the mask of a CFG edge $a \to b$ is set to `true` at position i, this means that the i-th instance of the code took the branch from a to b. Thus, the mask denotes which elements in a vector contain valid data on the corresponding control flow edge.

Algorithm 1: Pseudo-code for the main mask generation function.

```
Input: A CFG in SSA form.
Output: Mask information for every basic block and loop exit.
begin
    foreach $B \in$ return blocks do
        createMasks($B$);
    end
    foreach $L \in$ loops do
        createLoopExitMasks($L$);
    end
end
```

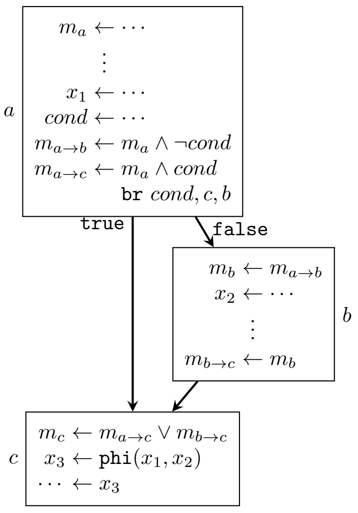
Algorithm 1 shows how masks are generated. The presented pseudo-code generates a graph where each node represents a mask. Code generation boils down to a straightforward depth-first traversal of the graph.

Function createMasks(Block B)

```
begin
    if B already has masks then
        return;
    end
    if B is loop header then
        createMasks(preheader);
    else
        foreach $P \in$ predecessors do
            createMasks($P$);
        end
    end
    createEntryMask($B$);
    createExitMasks($B$);
    if B is loop header then
        createMasks(latch);
        if loop is divergent then
            Mask latchMask $\leftarrow$ ExitMasks[latch$\rightarrow$header];
            EntryMasks[B].blocks.push(latch);
            EntryMasks[B].values.push(latchMask);
        end
    end
end
```

The edge masks implicitly define entry masks on blocks (Function createEntryMask): The entry mask of a block that is no loop header is either `true` for `by_all` blocks or the disjunction of the masks of all incoming edges. The mask of a loop header is a ϕ-function with incoming values from the loop's preheader and latch for `divergent` loops. Otherwise, the mask is always the one coming from the preheader. This is because as long as the loop iterates, all instances that were active upon entry of the loop remain active.

The masks of the control flow edges that leave a block are given by the block entry mask and a potential conditional (Function createExitMasks). Note that this does not apply to edges to `rewire` loop exit blocks, which are discussed in the next paragraph. If a block exits with an unconditional



a
$m_a \leftarrow \cdots$
$x_1 \leftarrow \cdots$
$cond \leftarrow \cdots$
$m_{a\to b} \leftarrow m_a \wedge \neg cond$
$m_{a\to c} \leftarrow m_a \wedge cond$
br $cond, c, b$
true
false
b
$m_b \leftarrow m_{a\to b}$
$x_2 \leftarrow \cdots$
$m_{b\to c} \leftarrow m_b$
c
$m_c \leftarrow m_{a\to c} \vee m_{b\to c}$
$x_3 \leftarrow$ phi(x_1, x_2)
$\cdots \leftarrow x_3$

Figure 6.1: Edge and block entry masks. m_a, m_b, and m_c are the entry masks of the corresponding blocks a, b, and c. $m_{a\to b}$, $m_{a\to c}$, and $m_{b\to c}$ are the block exit masks connected to the edges $a \to b$, $a \to c$, and $b \to c$.

branch, the mask of its single exit-edge is equal to the entry mask. If the block ends with a `varying` conditional branch, the exit mask of the "`true` edge" of the block is the conjunction of its entry mask and the branch condition. The exit mask of the "`false` edge" is the conjunction of the entry mask and the negated branch condition. For the "`true` edge" of a `uniform`, conditional branch condition, a select returns the entry mask of the block if the condition is met, otherwise it returns `false`. For the corresponding "`false` edge," an inverse select is used. This scheme extends naturally to blocks with more than two outgoing edges, e.g. due to a `switch` statement: The comparison of each `case` value to the `switch` value is the condition of that edge. Figure 6.1 shows an example with three basic blocks a, b, and c with corresponding block entry masks $(m_a, \dots)$ and edge masks $(m_{a\to b}, \dots)$.

The analyses presented in Chapter 5 enable various optimizations here. First, if our analysis found out that a block is always executed by *all* instances (`by_all`), the mask is set to `true`. Second, at the end of regions with a single entry and exit block, the mask can be reset to the one of the entry block. However, there is a trade-off involved: The mask has to be kept alive for the entire region, which can result in inferior performance to recomputing

Function createEntryMask(Block B)

```
begin
    if B is entry block then
        if function has mask argument then
            EntryMasks[B] ← Mask(VALUE, mask argument);
        else
            EntryMasks[B] ← Mask(VALUE, true);
        end
        return;
    end
    if B is by_all then
        EntryMasks[B] ← Mask(VALUE, true);
        return;
    end
    if has unique predecessor P then
        EntryMasks[B] ← ExitMasks[P→B];
        return;
    end
    if B is header of loop with preheader P then
        Mask loopEntryMask ← ExitMasks[P→B];
        if loop is divergent then
            EntryMasks[B] ← Mask(PHI);
            EntryMasks[B].blocks.push(P);
            EntryMasks[B].values.push(loopEntryMask);
        else
            EntryMasks[B] ← ExitMasks[P→B];
        end
        return;
    end
    if B is blend then
        Mask entryMask ← Mask(OR);
        foreach P ∈ predecessors do
            entryMask.push(ExitMasks[P→B]);
        end
        EntryMasks[B] ← entryMask;
    else
        Mask entryMask ← Mask(PHI);
        foreach P ∈ predecessors do
            Mask predMask ← ExitMasks[P→B];
            entryMask.blocks.push(P);
            entryMask.values.push(predMask);
        end
        EntryMasks[B] ← entryMask;
    end
end
```

Function createExitMasks(Block B)

```
begin
    if no successors then
        return;
    end
    if has unique successor S then
        ExitMasks[B→S] ← EntryMasks[B];
        return;
    end
    foreach S ∈ successors do
        // C is the condition of edge B→S,
        // e.g. Mask(NEG, C) for false edge of cond. branch.
        if exit condition C is uniform then
            ExitMasks[B→S] ← Mask(SELECT);
            ExitMasks[B→S].cond ← C;
            ExitMasks[B→S].trueVal ← EntryMasks[B];
            ExitMasks[B→S].falseVal ← Mask(VALUE, false);
        else
            ExitMasks[B→S] ← Mask(AND);
            ExitMasks[B→S].push(EntryMasks[B]);
            ExitMasks[B→S].push(Mask(VALUE, C));
        end
    end
end
```

the mask with a disjunction. Third, `optional` blocks always use the mask of their only predecessor or a phi with the incoming masks if the block has multiple incoming edges. This is because all instances that were active in the executed predecessor will also be active in the `optional` block (and none from a different direction, or the block would have been marked `rewire`). Otherwise, disjunctions would be generated, introducing some performance overhead compared to the original, scalar function. Also, the incoming mask of a block with a `uniform` branch and only `optional` successor blocks is used for both outgoing edges. Without the *Rewire Target Analysis*, the mask would have to be updated with the comparison result first. This implies that on paths with only `optional` blocks, all edges have the same mask as the first block, without requiring mask update operations. In the rightmost CFG of Figure 5.12, blocks c and d can both use the entry mask of block b instead of performing conjunction-operations with the (negated) branch condition in b, and block f can use the same mask instead of the disjunction of both incoming masks.

6.1.1 Loop Masks

Each `divergent` loop has to maintain a mask that is `true` for all instances that are still active in the loop. The loop can only be exited when this mask is `false` for *all* instances. While the loop is still iterating, results of inactive instances must not be altered. Therefore, a special ϕ-function—the *loop mask phi*—is generated in the loop header (m_b in Figure 6.2). Its first incoming value is the mask of the incoming edge from the preheader, the second value is the mask of the loop back edge.

Also, to ensure correct execution after the loop is finished, a `divergent` loop with multiple `rewire` exit blocks needs to persist the information which instance left the loop over which edge. This is achieved by introducing *loop exit masks* that are maintained by the following instructions (Function createLoopExitMasks): a mask update operation (m_{up}) and the *loop exit mask phi*, which is a ϕ-function in the loop header (m_{exit}). The update operation is the disjunction of the loop exit mask phi of the current loop and the accumulated mask of the next inner loop that is left via this exit, if there is one. Otherwise, the second operand is simply the exit condition of the exit edge. The loop exit mask phi has one incoming value from the preheader and one from the latch. The value coming from the latch is the result of the update operation. The value coming from the preheader is an empty mask (all elements set to `false`). Note that it is not necessary to persist the complete exit mask of an exit that leaves multiple loops in any other loop than the outermost one that is left. The inner loops only require information about which instances left in their current iteration.

After mask generation, each `rewire` loop exit mask thus has one update operation per loop that is left and one loop exit mask phi in the header of each loop that is left.

Note that, again, the analyses presented in Chapter 5 allow us to generate more efficient code: If an exit block is `optional`, we omit its loop exit mask because it is equal to the active mask. If the entire loop is not `divergent`, the loop mask and all loop exit masks can be omitted. This is because in such a loop, all instances that enter the loop will exit together through the same exit.

Function createLoopExitMasks(Loop L)

```
begin
    if L is divergent then
        foreach $E \in$ rewire exit blocks of L do
            ExitMaskPhis[E][L] ← Mask(PHI);
        end
    end
    foreach $N \in$ nested loops of L do
        createLoopExitMasks(N);
    end
    if L is not divergent then
        return;
    end
    Block P ← preheader of L;
    foreach $X \to E \in$ rewire exit edges of L do
        Mask exitMaskPhi ← ExitMaskPhis[E][L];
        exitMaskPhi.blocks.push(P);
        exitMaskPhi.values.push(Mask(VALUE, false));
        Mask maskUpdate ← Mask(OR, exitMaskPhi);
        if exit leaves multiple loops and L is not innermost loop left by
        this exit then
            N ← next nested loop of exit;
            maskUpdate.push(ExitMaskUpdates[E][N]);
        else
            maskUpdate.push(ExitMasks[X→E]);
        end
        ExitMaskUpdates[E][L] ← maskUpdate;
        if L is top level loop of exit then
            ExitMasks[X→E] ← maskUpdate;
        end
        exitMaskPhi.blocks.push(latch);
        exitMaskPhi.values.push(maskUpdate);
    end
    createCombinedLoopExitMask(L);
end
```

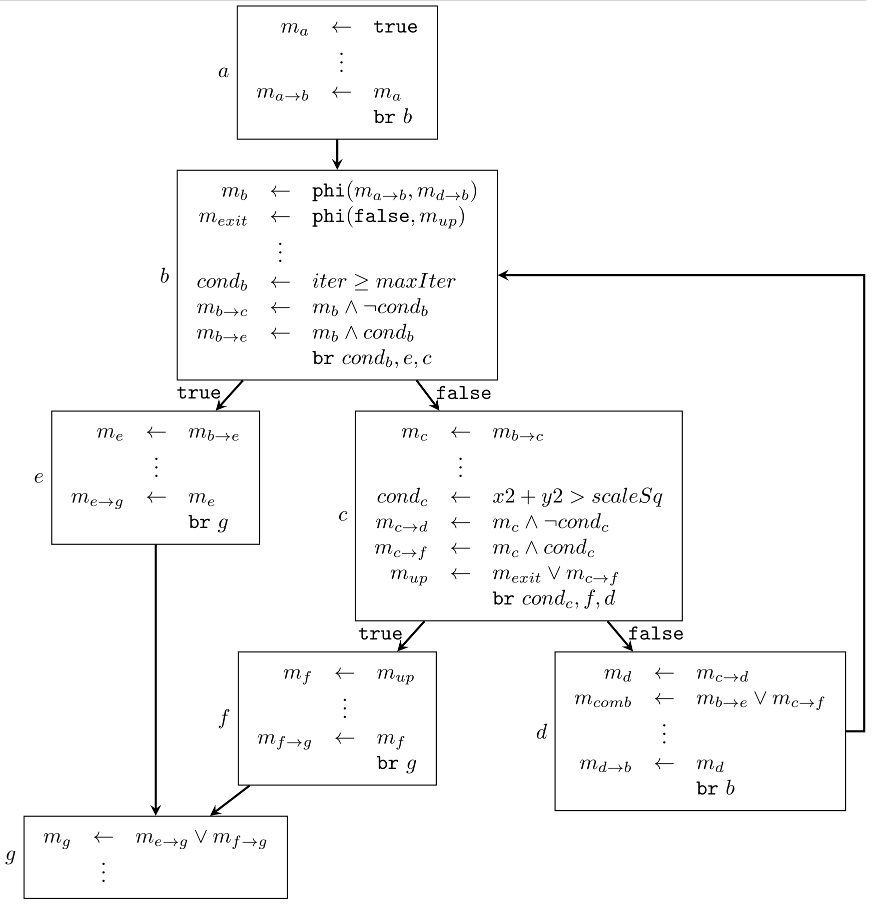
Function createCombinedLoopExitMask(Loop L)

```
begin
    Mask combinedMask ← Mask(OR);
    foreach $E \in$ rewire exit blocks of $L$ do
        combinedMask.push(ExitMaskUpdates[E][L][1]);
    end
    CombinedExitMasks[L] ← combinedMask;
end
```

Combined Loop Exit Masks. Finally, to reduce the number of instructions required to persist loop results, a *combined loop exit mask* may be used during select generation (see Section 6.2). This mask combines all information about instances that left the loop in the *current* iteration. In case of a loop that contains more nested loops, the current iteration of the parent includes all iterations of all nested loops. Thus, the combined loop exit mask is a disjunction of all accumulated loop exit masks of exits from nested loops and the exit masks of exits from the current loop. Function createCombinedLoopExitMask shows this in pseudo code.

6.1.2 Running Example

Figure 6.2 shows the masks generated for the `Mandelbrot` kernel. The `divergent` loop has one `uniform` exit ($b \rightarrow e$) and one `varying` exit ($c \rightarrow f$). The `uniform` exit does not require a dedicated exit mask, but the `varying` one does. It is maintained by the ϕ-function m_{exit} in the loop header b and updated by the disjunction m_{up} in c. Since m_{exit} is initialized with `false`, the disjunction accumulates those instances that have left the loop in each iteration, given by $m_{c\rightarrow f}$. The mask in block f is exactly this accumulated exit mask. The mask in e is simply the active mask if the exit is taken. This is because the exit condition is `uniform` and the block is an `optional` exit: *if* the exit is taken, it is taken by all instances that are still active. In block g, the masks from both sides are merged by a disjunction. In this case, since the block is `by_all`, it is equal to `true`. The combined loop exit mask m_{comb} has no uses before Select Generation (Section 6.2). It consists of a disjunction of the exit masks of the loop exits in the current iteration ($m_{b\rightarrow e}$ and $m_{c\rightarrow f}$).



a
$m_a \leftarrow$ true
$m_{a\to b} \leftarrow m_a$
br b
b
$m_b \leftarrow$ phi$(m_{a\to b}, m_{d\to b})$
$m_{exit} \leftarrow$ phi(false$, m_{up})$
$cond_b \leftarrow iter \geq maxIter$
$m_{b\to c} \leftarrow m_b \wedge \neg cond_b$
$m_{b\to e} \leftarrow m_b \wedge cond_b$
br $cond_b, e, c$
true
false
e
$m_e \leftarrow m_{b\to e}$
$m_{e\to g} \leftarrow m_e$
br g
c
$m_c \leftarrow m_{b\to c}$
$cond_c \leftarrow x2 + y2 > scaleSq$
$m_{c\to d} \leftarrow m_c \wedge \neg cond_c$
$m_{c\to f} \leftarrow m_c \wedge cond_c$
$m_{up} \leftarrow m_{exit} \vee m_{c\to f}$
br $cond_c, f, d$
true
false
f
$m_f \leftarrow m_{up}$
$m_{f\to g} \leftarrow m_f$
br g
d
$m_d \leftarrow m_{c\to d}$
$m_{comb} \leftarrow m_{b\to e} \vee m_{c\to f}$
$m_{d\to b} \leftarrow m_d$
br b
g
$m_g \leftarrow m_{e\to g} \vee m_{f\to g}$

Figure 6.2: Mask generation for the `Mandelbrot` kernel. m_{exit} is the accumulated exit mask of edge $c \to f$. m_{up} is the update operation of that exit mask. m_{comb} is the combined exit mask.

6.1.3 Alternative for Exits Leaving Multiple Loops

Our approach of maintaining loop exit masks aims at reducing the number of instructions required to persist loop results. However, there is an alternative that results in better code in some situations:

It is possible to do only a single update in the innermost loop instead of an update in every nested loop. This requires to use the loop exit mask phi of the current loop as input value coming from the preheader for the next nested mask phi. Basically, this means that the mask of a `rewire` loop exit is persisted across all loops. However, since our Select Generation approach also requires the information which instance left in the current iteration of each loop, this would require an additional phi per nested loop that is left (one for the instances that left during the current iteration, one for the instances that left at any iteration). If Select Generation introduces one `blend` operation per result per exit instead of one per result, however, the current iteration's mask would not be required. So, the choice during Mask Generation is to either use more update operations or persist more values across loop iterations. On a higher level, though, there is a choice between different algorithmic approaches to how results in loops are persisted, and that may influence the decision. For the approach described in this thesis, since the combined exit mask is used, generating more update operations instead of persisting more values is the natural choice to keep register pressure low.

6.2 Select Generation

Linearization of control flow is only possible if results of inactive instances are discarded. This is achieved by inserting *blend operations* at control flow join points and loop latches. The corresponding code is shown in Algorithm 2.

Algorithm 2: Pseudo-code for the main select generation function.

```
Input: A CFG in SSA form.
Output: A modified CFG that includes blend operations at control
        flow join points and loop latches.
begin
    for $B \in$ blend blocks do
        foreach $P \in$ phis of block do
            generateSelectFromPhi($P$, $B$);
        end
    end
    foreach $L \in$ loops do
        generateLoopExitSelects($L$);
    end
end
```

First, the values that have to be blended in `blend` blocks are given by the ϕ-functions, which are replaced by `select` instructions. Function generateSelectFromPhi shows how ϕ-functions with n incoming values are transformed into series of $n-1$ connected select instructions:

```
; before:
%s  = phi [ %val0, %bb0 ], [ %val1, %bb1 ], [ %val2, %bb2 ]
... = %s

; after:
%s0 = select i1 %m0, %val0, %val1 ; %m0 = mask from %bb0
%s1 = select i1 %m1, %val2, %s0   ; %m1 = mask from %bb2
... = %s1
```

The analyses from Chapter 5 help us to reduce the overhead of the generated code: If a block with multiple incoming edges is not `blend`, its ϕ-functions are not transformed into selects. Blending in such a case is not necessary because only one of the incoming paths may have been executed. For example, in the leftmost CFG of Figure 5.12, the ϕ's in block h remain untouched.

Function generateSelectFromPhi(Phi P, Block B)

```
begin
    S ← P.values[0];
    for $i = 1 \rightarrow$ *P.values.size()* $- 1$ do
        Value V ← P.values[i];
        Block P ← P.blocks[i];
        Mask M ← ExitMasks[P→B];
        S ← Select(M, V, S) in B;
    end
    replace all uses of P with S;
    delete P;
end
```

6.2.1 Loop Blending

Additionally, each loop requires *result vectors* in order to conserve the *loop live values* of instances that leave the loop early. Functions generateLoopExitSelects and generateLoopExitInsts show how loop exit selects are created.

Recall that loop live values are the incoming values of LCSSA phis in loop exit blocks. For each loop live value, a result vector that is maintained by two instructions—a ϕ-function and an update operation—is introduced per nested loop that is left over the corresponding exit. This vector is only updated whenever an instance has left the loop in the *current* iteration of the loop. Note that the loop live value of an exit that leaves multiple loops is defined in an inner loop. For each outer loop that is left, this means that all iterations of all inner loops have to be taken into account, since they correspond to the current iteration of the outer loop.

The update of each result vector is performed in the latch of the loop with the combined loop exit mask. This allows to only insert one select instruction for each loop live value per loop, regardless of the number of loop exits. As discussed in Section 6.1.3, there is an alternative: Instead of blending once per value, values could be blended once per value per `rewire` loop exit. However, this is only beneficial if there are `rewire` loop exits "above" `optional` ones. We will exemplify and discuss this in more detail in Section 6.3.5.

The result update operation is a `select` which uses the combined loop exit mask as the condition, and the result phi as the `false` value. This way, the result vector is updated with a new value only if one or more instances

Function generateLoopExitSelects(Loop L)

```
begin
    if $L$ is divergent then
        foreach $V \in$ LiveValues[L] do
            LoopResultPhis[L][V] $\leftarrow$ Phi() in header;
        end
    end
    foreach $N \in$ nested loops of $L$ do
        generateLoopExitSelects($N$);
    end
    if $L$ is divergent then
        foreach $V \in$ LiveValues[L] do
            generateLoopExitInsts($V$, $L$);
        end
        foreach $V \in$ LiveValues[L] do
            updateOptionalLCSSAPhis (V, L);
        end
    end
end
```

left the loop in the *current* iteration. If the mask is `true` for an instance, the blended value is either the live value or the update operation of the next child loop that is also left over this exit, if any exists. This means that the live value is used if the current loop is the innermost loop that is left, or if all deeper nested loops are not `divergent`.

The result-ϕ-function is placed in the header and has two incoming values. The incoming value from the loop preheader is undefined for the outermost loop that is left over this exit. This is because there is no result until the loop has iterated at least once. For nested loops, the incoming value is the parent loop's result-ϕ-function. The incoming value from the loop latch is the result update operation.

The usage of result vectors enables us to vectorize all kinds of loops. This especially includes control flow with multiple nesting levels, multiple exits and edges exiting multiple loops. Figure 6.3 shows an example of mask and select operations in a loop.

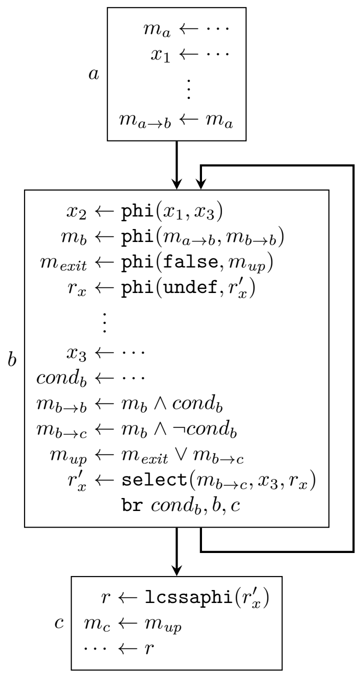
Function generateLoopExitInsts(LiveValue V, Loop L)

```
begin
    Mask M ← CombinedExitMask[L];
    Loop X ← parent loop of L;
    Phi P ← LoopResultPhis[L][V];
    Select S ← Select(M, V, P) in latch;
    LoopResultUpdates[L][V] ← S;
    P.blocks.push(preheader);
    if L is outermost divergent or !LoopResultUpdates[X][V] then
        P.values.push(undef);
    else
        P.values.push(LoopResultPhis[X][V]);
    end
    P.blocks.push(latch);
    P.values.push(S);
    if V is defined in deeper nested divergent loop N then
        S.trueVal ← LoopResultUpdates[N][V];
    end
    replace uses of V in parent loop of L with S;
    if L is outermost divergent then
        replace uses of V that are in no loop with S;
    end
end
```

If the Vectorization Analysis determines that a loop is not `divergent`, we do not need to blend any results because all instances will iterate equally often and use the same exit.

6.2.2 Blending of Optional Loop Exit Results

As described in Section 5.9, one of the goals of WFV is to retain parts of the CFG by linearizing only the necessary parts defined by `rewire` target blocks. This also affects loop exits: If all exit blocks were considered `rewire` targets, the code of all exits would always be executed. If a loop exit has an `optional` target, this means that even if the instances *can* diverge in the loop, if this exit is taken, it is taken for *all* remaining instances. This in



a
$m_a \leftarrow \cdots$
$x_1 \leftarrow \cdots$
$\vdots$
$m_{a\to b} \leftarrow m_a$
b
$x_2 \leftarrow$ phi(x_1, x_3)
$m_b \leftarrow$ phi$(m_{a\to b}, m_{b\to b})$
$m_{exit} \leftarrow$ phi(false$, m_{up})$
$r_x \leftarrow$ phi(undef$, r'_x)$
$\vdots$
$x_3 \leftarrow \cdots$
$cond_b \leftarrow \cdots$
$m_{b\to b} \leftarrow m_b \wedge cond_b$
$m_{b\to c} \leftarrow m_b \wedge \neg cond_b$
$m_{up} \leftarrow m_{exit} \vee m_{b\to c}$
$r'_x \leftarrow$ select$(m_{b\to c}, x_3, r_x)$
br $cond_b, b, c$
c
$r \leftarrow$ lcssaphi(r'_x)
$m_c \leftarrow m_{up}$
$\cdots \leftarrow r$

Figure 6.3: Mask and select generation for a loop. In general, each exit is assigned a mask update operation m_{up} and a ϕ-function m_{exit}. The exit mask is updated by setting elements of instances to `true` that leave the loop in the current iteration. The ϕ-function holds the current exit mask. Note that, after this pass, the mask of the edge $b \to c$ is m_{exit} instead of $m_{b\to c}$. The `select` r'_x in the latch and the ϕ-function r_x form the *result vector* of loop live value x. Each time an instance leaves the loop, the corresponding element of x is blended into the result vector.

turn allows to exit the loop immediately without an additional "all-`false`" mask test and without executing the remaining blocks of the loop. Also, there is no need to blend loop results: All results from previous iterations have been persisted, and in the current iteration no divergence happened. Thus, all current values are valid for the active instances, and can be passed directly to the corresponding LCSSA phis of this exit. The corresponding pseudo code is shown in Function updateOptionalLCSSAPhis.

Function updateOptionalLCSSAPhis(LiveValue V, Loop L)

```
begin
    foreach $(X \to E) \in$ optional exit edges of $L$ do
        foreach $phi \in E$ do
            if $phi.values[X]$ == LoopResultUpdates[L][V] then
                lcssaPhi $\leftarrow$ phi;
            end
        end
        if $lcssaPhi$ then
            replace all uses of lcssaPhi with V;
        end
    end
end
```

6.2.3 Running Example

Figure 6.4 again shows the `Mandelbrot` kernel, now with `blend` operations. Correct results are achieved with the use of the result vector r_{it} that stores the result of `iter` of each instance that leaves the loop over the edge $c \to f$ while others keep iterating. The vector is initialized with `undef` since there is no result before the first iteration. It is updated in each iteration in the loop latch d by the `select` operation r_{up} that uses the combined loop exit mask. If the mask is `false` for an instance, the old result is retained. If it is `true` for an instance, the corresponding element of the vector is set to the current value of `iter'`. Note that this means that the update operation sets the value of each instance at most once.

The LCSSA ϕ-functions in e and f forward the results that correspond to their exits: In f, this is the accumulated result vector. In e, it is the value of `iter` itself. This is because the exit block e is `optional`, so all values that leave over this exit leave in the same iteration. This requires no explicit result vector. Finally, the values are blended together to form the final result vector in block g. Note that this `select` can use either incoming mask m_e or m_f, only the blended values have to be supplied to the `true` or `false` operands of the operation accordingly.

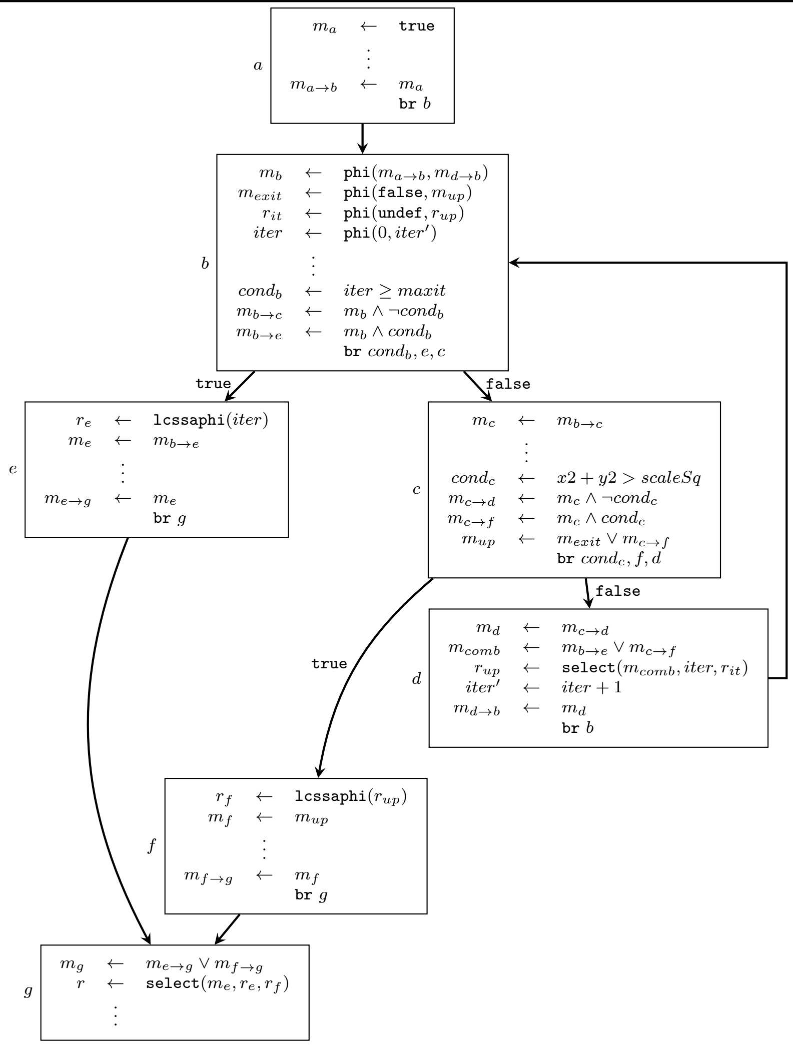

a
$m_a \leftarrow$ true
$m_{a\to b} \leftarrow m_a$
br b
b
$m_b \leftarrow$ phi$(m_{a\to b}, m_{d\to b})$
$m_{exit} \leftarrow$ phi(false, m_{up})
$r_{it} \leftarrow$ phi(undef, r_{up})
$iter \leftarrow$ phi$(0, iter')$
$cond_b \leftarrow iter \geq maxit$
$m_{b\to c} \leftarrow m_b \wedge \neg cond_b$
$m_{b\to e} \leftarrow m_b \wedge cond_b$
br $cond_b, e, c$
true
false
e
$r_e \leftarrow$ lcssaphi$(iter)$
$m_e \leftarrow m_{b\to e}$
$m_{e\to g} \leftarrow m_e$
br g
c
$m_c \leftarrow m_{b\to c}$
$cond_c \leftarrow x2 + y2 > scaleSq$
$m_{c\to d} \leftarrow m_c \wedge \neg cond_c$
$m_{c\to f} \leftarrow m_c \wedge cond_c$
$m_{up} \leftarrow m_{exit} \vee m_{c\to f}$
br $cond_c, f, d$
false
true
d
$m_d \leftarrow m_{c\to d}$
$m_{comb} \leftarrow m_{b\to e} \vee m_{c\to f}$
$r_{up} \leftarrow$ select$(m_{comb}, iter, r_{it})$
$iter' \leftarrow iter + 1$
$m_{d\to b} \leftarrow m_d$
br b
f
$r_f \leftarrow$ lcssaphi(r_{up})
$m_f \leftarrow m_{up}$
$m_{f\to g} \leftarrow m_f$
br g
g
$m_g \leftarrow m_{e\to g} \vee m_{f\to g}$
$r \leftarrow$ select(m_e, r_e, r_f)

Figure 6.4: Select generation for the `Mandelbrot` kernel. r_{it} is the accumulated result vector. r_{up} is the corresponding update operation. r is the final result, blended together from the two values incoming from both exits. Note that this is invalid code until linearized (see Figure 6.8).

```
      $m_a \leftarrow$
             ⋮
      $x_1 \leftarrow \cdots$
    $cond \leftarrow \cdots$
  $m_{a\to b} \leftarrow m_a \wedge \neg cond$
  $m_{a\to c} \leftarrow m_a \wedge cond$
----------------------------------------
      $m_b \leftarrow m_{a\to b}$
      $x_2 \leftarrow \cdots$
             ⋮
  $m_{b\to c} \leftarrow m_b$
----------------------------------------
      $m_c \leftarrow m_{a\to c} \vee m_{b\to c}$
      $x_3 \leftarrow \texttt{select}(m_{a\to c}, x_1, x_2)$
   $\cdots \leftarrow x_3$
```

Figure 6.5: The linearized control flow of Figure 6.1 with value blending.

6.3 Partial CFG Linearization

After all mask and select operations are inserted, all control flow except for loop back edges is encoded by data flow. To create a valid CFG for vector code, conditional branches that may result in divergence have to be removed. However, the transformation has to guarantee that, if a block that previously ended with such a `varying` branch is executed, all outgoing paths are executed as well. To this end, the basic blocks have to be put into a sequence that preserves the execution order of the original CFG G: If a block a is executed before b in every possible execution of G, then a has to be scheduled before b in the linearized CFG G'. This can be easily achieved by a topological sort of the CFG.

The naïve way of linearizing a CFG is to consider all blocks to be `rewire` blocks and linearize the entire graph: If the CFG splits up into separate paths, one path is chosen to be executed entirely before the other. This can be seen in Figure 6.5, which shows the linearized version of the CFG in Figure 6.1. There, both outgoing edges of a are *rewired* to b. If there was a block on the other path, the outgoing edge of b would be rewired to it. The result is a CFG that only has conditional branches remaining at loop exits and unconditional branches at loop entries. All other branches can be removed.

However, as discussed in Section 3.3, such a complete linearization is not desirable, since it may introduce a lot of overhead. Instead, we introduce an algorithm that retains all `uniform` branches. Suppose the conditional branch

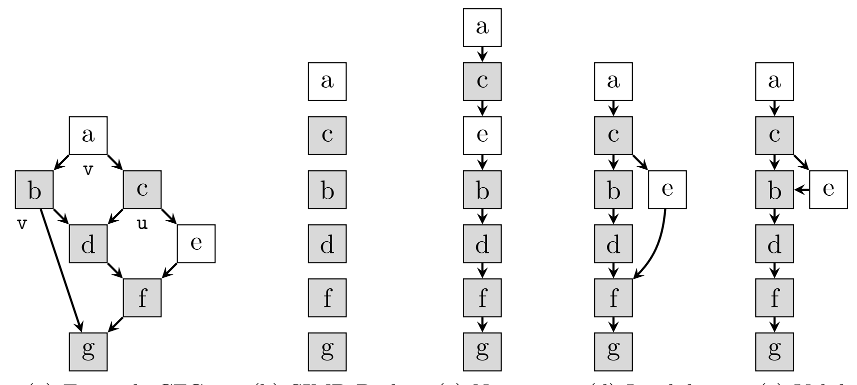


Figure 6.6: CFG linearization example. In the original CFG (a), *e* is `optional` because it cannot be reached over disjoint paths from a `varying` branch. The mandatory SIMD path (Section 6.3.3) that is used is shown in (b). The partial linearization (d) is invalid because it must not be possible to skip *b* and *d*. The graph (e) shows a valid, partial linearization, which improves performance over the naïve approach (c).

in Function 6.1 is `uniform`, i.e., all active instances will always execute the same path. This means that the structure does not have to be linearized as in Figure 6.5 but can be left intact. The *barrier elimination* optimization by Coutinho et al. [2011] covers such simple cases. However, simply retaining `uniform` branches produces invalid code if the CFG exhibits unstructured control flow. In fact, other parts of the CFG have to be modified to account for retained branches, as Figure 6.6 shows: If the `uniform` branch from block c to e is taken, SIMD execution after e must not continue at f, but at b (or d, if b was executed before c). This is because although all instances that arrived at c continued at e, there may have been instances that diverged at block a already. These may now be active in b (or d), so this block must not be skipped. If the `uniform` branch is retained without additional changes, the resulting CFG in Figure 6.6(d) is invalid: blocks b and d are skipped whenever e is executed. Rewiring the edge $e \rightarrow f$ to $e \rightarrow b$ as shown in Figure 6.6(e) yields a correct partial linearization.

The resulting function is more efficient than the fully linearized graph shown in Figure 6.6(c). This is because less code is executed in all cases where no instance is active in e, at no additional cost.

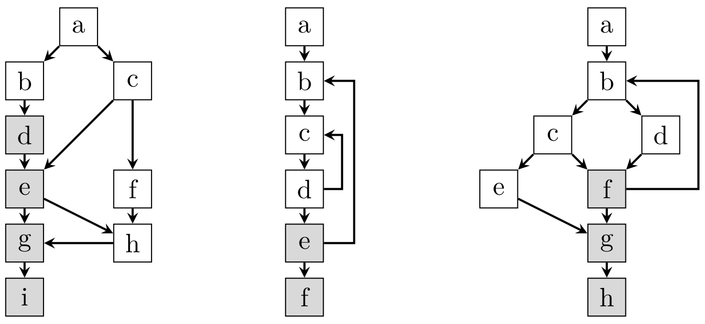


Figure 6.7: Partial linearizations of the CFGs shown in Figure 5.12.

Instead of retaining all edges that leave `uniform` branches, Partial CFG Linearization retains edges that target `optional` blocks. This is a subtle difference: Targets of `uniform` branches are usually `optional` blocks, unless the edge is critical, in which case the target block has an additional incoming edge which makes it a `blend` block.

Figure 6.7 shows linearizations of the examples of Figure 5.12. In the leftmost CFG, the blocks d, e, g, and i are `rewire` targets of the `div_causing` block b. Because there are no disjoint paths from b to h in the original CFG, h is no `rewire` target. In the middle CFG, the inner loop, although being nested in a loop with varying exit, does not require any mask updates or blending because all active instances always leave the loop together. The rightmost CFG shows a case where it is allowed to retain the `uniform` loop exit branch in c: there are only `uniform` branches inside the loop, so either all or no active instance will leave the loop at this exit. However, g must not be skipped because of instances that might have left the loop earlier. This makes it necessary to rewire the edge $e \to h$ to $e \to g$.

The Partial CFG Linearization shown in Algorithm 3 can handle arbitrary CFG structures. To this end, it exploits the information given by the Rewire Target Analysis (Section 5.9). The algorithm consists of five steps:

1. create *clusters* of `div_causing` blocks (Section 6.3.2).
2. determine the *mandatory SIMD path* of each cluster (Section 6.3.3).
3. determine how to modify the outgoing edges of each block (Section 6.3.4).
4. create new, partially linearized CFG (Section 6.3.5).
5. repair SSA form if any edges were modified (Section 6.3.6).

Algorithm 3: Pseudo-code for the main CFG linearization function.

```
Input: A CFG in SSA form with mask and selects.
Input: div_causing blocks, rewire targets.
Output: A partially linearized CFG in SSA form.
begin
    createClusters(div_causing blocks);
    determineRewireSchedule(clusters);
    determineNewEdges(function);
    regToMem(function);
    linearize(function);
    memToReg(function);
end
```

The main challenge with a partial linearization approach is that the resulting graph must not allow to skip a block that one of the grouped instances has to execute. These blocks form mandatory SIMD paths below each branch that may cause control flow to diverge. A partial linearization is achieved by scheduling these blocks into a linear sequence. Whenever a "side path" is executed due to a retained, `uniform` branch, execution has to continue with the next block in the sequence after the side path has been finished. In the example in Figure 6.6(e), the rewired edge forces exactly this behavior. Rewiring of edges ensures that all code that must be executed *is* always executed. In addition, when rewiring edges, care must be taken not to create paths where the execution of a block suddenly depends on conditional branches that were previously unrelated.

6.3.1 Running Example

The partially linearized `Mandelbrot` kernel is shown in Figure 6.8. Concrete operations are omitted to focus on the CFG linearization. The exit condition in d is an "all-`false`" comparison operation now, while the exit condition in b is left untouched. The `optional` exit $b \rightarrow e$ is retained: If the exit condition evaluates to `true`, *all* instances that are still active leave the loop at once. However, since there may have been others that left over the former exit $c \rightarrow f$, block f still has to be executed. This can only be avoided by using a dynamic variant (see Chapter 7). Full CFG linearization would have to move block e down below d, either before or after f. This means that in situations where the exit condition in e is never met, the partially linearized

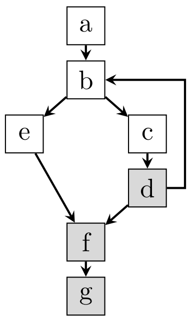


Figure 6.8: Partially linearized `Mandelbrot` kernel. The `optional` exit $b \to e$ is retained: If the iteration threshold is crossed, it is crossed by all active instances. However, the edge $e \to g$ is rewired to $e \to f$ to account for instances that may have left the loop in earlier iterations.

graph executes less code. This is the case for all instances that compute values that belong to the Mandelbrot set, since the first exit corresponds to reaching the iteration threshold.

Notice that the `rewire` exit now leaves from block d. This is because we blend all results in the loop latch instead of blending before every exit. If the original loop had multiple `rewire` exits, there would still be only one exit in the linearized version.

6.3.2 Clusters of Divergence-Causing Blocks

The first step of linearization is to determine which `div_causing` blocks form *clusters*.

Definition 16 (Cluster) *A* cluster *is the transitive closure of* `div_causing` *blocks that can be reached from each other within a larger single-entry, single-exit (SESE) region of the CFG.*

Clusters describe disjoint regions of a CFG: Each block b of a cluster C is either reachable from another block of C, or another block of C can be reached from b. Out of a set of clusters in a SESE region, only one can be entered by a group of instances, while no instance will be active in the others. The examples in Figures 6.9 and 6.10 show cases of disjoint clusters. Each cluster only consists of a single block that ends with a `varying` branch. Due to the `uniform` branch in a in both CFGs, only one of the clusters

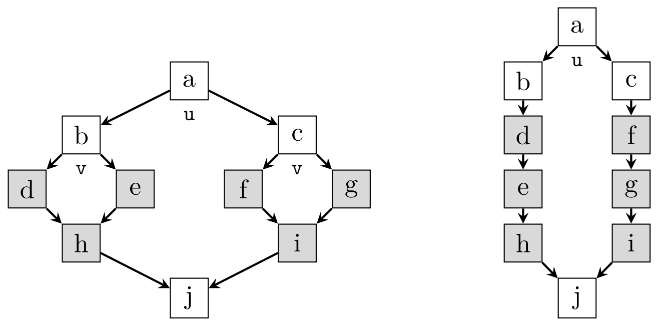


Figure 6.9: A CFG with two disjoint clusters and a partial linearization.

is executed by any group of instances. The restriction to SESE regions is necessary to keep clusters local: Consider a modification of the CFG in Figure 6.9 with an additional, diamond-shaped region starting with a `div_causing` block and ending at block *a*. Without the restriction to SESE regions, this CFG would have only a single cluster instead of three separate ones since all `div_causing` blocks could be reached from the first one. This could result in missed opportunities to retain control flow because a single mandatory SIMD path (see below) has to be created.

The pseudo code in Function createClusters shows how to determine which blocks belong to the same cluster.

6.3.3 Rewire Target Block Scheduling

Each cluster of blocks has a *mandatory SIMD path*:

Definition 17 (Mandatory SIMD Path) *A* mandatory SIMD path *of a cluster C is a list of the* `rewire` *targets of the blocks of C that is sorted topologically by control flow dependencies.*

A valid schedule of the `rewire` targets of a cluster can be determined by a simple top-down, depth-first traversal of the CFG as shown in Function determineRewireSchedule. During traversal, if a block is encountered that is a `rewire` target of the cluster, it is appended to an ordered list. This list forms the mandatory SIMD path, a *schedule* of `rewire` targets: it is a topological order of these blocks with respect to their control flow dependencies. Thereby, it defines in what order the disjoint paths of the cluster are executed.

Function createClusters(List[Block] DivCausingBlocks)

```
begin
    changed ← true;
    while changed do
        changed ← false;
        foreach BB0 ∈ DivCausingBlocks do
            Cluster C0 ← ClusterMap[BB0];
            foreach BB1 ∈ DivCausingBlocks do
                if BB0 = BB1 then
                    continue;
                end
                Cluster C1 ← ClusterMap[BB1];
                if C0 = C1 then
                    continue;
                end
                if BB1 is postdominator of C0 then
                    continue;
                end
                if BB1 is not reachable from BB0 then
                    continue;
                end
                C0 ← merge(C1, C0);
                foreach (BB,C) ∈ ClusterMap do
                    if C = C1 then
                        ClusterMap[BB] ← C0;
                    end
                end
                delete C1;
                changed ← true;
                break;
            end
            if changed then
                break;
            end
        end
    end
end
```

Function determineRewireSchedule(Cluster C)

```
begin
    Stack[Block] WorkList;
    WorkList.push(function entry block);
    while !WorkList.empty() do
        B ← WorkList.pop();
        if B is rewire target of C and no loop header then
            C.rewireList.push(B);
        end
        if B is postdominator of C then
            continue;
        end
        if B is header of loop L then
            foreach $X \to E \in$ exit edges of L do
                if L is not innermost loop left by this exit then
                    continue;
                end
                workList.push(E);
            end
        end
        foreach $S \in$ successors of B do
            if $B \to S$ is loop exit or loop back edge then
                continue;
            end
            if S has unseen non-latch predecessor then
                continue;
            end
            workList.push(S);
        end
    end
end
```

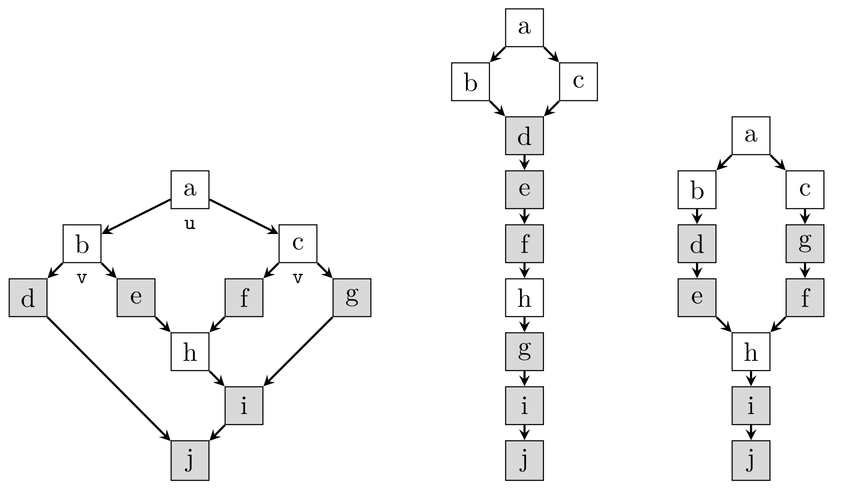


Figure 6.10: From left to right: Original CFG with two disjoint clusters $\{b\}$ and $\{c\}$ with overlapping mandatory SIMD paths, linearization with merged clusters, linearization with prioritized blocks d and g. The mandatory SIMD paths are $b \rightarrow d \rightarrow e \rightarrow i \rightarrow j$ and $c \rightarrow f \rightarrow g \rightarrow i$ for the middle CFG, the rightmost linearization requires g to be in front of f in the second path.

Note that the order of the depth-first traversal has a direct influence on the quality of the linearized CFG: The decision whether to schedule paths that start at a `varying` branch in one or another order influences how edges have to be rewired. Figure 6.10 is a good example for this: In the rightmost graph, the mandatory SIMD path of the cluster of block b is $d \rightarrow e \rightarrow i \rightarrow j$, and the path of the cluster of c is $g \rightarrow f \rightarrow i \rightarrow j$. This results in a linearization where d is never executed when the edge $a \rightarrow c$ is taken, and g is never executed when $a \rightarrow b$ is taken. However, achieving this particular partial linearization induces constraints upon the topological order of the mandatory SIMD paths: Block g has to be scheduled before f, and d before e, i.e., they have to be *prioritized*. Otherwise, either block would be moved below h, and thus be always executed. The resulting graph would still be more efficient than the fully linearized one, but not optimal. We leave the question how to best prioritize paths for future work and focus on an algorithm that produces valid partial linearizations for *any* topological order.

The decision how to schedule paths may also influence register pressure, which we discuss in more detail in Chapter 10. In the presented approach, the decision which path to execute first is made simply by the order in which successors are traversed. Employing some explicit heuristic instead is likely to improve the generated code. Such a heuristic can e.g. take into account the code size or register pressure of the different paths, or otherwise quantify the impact of the resulting CFG layout.

6.3.4 Computation of New Outgoing Edges

Next, the algorithm determines which edges of the CFG can be left untouched, which ones have to be rewired to other targets, which ones have to be removed, and where new edges are required. The corresponding pseudo-code is shown in Functions determineNewEdges and getNewOutgoingEdges.

All edges that target `optional` blocks are left untouched. Loop exit edges of `divergent` loops that do not target `optional` blocks are removed.[1] A new exit edge is created at the latch of each `divergent` loop: The target is the first `rewire` exit block for which the loop is the innermost that is left. If no such exit block exists, the target is the loop latch of the next outer loop.

Other edges that target `rewire` blocks are rewired as follows: Consider an edge of the original CFG that goes from a block b to a successor s. First, the `div_causing` blocks of b are queried and the corresponding clusters are collected. For each disjoint cluster, a new target of the edge is determined. The target is the first block r of the cluster's mandatory SIMD path for which the following holds: r is in the same loop as s, b cannot be reached from r, and b cannot be reached from any block that comes after r in the mandatory SIMD path. This ensures that the edge is rewired to the next `rewire` target that has not yet been targeted from within the current cluster.

In addition, all clusters that do *not* correspond to `div_causing` blocks of b but from which the current block can be reached also require a `rewire` target: the original successor s. While these clusters do not require the edge to be rewired to one of their `rewire` targets, they are still disjoint clusters and as such may require a different path to be executed. These paths did not change from the original graph, so the original successor is chosen.

This may result in edges that require multiple new targets, one for each disjoint cluster from which the block is reachable. An example for this can be seen in Figure 6.11, which we discuss in the next section.

[1] The restriction to `divergent` loops is important: There may be cases where a non-`divergent` inner loop is left via an exit to the latch of a `divergent` outer loop (which is not `optional`). This exit edge must not be removed.

Function determineNewEdges(Block B)

```
begin
    foreach $S \in$ successors of $B$ do
        if $S$ is optional then
            NewTargets[$B \rightarrow S$].push(S);
        else
            if $B \rightarrow S$ is exit of loop $L$ then
                if $L$ is not divergent then
                    NewTargets[$B \rightarrow S$].push(S);
                end
            else
                getNewOutgoingEdges($B$, $S$);
            end
        end
    end
    if $B$ is latch of divergent loop $L$ then
        X $\leftarrow$ first innermost, rewire exit of L;
        if !$X$ then
            X $\leftarrow$ latch of parent loop of L;
        end
        NewTargets[$B \rightarrow$ _].push(X);
    end
end
```

6.3.5 Linearization

Linearization is performed by creating the edges as determined by the Function determineNewEdges. The pseudo-code for the linearization is shown in Function linearize. Each edge $b \rightarrow s$ of the function is visited. If the edge has no new targets associated, it is removed. Otherwise, a new block x is created, and the edge is rewired to target that block ($b \rightarrow x$). In the new block, a *cluster-dependent branch* in the form of a `switch` statement is created:

Definition 18 (Cluster-Dependent Branch) *A branch is a* cluster-dependent *branch if its target depends only upon which disjoint cluster of the enclosing SESE region was executed on the path to the branch.*

Function getNewOutgoingEdges(Block B, Block S)

```
begin
    Set[Cluster] Clusters;
    foreach $X \in$ DivCausingBlocks[S] do
        Cluster C ← ClusterMap[X];
        Clusters.insert(C);
    end
    foreach $C \in$ Clusters do
        findNext ← true;
        foreach $R \in$ C.rewireList do
            if !findNext then
                if B is reachable from R then
                    findNext ← true;
                end
                continue;
            end
            if B is not reachable from R and R is in same loop as S
            then
                T ← R;
                findNext ← false;
            end
        end
        NewTargets[$B \to S$].push(T);
        NewTargetDivCausingBlocks[$B \to S$][T] ← C.entry;
    end
    foreach $C \in$ ClusterMap do
        if $C \notin$ Clusters and S is reachable from C.entry then
            NewTargets[$B \to S$].push(S);
            NewTargetDivCausingBlocks[$B \to S$][S] ← C.entry;
        end
    end
end
```

Function linearize(Function F)

```
begin
    foreach $B \in$ F do
        foreach $S \in$ successors do
            if NewTargets[$B \to S$].empty() then
                remove edge B→S;
                continue;
            end
            X ← new Block;
            rewire edge $B \to S$ to $B \to X$;
            foreach $T \in$ NewTargets[B → S] do
                P ← NewTargetDivCausingBlocks[$B \to S$][T];
                create edge $X \to T$ under condition P;
            end
        end
        if B is latch of divergent loop L then
            remove outgoing edges;
            X ← NewTargets[$B \to$ _];
            cond ← ExitMasks[$B \to$ header];
            create edges (cond ? $B \to$ header : $B \to X$);
            mask ← CombinedExitMask[L];
            ExitMasks[$B \to X$] ← mask;
        end
    end
end
```

The `switch` receives a case for each new target of the edge, i.e., one case per disjoint cluster from which the edge can be reached. To determine which case belongs to which cluster, an identifier value is defined in the entry block of each cluster. This value is tested by the `switch`, and the appropriate target is chosen to execute next. Finally, each latch of a `divergent` loop receives a new, "all-`false`" exit edge.

The two rightmost graphs in Figure 6.11 show two different possibilities for linearization with cluster-dependent branches.[2] If the edge $a \to b$ was

[2]Note that, in this particular example, it is possible to create a similar schedule by removing edges to `optional` blocks. For more complex examples it is not possible anymore to retain as much structure of the CFG without retaining these edges.

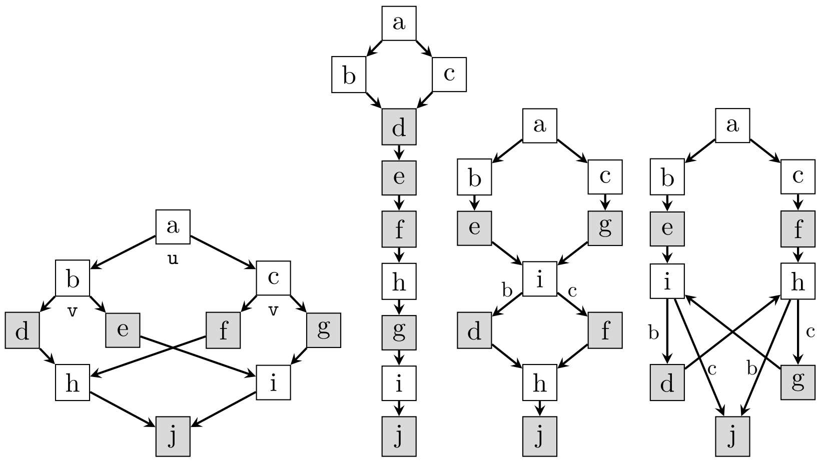


Figure 6.11: From left to right: Original CFG with two disjoint clusters whose mandatory SIMD paths overlap, linearization with merged clusters, partial linearization with cluster-dependent branch in i, partial linearization with cluster-dependent branches in i and h. Edge annotations denote that the edge is only taken if the corresponding block was executed before. The mandatory SIMD paths are $b \to e \to d \to j$ and $c \to g \to f \to j$.

taken, the conditional branch in i jumps to d. If the edge $a \to c$ was taken, the conditional branch in i jumps to f in the third graph, and to j in the last graph. Furthermore, the conditional branch in h in the rightmost graph jumps to g if coming from c, and to j if coming from b. Although this graph looks like it contains an irreducible loop with two headers, no block will be executed more than once due to the branch conditions.

Notice again that the chosen schedule of the `rewire` targets influences code quality: In the second graph from the right in Figure 6.11, the rewire schedule of block b is $e \to d \to j$, and the schedule of block c is $g \to f \to j$. This results in a linearization where only i has a cluster-dependent branch. In the rightmost graph, however, block f is scheduled before g in the cluster of c ($f \to g \to j$), while the schedule of the cluster of b remains as before. Because of the mutual dependencies of i and h, these schedules now force two cluster-dependent branches.

A CFG simplification phase is required to clean up the resulting graph: The algorithm may introduce unnecessary blocks, duplicate edges that target the same block, and edges that can never be executed. Blocks that were introduced on non-critical edges can always be removed. Also, the `switch` statements can be replaced by unconditional or conditional branches unless there were more than two disjoint clusters. The partially linearized graph on the right of Figure 6.12 is already cleaned up for space reasons.

6.3.5.1 Discussion

Figure 6.10 has two mandatory SIMD paths that overlap, one for each cluster. In such a case, there are multiple options for linearization: The naïve way is to merge the two clusters, define their common dominator as the entry, and create a single mandatory SIMD path instead of two. A possible result for this is the second graph of Figure 6.10. The third graph shows the result of our more involved, partial linearization that retains edges that target `optional` blocks. It is obvious that this approach yields more efficient code.

The next observation is that retaining edges to `optional` blocks is not possible with only rewiring of edges, as Figure 6.11 shows. In this example, both h and i are `optional` because they are no join points of *disjoint* paths from `varying` branches. This indicates that there are disjoint clusters whose mandatory SIMD paths somehow overlap. Indeed, all disjoint paths only merge again in block j. This imposes a challenge for partial linearization if edges to `optional` blocks should be retained: When executing block c, both f and g have to be executed. When executing block b, both d and e have to be executed. This means that no matter which blocks are chosen to be scheduled first, the edges $h \rightarrow j$ and $i \rightarrow j$ each have *two* different `rewire` targets where to continue execution.

This cannot be solved by clever scheduling or rewiring of edges. However, the fact that these different `rewire` targets belong to mandatory SIMD paths of different clusters allows to still retain disjoint paths. This is achieved with the help of *cluster-dependent branches*, which choose their target depending on the executed cluster. Effectively, this leads to disjoint paths that use the same blocks. Another option to implement this is to duplicate those blocks that are common to both paths.

6.3.5.2 Removing Loop Exit Edges

All `rewire` loop exit edges of a loop are replaced by a single exit in the loop latch. This is required because, as discussed in Section 6.1.3, we only blend loop live values once in the loop latch instead of before every exit. For short-running loops with a lot of code between different exits and high

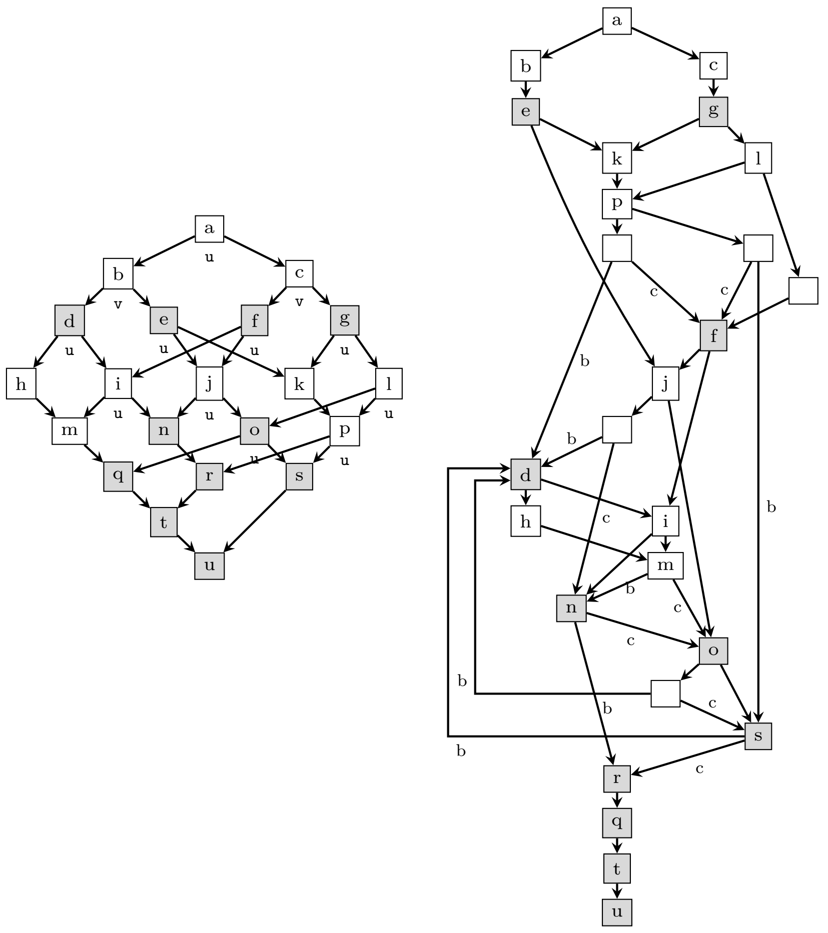


Figure 6.12: Left: Complex CFG that requires either code duplication or cluster-dependent branches when linearizing partially. Right: Partial linearization using cluster-dependent branches (edge annotations denote that the edge is only taken if the corresponding block was executed before). Blocks without letter were introduced for edges with multiple `rewire` targets. If critical edges are broken before vectorization, the introduction of additional blocks is not necessary. The cyclic regions are no loops, each path is executed at most once.

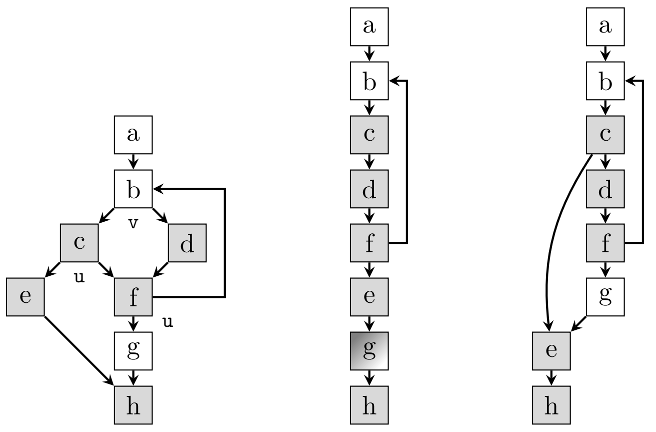


Figure 6.13: Left: Example of a loop with an `optional` exit that cannot be exploited with our approach (see Section 6.1.3). Middle: Blending in the loop latch forces complete linearization. Right: If `blend` operations are inserted before every exit instead, it is possible to retain the `uniform` exit in block *c*.

divergence of instances, it may be a better solution to retain more exits and introduce "all-`false`" tests at each one.

A second shortcoming of the approach described here is that we sacrifice optimization of one specific case: Consider the loop in Figure 6.13. It has an "upper" `rewire` exit and a "lower" `optional` one. If we would retain the `optional` exit and replace the `rewire` exit by an exit in the latch, it could happen that result vectors are not updated correctly. This is the case if some instances leave over the `rewire` exit, and all others leave in the same iteration over the `optional` exit. Since we only introduce `blend` operations in the latch, the necessary blending for the instances that left over the `rewire` exit would not have been executed when the `optional` exit is taken. This could be solved by introducing additional `blend` operations before each `rewire` exit from which an `optional` exit can be reached in the same iteration. This would introduce additional overhead for the blending, but make it possible to retain the `optional` exit. Otherwise, an `optional` exit has to be considered `rewire` if, in one iteration of a loop, it can be executed after some instances left the loop over another exit. The latter is the choice that was made for the approach presented here. Figure 6.14 shows additional examples for partial linearization in presence of complex loops.

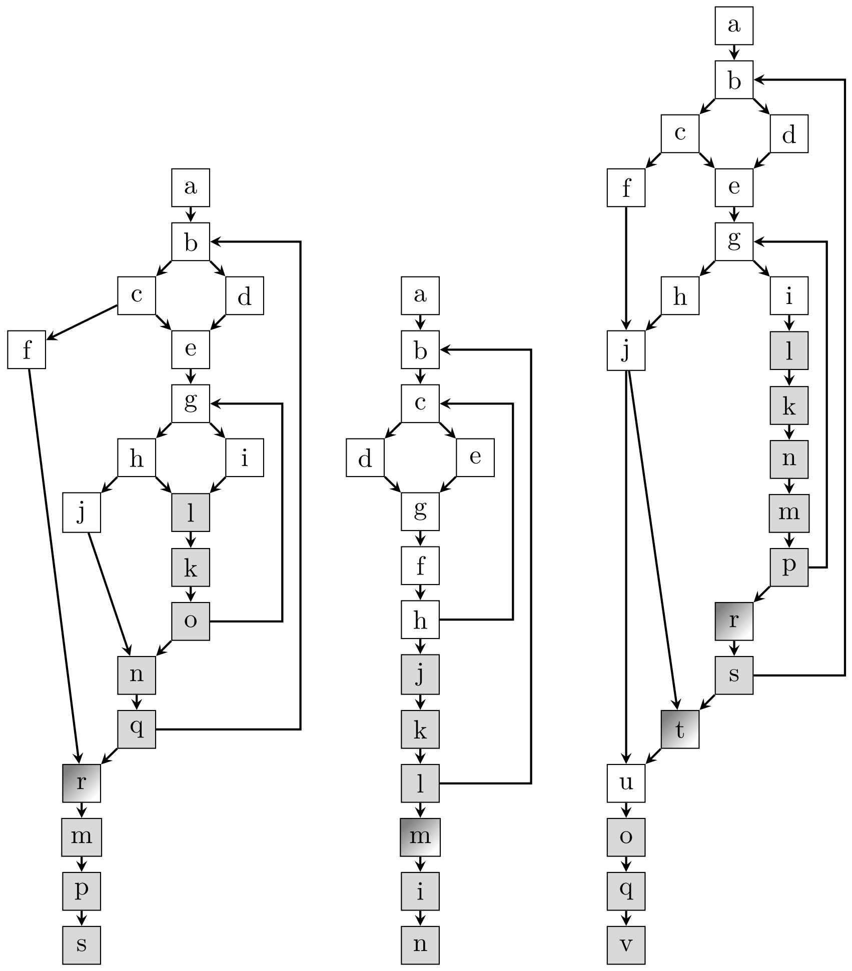

a b c d e f g h i j l k o n q r m p s
a b c d e g f h j k l m i n
a b c d e f g h i j l k n m p r s t u o q v

Figure 6.14: Partial linearizations for the examples from Figure 5.13. Partially shaded blocks are `optional` blocks that have to be treated like `rewire` blocks (see Section 6.1.3).

6.3.6 Repairing SSA Form

After edges were removed or rewired, SSA form may be broken and has to be repaired. This may involve significant amounts of rewrites of SSA values and introductions of ϕ-functions. Under these circumstances, an approach that first demotes all SSA values to memory before linearization, and then promotes memory back to registers again is less error prone and can reuse existing code for SSA construction.

There is only one obstacle to overcome: During demotion of a ϕ-function, it can happen that the `store` operations introduced for the incoming values are introduced on the same path because of rewired edges. This is because even if `optional` blocks were excluded from the mandatory SIMD path, their outgoing edges may have been rewired. This may result in paths overlapping that were disjoint before. A straightforward demotion algorithm thus does not work because the “later” `store` would overwrite the effect of the “earlier” one, and during register promotion, only one value would be kept. What has to be done is similar to the Select Generation phase—a `blend` operation has to be introduced at the position of the “later” `store` to merge the values. After this, a standard promotion algorithm can be used.

6.3.7 Branch Fusion

Branch fusion [Coutinho et al. 2011] (also *block unification if-conversion* [Rotem & Ben-Asher 2012]) is a technique that attempts to improve the code that is generated when a simple conditional structure is linearized. In many cases, the code of the `then` and `else` path will behave similarly, e.g. only loading a value from different addresses. The CFG linearization phase as presented simply schedules one path behind the other. However, in some cases it is possible to create more efficient code by merging the paths. This is the case if two paths that form a diamond-shape in the CFG contain a similar set of operations. Branch fusion pairs instructions with the same opcode from either path, and connects their operands with `blend` instructions. This way, only one of the two instructions is required. This improves performance if the removed instruction is more expensive than the `blend` operations, or if values can be reused without additional blending.

6.4 Instruction Vectorization

After linearization, the scalar source function is finally transformed into vector code. Vectorizing a single instruction in most cases is simply a one-to-one translation of the scalar instruction to its SIMD counterpart. This is true for most `varying` operations. Instructions marked `uniform` can remain scalar. In case of ϕ-functions and `selects`, they may return either scalar values if marked `uniform`, or vectors if marked `varying`. Instructions marked `sequential` or `guarded` remain scalar but are duplicated into W scalar instructions as detailed in Section 6.4.4.

6.4.1 Broadcasting of Uniform Values

The Vectorization Analysis allows us to leave `uniform` operations scalar. The benefit of using `uniform` computations is straightforward: Register pressure is taken from the vector units and scalar computations can be executed in parallel to the vector computations.

If a `varying` instruction uses a `uniform` value, it expects a vector. To produce a vector, the value is *broadcast* before the use. This means that a new vector value is created in which each element is the same as the `uniform` value (see Section 3.3).

The `Mandelbrot` application shows a use case where multiple parts of our algorithm play together. Consider the main loop from Listing 1.1:

```
uint iter=0;
for (; (x2+y2<=scaleSq) && (iter<maxIter); ++iter)
{
    y  = 2 * x * y + y0;
    x  = x2 - y2   + x0;
    x2 = x*x;
    y2 = y*y;
}
image[tid] = 255*iter/maxIter;
```

The main loop is `divergent`, since it has an exit condition that depends on the coordinates $x2$ and $y2$, which are different for each instance, and thus `varying`. This means that each instance may iterate the loop a different number of times. In consequence, the use of `iter` behind the loop has to be `varying`, since the returned iteration count of different instances may differ.

A naïve approach would simply consider all uses of `iter` as `varying` and vectorize them. Introducing result vectors for the loop live values as described in Section 6.2 allows the following: The result vector of `iter` is updated after every iteration of the loop. The update is performed by a broadcast of the scalar `iter`, followed by a `blend` operation. This allows to perform all computations that only depend on `uniform` values in scalar registers within the same loop iteration. Only uses outside the loop require the result vector.

In the `Mandelbrot` example, this means that the increment of `iter` and the comparison `iter < maxIter` can remain scalar and `maxIter` does not require a broadcast. This seemingly small difference can have a big impact, since the optimized operations are inside a frequently executed loop. One required vector register less or more in this critical part of the code can affect performance significantly.

6.4.2 Consecutive Value Optimization

Operations that produce a `consecutive` value can be left scalar as long as their operands do not require them to be vectorized as well. If the usage of a value expects it to be a vector, the value is broadcast and added to the vector $< 0, 1, \ldots, W - 1 >$:

```
; scalar code:
%idx = call i32 @get_global_id(i32 0) ; consecutive/aligned
%a   = add i32 %idx, 4                ; consecutive/aligned
%b   = add i32 %a, %v                 ; varying

; vector code:
%idx = call i32 @get_global_id(i32 0)
%a   = add i32 %idx, 4
%av  = broadcast i32 %a to <4 x i32>
%b   = add <4 x i32> %av, %v
```

Again, this allows to use the scalar unit a little more and occupy less vector registers.

6.4.3 Merging of Sequential Results

An instruction that is `sequential` or `guarded` may still yield a vector value as its result. This reflects the fact that even if the operation may require scalar, sequential execution, the result type allows to combine the scalar results to a vector. Such a *merge* or *pack* operation happens when a user of this value expects a vector. If merging was not possible, the Vectorization Analysis would have marked the operation `nonvectorizable` and the user would be `sequential`.

It is also worth mentioning that the result of a `sequential` or `guarded` operation can be used in different ways. For example, one use could be another `sequential` instruction, for which the scalar results are used directly. Another use could be `varying`, for which the results are merged first.

6.4.4 Duplication of Non-Vectorizable Operations

Operations with side effects such as `store` operations and function calls have to be duplicated W times. None of these scalar operations is allowed to be executed unless the corresponding mask element is `true`.

In general, all instructions that are marked as `sequential` or `guarded` by the Vectorization Analysis have to be executed W times sequentially in scalar mode. To this end, the corresponding scalar instruction is duplicated W times. Operands marked `uniform` are forwarded to each new instruction. Operands marked `varying` are extracted to produce a scalar operand for each instance. Operands marked `nonvectorizable` are scalar already, they are mapped to the scalar instruction that corresponds to their instance.

In case of function calls with pointer arguments we also have to generate code that writes back possible changes. Recall Listing 3.3 from Section 3.3. It demonstrated that this extracting and merging may involve significant amounts of memory operations that reduce the overall benefit of vectorization.

Additionally, for `guarded` instructions, we have to guard each scalar execution by an `if` construct that skips the instruction if the mask of that instance is `false`. Note that `guarded` marks are influenced by the `by_all` mark (Sections 5.6.6 and 5.6.7): If a block is `by_all` and thus proven to be always executed by *all* instances, none of its instructions will be `guarded`, only `sequential`. This way, we prevent generation of expensive guards since all instances will execute all instructions of the block anyway.

In case of a `guarded store`, there is another possibility to improve the generated code by employing a `load-blend-store` sequence. These vector operations are faster than guards with scalar `stores` as described above. However, this is not possible if the resulting code is executed in a concurrent environment, which is usually the case for data-parallel languages on machines with multiple processors. This is because a *race condition* is introduced: The `store` operation is executed for all instances, even though some may be inactive. Assume the memory location pointed to by one of these instances is concurrently written to. If this happens between the `load` and the `store`, the memory location is reset to the old content and the effect of the concurrent write is lost. Thus, this optimization can only be applied if the semantics of the source language explicitly allow it, or if there is no concurrency.

Note that it may make sense to still introduce guards for `sequential` operations if the operation is expensive. This has to be decided by a heuristic, weighing the cost of the operation against the overhead of the guards and the average number of inactive instances at that program point.

Another way to avoid the inefficiencies of `guarded` memory operations is to use hardware support for *gather* and *scatter* operations if available. Currently, only LRBni supports both constructs (which will be inherited by AVX-512), while AVX2 only has a gather instruction. Unfortunately, current implementations of these instruction sets only use microcodes instead of specialized circuitry, reducing the purpose of gather and scatter to simplified code generation without much performance improvement.

6.4.5 Pumped Vectorization

For certain applications it makes sense to vectorize with a factor W larger than the natural SIMD width S. This *pumped vectorization* results in values that require $V = W/S$ SIMD registers. Also, each vector operation has to be implemented by multiple operations. This technique improves locality at the expense of more registers being used. Generating machine code from a representation that uses larger vectors ("vector type legalization") has been studied by Asher & Rotem [2008]. Since this can be done by the back end, pumped vectorization is straightforward by using a larger value for W. Special attention has to be paid to calls to functions for which only a mapping for $W = S$ exists: The arguments of the call have to be extracted and merged into arguments of the smaller vector size S, similar to Listing 3.3 in Section 3.3.

6.5 Extension for Irreducible Control Flow

If the CFG of a function is irreducible (e.g. if there is a loop that has more than one header), the commonly used technique for many program analysis algorithms is to apply node splitting [Janssen & Corporaal 1997] to transform the CFG into reducible code before the analysis. However, this can result in an exponential blowup of the code size [Carter et al. 2003]. Although irreducible CFGs are rare [Stanier & Watson 2012], this can still be a problem for a specific application.

Our algorithm is able to deal with irreducible control flow without code duplication: During CFG linearization, one of the headers of an irreducible loop has to be chosen to be the *primary* header. This results in only the mask of the incoming edge of this header to be updated in every iteration. Entry masks from the other headers remain untouched: If a join point with one of these headers is executed during a later iteration, the incoming mask might falsely “reactivate” an instance that already left the loop.

In order to handle irreducible control flow directly, we have to ensure that these masks are joined with the loop mask *in the first iteration only*. This is achieved by performing the `blend` operations at those join points with a modified mask: In the first iteration, the new active mask is given by a disjunction of the current active mask with the incoming mask from the other header. In all subsequent iterations, it is given by a disjunction with `false`, which means the current loop mask is not modified.

7 Dynamic Code Variants

During or after vectorization of a function, additional optimizations that exploit *dynamic properties* of the function can be applied. Such an optimization duplicates an existing code path, improves the new code by assuming certain properties, and guards the new path by a runtime check that ensures that the assumed properties hold. We call such a guarded path a *dynamic variant*.

Obviously, introducing such a variant does not always make sense. Several factors influence the effects on performance: First, the dynamic check introduces overhead. Second, the improved code is more efficient than the original code that only used conservative static analyses. Third, the tested property may not always be valid, so the improved code is not always executed. Finally, parameters like the code size and instruction cache may also play a role, depending on the variant. Thus, each optimization presented in this chapter is subject to a heuristic that determines whether it is beneficial for a given code region to apply the transformation.

The properties that can be exploited all go back to the results of our analyses (Chapter 5). In general, each of the value properties such as `uniform`, `consecutive`, or `aligned` (see Table 5.1) can be tested at runtime. Since most of our analyses influence each other, the validation of a single value property can have a big impact on the quality of the generated code. For example, a value that conservatively has to be expected to be `varying` during static analysis could be proven to be `uniform` at runtime. If this value is the condition of a branch, less blocks are rewire targets and a smaller part of the CFG has to be linearized, which results in less executed code, and less mask and `blend` operations are required. The proven fact that control flow in this region does not diverge may in turn result in ϕ-functions to be `uniform`, which again may influence properties of other instructions.

In many cases, a heuristic will have a hard time to figure out the probability of a dynamic property to hold. So far, there have been no studies that attempted to classify under which circumstances a value is likely to be `uniform`, `consecutive`, or any other SIMD-execution-related property. An approach based on machine-learning techniques would offer a good starting point for such work, and also for heuristics.

Another possibility could be to provide compiler hints in the form of code annotations. This would allow the programmer to explicitly encode information, e.g. that a value is “often” `uniform`, hinting that a dynamic variant would be beneficial.

The following sections describe a variety of different dynamic variants ranging from enabling more efficient memory operations to complex transformations that modify the entire vectorization scheme of a region. For the most part, the presented variants have yet to be evaluated thoroughly.

7.1 Uniform Values and Control Flow

Definition 1 in Chapter 5 describes that the result of an instruction is `uniform` if it produces the same result for all instances of the executing SIMD group. The Vectorization Analysis can only prove a static subset of this property.

Consider the scalar code in Listing 7.1. The value x is loaded from the array at runtime. Without additional information, the Vectorization Analysis has to expect x to be different for each instance and thus `varying`. Because of this, the condition of the `if` statement is also `varying`, which forces control-flow to data-flow conversion, as shown in the vectorized function `kernelfn4`.

Function `kernelfn4v` shows the code with an introduced variant. The question whether x is `uniform` at runtime is answered by a comparison of all vector elements. If this holds, the condition of the `if` statement is also `uniform`, and the control flow can be retained. The code that is generated closely resembles the original code, up to the point where the scalar value of $x0$ is broadcast into a vector again.

Obvious choices for locations to introduce such a variant are `varying` values that have a big impact on the properties of other instructions and blocks. Because the test whether a value is `uniform` or not is fairly cheap, it is easy to find places where the variant is likely to improve performance. However, the problematic part for a heuristic is to estimate the probability of the value to be `varying`. For example, the input array in Listing 7.1 could never have 4 times the same value in consecutive elements. Then, the variant would result in a slowdown due to the added overhead of the dynamic test. Thus, it may often be a better idea to directly test conditions for uniformity instead of values. This is described in Section 7.6.

Listing 7.1 Example of a variant that exploits dynamic uniformity of value x. `kernelfn` is the original, scalar code. `kernelfn4` is a vectorized version. `kernelfn4v` is vectorized and employs a variant if x is `uniform`.

```
__kernel void
kernelfn(float* array,
         int    c)
{
  int tid = get_global_id(0);
  float x = array[tid];
  if (x > c)
    x += 1.f;
  else
    x += 2.f;
  array[tid] = x;
}
```

```
__kernel void
kernelfn4(float* array,
          int    c)
{
  int tid = get_global_id(0);
  if (tid % 4 != 0) return;
  float4 x4 =
    *((float4*)(array+tid));
  int4 c4 = (int4)(c);
  bool4 cond = x4 > c4;
  float4 x1 = x4 + 1.f;
  float4 x2 = x4 + 2.f;
  x4 = blend(cond, x1, x2);
  *((float4*)(array+tid)) = x4;
}
```

```
__kernel void
kernelfn4v(float* tArray,
           int    c)
{
  int tid = get_global_id();
  if (tid % 4 != 0) return;
  float4 x4 =
    *((float4*)(array+tid));
  if (x4[0]==x4[1]==x4[2]==x4[3])
  {
    float x0 = x4[0];
    if (x0 > c)
      x0 += 1.f;
    else
      x0 += 2.f;
    x4 = (float4)(x0);
  }
  else
  {
    int4 c4 = (int4)(c);
    bool4 cond = x4 > c4;
    float4 x1 = x4 + 1.f;
    float4 x2 = x4 + 2.f;
    x4 = blend(cond, x1, x2);
  }
  *((float4*)(array+tid)) = x4;
}
```

Listing 7.2 Left: Conservative WFV requires sequential execution of the `store`. Right: The dynamic variant executes a more efficient vector `store` if the memory indices are consecutive.

```
__kernel void
kernelfn4(int* array,
          int  c)
{
  int tid = get_global_id();
  if (tid % 4 != 0) return;
  int4 tid4 = (int4)(tid);
  tid4 += (int4)(0,1,2,3);
  int4 c4 = (int4)(c);
  int4 p = (tid4/c4)+(tid4%c4);
  array[p[0]] = p[0];
  array[p[1]] = p[1];
  array[p[2]] = p[2];
  array[p[3]] = p[3];
}
```

```
__kernel void
kernelfn4v(int* array,
           int  c)
{
  int tid = get_global_id();
  if (tid % 4 != 0) return;
  int4 tid4 = (int4)(tid);
  tid4 += (int4)(0,1,2,3);
  int4 c4 = (int4)(c);
  int4 p = (tid4/c4)+(tid4%c4);
  int4 px = p - <0,1,2,3>;
  if (px[0]==...==px[3]) {
    *((float4*)(array+p[0]))=p;
  } else {
    array[p[0]] = p[0];
    array[p[1]] = p[1];
    array[p[2]] = p[2];
    array[p[3]] = p[3];
  }
}
```

7.2 Consecutive Memory Access Operations

The Vectorization Analysis attempts to prove that the address of a memory operation is `uniform` or `consecutive`. While an `unknown` `load` or `store` requires W sequential operations, a `uniform` `load` or `store` only requires a single, scalar operation. A `consecutive` `load` or `store` can use a vector operation to access W elements at once.

Consider the example in Listing 7.2. The code on the left shows a conservatively vectorized kernel. It has to use sequential `store` operations because the analysis cannot prove `p` to be `consecutive`. The code on the right employs a dynamic variant: First, `p` is tested for consecutivity by substracting the vector $< 0, 1, 2, 3 >$ and comparing all elements for equality. If that holds, a single vector `store` is executed. Otherwise, the original code with W sequential operations is executed.

Note that in some cases, the SMT-based extension of the Vectorization Analysis (Section 5.8) is able to produce the same or even more precise variants based on the value of `c`. Consider the `FastWalshTransform` kernel in Listing 5.2 in Section 5.8 for an example.

It is also important to note that a vector `store` is only possible if the mask is entirely `true`, so we either need a block that is marked `by_all` or an additional dynamic test whether all instances are active (a `movemask` followed by a `cmp`).

7.3 Switching to Scalar Code

If the function in question has complex code that is frequently executed with only one active instance, it may be beneficial to switch back to sequential execution for that part. Ray tracing of scenes whose performance is dominated by incoherent, secondary rays is a good example for this: SIMD bundles of rays will often not hit the same surface anymore after the first bounce.

The dynamic variant first determines the index of the single active instance. On the optimized code path, all required values for that instance are extracted into scalar registers. The rest of that path consists of the scalar code of the original kernel before vectorization. Finally, at the end of the path, the results are packed back into the corresponding vectors. Listing 7.3 shows some exemplary OpenCL code.

Of course, this transformation is only beneficial if the vectorized code suffers from more overhead (e.g. due to control-flow to data-flow conversion) than the extraction/insertion that is required to execute the scalar code. However, additionally, the scalar code may hold the potential to use an orthogonal vectorization approach such as *Superword-Level Parallelism* (SLP) [Larsen & Amarasinghe 2000] (Section 4.2). Benthin et al. [2012] presented a manual approach of switching to sequential execution in the context of ray tracing. This work shows that there is potential for the dynamic application of this variant.

We were able to improve performance of an ambient occlusion ray tracing application similar to AOBench by a few percent when switching to sequential intersection code if only one ray of the SIMD group is active. However, this was only successful when rendering scenes with high divergence, e.g. due to large amounts of small objects. In other scenarios such as the simple scene rendered by AOBench, the overhead for the test outweighs the benefits of the sequential code because it is not executed often enough.

7.4 WFV-SLP

This variant generation technique can be seen as an extension of the previously described switching to sequential execution. The variant can be

Listing 7.3 The dynamic variant checks whether there is only one instance active. If this is the case, values are extracted, and scalar code is executed.

```
__kernel void
kernelfn4v(float* array,
           float  c)
{
  int tid = get_global_id();
  if (tid % 4 != 0) return;
  float4 x = *((float4*)(array+tid));
  float4 c4 = (float4)(c);
  float4 r = 0;
  do {
    bool4 m = x < c4;
    int n = m[0]+m[1]+m[2]+m[3];
    if (n==1) {
      int idx = m[0]?0:m[1]?1:m[2]?2:3;
      float xi = x[idx];
      // Execute original, scalar code.
      // ...
      r[idx] = expr(xi);
    } else {
      // Execute vectorized code.
      // ...
      r2 = expr(x);
      r = m ? r2 : r;
    }
  } while (any(m));
  *((float4*)(array+tid)) = r;
}
```

executed if only a subset of the SIMD group is active. The code transformation for the optimized code path merges independent, isomorphic vector operations into a single vector operation. This is similar to what *Superword-Level Parallelism* (SLP) [Larsen & Amarasinghe 2000] (Section 4.2) does for scalar code. Intuitively, this can be seen as switching the "vectorization direction:" Where normal WFV vectorizes horizontally (each operation works on combined inputs), SLP vectorizes vertically (different operations with different inputs are combined).

Since the original code is already vector code, combining values is more complicated than when transforming scalar code to SLP code. This is because the values of active instances first have to be extracted from their original vectors and then combined to a new one. This means that this variant possibly involves significant amounts of overhead due to the data reorganization.

Listing 7.4 The dynamic variant checks whether there are two out of eight instances active. If this is the case, values are extracted, and the variant code that employs a mixture of WFV and SLP is executed.

```
__kernel void
kernelfn8v(float* xA,
           float* yA,
           float* zA,
           float  c)
{
  int tid = get_global_id();
  if (tid % 8 != 0) return;
  float8 x = *((float8*)(xA+tid));
  float8 y = *((float8*)(yA+tid));
  float8 z = *((float8*)(zA+tid));
  float8 c8 = (float8)(c);
  float8 a,b,c = ...;
  float8 r = 0;
  do {
    bool8 m = x < c8;
    int n = m[0]+...+m[7];
    if (n==2) {
      int idx0=-1, idx1=-1;
      for (int i=0; i<8; ++i) {
          if (m[i]==0) continue;
          if (idx0==-1) idx0 = i;
          else idx1 = i;
      }
      float8 m = (float8)(x[idx0], x[idx1], y[idx0], y[idx1],
                          z[idx0], z[idx1], 0, 0);
      float8 n = (float8)(a[idx0], a[idx1], b[idx0], b[idx1],
                          c[idx0], c[idx1], 0, 0);
      float8 oS = m - n;
      float8 oA = m + n;
      float8 o  = oS * oA;
      r[idx0] = o[0] + o[2] + o[4];
      r[idx1] = o[1] + o[3] + o[5];
    } else {
      float8 oxS = x - a;
      float8 oyS = y - b;
      float8 ozS = z - c;
      float8 oxA = x + a;
      float8 oyA = y + b;
      float8 ozA = z + c;
      float8 ox = oxS * oxA;
      float8 oy = oyS * oyA;
      float8 oz = ozS * ozA;
      r2 = ox + oy + oz;
      r = m ? r2 : r;
    }
  } while (any(m));

  *((float8*)(array+tid)) = r;
}
```

Consider the example in Listing 7.4, which shows a WFV-SLP variant for two out of eight active instances ($W = 8$). Inside the `while` loop, there are two code paths. The `else` path is the original vector code that computes products of sums and differences. The results of 3 of these operations that are independent are added up and stored in variable r. This is very efficient if all or most instances are active. However, if only few are active, a lot of computations are wasted because their results are discarded by the `blend` (the ternary operator).

The variant shown in the `then` part improves on this. It is only executed if two instances are active. First, the indices $idx0$ and $idx1$ of the active instances are determined. Then, the input values of the three independent additions and subtractions are combined into two vectors (m, n) instead of six (x, y, z, a, b, c) as in the original vector code. Because these input values do not depend on each other, it is safe to use a single vector addition and subtraction instead of 3 operations of either type. Finally, scalar additions are required per active instance instead of the vector additions used in the original code. However, the variant code does not require an additional `blend` operation. This is because each result is inserted directly into the result vector at the appropriate index.

Again, we were able to improve performance by a few percent in an ambient occlusion ray tracer with this variant. And, again, as when switching to sequential execution, the variant is only successful if there is enough divergence to work with.

7.5 Instance Reorganization

This variant is only relevant for *pumped vectorization* (Section 6.4.5) with $W = V \cdot S$. Executing more instances in the same function than the number of available SIMD lanes offers an additional optimization opportunity used by *Instance Reorganization*: Instead of executing a single instruction or block V times sequentially, a larger code region is executed V times sequentially, but the instances are reorganized. The example in Figure 7.1 depicts the difference in the CFG layout. The reorganization aims at improving coherence of the executed code, either in terms of control flow behavior or in terms of memory access patterns. This trades back some of the gained locality of pumped execution against improved control-flow or memory-access behavior.

Consider the following example, where Instance Reorganization is used to improve control flow coherence. Figure 7.1 shows the CFGs that correspond to the different stages described next.

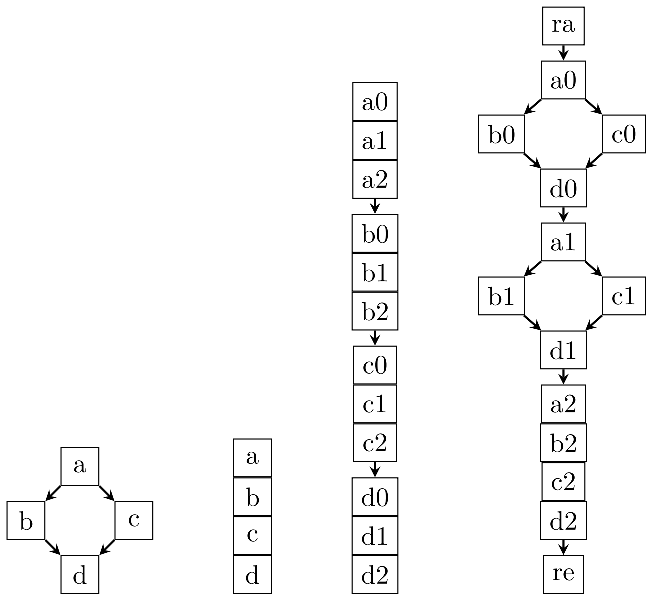


Figure 7.1: Visualization of the CFG transformation for Instance Reorganization. From left to right: scalar CFG, vectorized CFG, triple pumped, vectorized CFG ($W = V \cdot S, V = 3$), triple pumped, vectorized CFG with Instance Reorganization. Instance Reorganization allows to execute $V - 1$ times the vector code with the non-linearized control flow. Only a single execution has to account for diverging control flow. Blocks *ra* and *re* denote reorganization code.

```
float x = ...
if (x > 0.f)
  x += 1.f;
else
  x += 2.f;
... = x;
```

For the W instances that arrive at the `if` statement, $w_1 < W$ instances execute the `then` part, and $w_2 = W - w_1$ instances execute the `else` part. Assume $W = 24$ and $S = 8$, then V is 3. Normal WFV would yield the following code:

```
float24 x  = ...
bool24  m  = x > (float24)(0.f);
float24 xt = x + (float24)(1.f);
float24 xe = x + (float24)(2.f);
```

```
x = m ? xe : xt;
array[tid] = x;
```

Broken down to vectors that can be mapped to the machine, this becomes:

```
float8 x[3]; x[0]=...; x[1]=...; x[2]=...;
bool8  m[0] = x[0] > (float8)(0.f);
bool8  m[1] = x[1] > (float8)(0.f);
bool8  m[2] = x[2] > (float8)(0.f);
float8 xt[0] = x[0] + (float8)(1.f);
float8 xt[1] = x[1] + (float8)(1.f);
float8 xt[2] = x[2] + (float8)(1.f);
float8 xe[0] = x[0] + (float8)(2.f);
float8 xe[1] = x[1] + (float8)(2.f);
float8 xe[2] = x[2] + (float8)(2.f);
x[0] = m[0] ? xt[0] : xe[0];
x[1] = m[1] ? xt[1] : xe[1];
x[2] = m[2] ? xt[2] : xe[2];
...=x[0]; ...=x[1]; ...=x[2];
```

Instance Reorganization exploits the fact that there are only two possible decisions where to go for each instance. Using Instance Reorganization to improve control flow coherence, the code shown in Listing 7.5 is generated instead. The code makes use of the fact that if the W instances diverge into two sets, they can be reorganized such that there is at most one subset of S instances that do not agree on the direction. This allows, after reorganization, to execute $V-1$ times the code with `uniform` control flow, and only once the linearized code which accounts for diverged instances. This can be achieved either with code duplication or with a loop with a switch statement that determines which path to execute with which reorganized group.

On the flipside, as can be seen easily in Listing 7.5, reorganization may impose significant cost if many values are live at the point of reorganization. Note that the example was chosen for illustration only, and that code with Instance Reorganization may in many cases be less efficient than the code obtained by standard WFV or pumped WFV. Also, the reorganization is only valid for a single `varying` branch. Only the corresponding control flow of this branch can be retained in the $V-1$ coherent executions, each nested `varying` branch requires linearization again. Thus, this variant is more of a candidate for activation upon explicit request by the programmer instead of a fully automated technique guided by heuristics.

Listing 7.5 Example for Instance Reorganization for vectorization factor 24 and native SIMD width 8.

```
float8 x[3]; x[0]=...; x[1]=...; x[2]=...;
bool8  m[3];
bool8  m[0] = x[0] > (float8)(0.f);
bool8  m[1] = x[1] > (float8)(0.f);
bool8  m[2] = x[2] > (float8)(0.f);
// Reorganize.
float8 xR[3];
int idx = 0;
// Scan "true" instances.
for (int i=0; i<24; ++i) {
  if (m[i/8][i%8]) xR[idx/8][idx++] = x[i/8][i%8];
}
int mixedIdx = idx/8; // index of mixed group
bool8 mM; // mask of mixed group
for (int i=0; i<8; ++i) {
  mM[i] = i < idx%8;
}
// Scan "false" instances.
for (int i=0; i<24; ++i) {
  if (!m[i/8][i%8]) xR[idx/8][idx++] = x[i/8][i%8];
}
int uniIdx[2]; // Indices of non-mixed groups
for (int i=0, j=0; i<4; ++i) {
  if (i == mixedIdx) continue;
  uniIdx[j++] = i;
}
// Execute code with uniform control flow twice.
if (xR[uniIdx[0]][0] > 0.f)
  xR[uniIdx[0]] += (float8)(1.f);
else
  xR[uniIdx[0]] += (float8)(2.f);
if (xR[uniIdx[1]][0] > 0.f)
  xR[uniIdx[1]] += (float8)(1.f);
else
  xR[uniIdx[1]] += (float8)(2.f);
// Execute code with varying control flow once.
float8 xtM = xR[mixedIdx] + (float24)(1.f);
float8 xeM = xR[mixedIdx] + (float24)(2.f);
xR[mixedIdx] = mM ? xtM : xeM;
// Transform back to original order.
idx = 0;
for (int i=0; i<24; ++i) {
  if (m[i/8][i%8]) x[i/8][i%8] = xR[idx/8][idx++];
}
for (int i=0; i<24; ++i) {
  if (!m[i/8][i%8]) x[i/8][i%8] = xR[idx/8][idx++];
}
...=x[0]; ...=x[1]; ...=x[2];
```

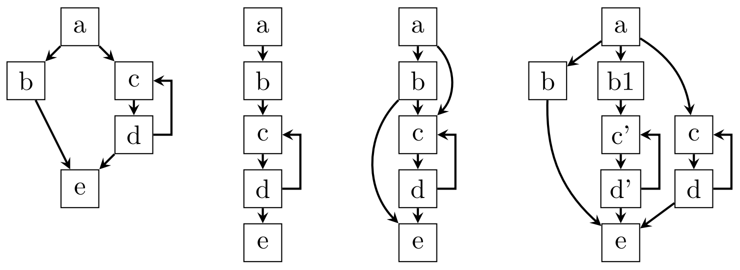


Figure 7.2: Visualization of different variants that allow to skip code regions where no instance is active. First: Original CFG. Second: Normal WFV. Third: Dynamic checks allow to skip either the `then` or the `else` path. Fourth: The same dynamic checks invoke execution of disjoint code paths.

7.6 Skipping All-Inactive Paths

For linearized regions, an optimization similar to *branch-on-superword-condition-codes (BOSCC)* [Shin 2007] can be applied. Such a technique reintroduces branches after linearization to skip a basic block or an entire control flow path if the mask of the corresponding entry edge is entirely `false` at runtime. This way, it trades some performance for the runtime check for a larger gain every time the block can be skipped.

The original BOSCC-algorithm operates independently after SLP vectorization. Identification of predicated regions relies on subsequent instructions being guarded by the same predicate. In the setting of WFV, predicates are stored on a per-block basis. In addition, it is easy to maintain information about structured control flow even across the CFG linearization phase. For this variant, only the start and end block of the path have to be stored. After linearization, an edge is introduced that goes directly from the start to the end. If there is a disjoint neighboring path, as in case of an `if` statement, an additional edge can be introduced.

Figure 7.2 shows an example of an `if` statement that allows to skip a code region that includes a loop. The left CFG shows the original layout. The second CFG shows the layout after WFV. The third CFG shows two edges that allow to skip either the `then` or the `else` path. Note that this variant, although it allows to skip code, still may suffer from increased register pressure: The code still has to account for the cases where both paths are executed. Thus, all values live-in at the start of the second path occupy registers while the first path is executed, and all values live-out at the end of the first path occupy registers while the second path is executed. The last CFG in Figure 7.2 shows a variant as described by Timnat et al. [2014] that also improves register pressure by duplicating code. Not depicted is the variation of Timnat et al. that allows to switch from one path to another, e.g. when a loop changes its behavior from non-`divergent` to `divergent` after the first instance dropped out.

A heuristic that controls where to apply the rightmost variant of Figure 7.2 could e.g. depend on the complexity or size of a path or the difference to a neighboring path.

Note that usage of this approach in some ways does not make it possible anymore to exploit branch fusion (Section 6.3.7). This is because branch fusion merges two disjoint paths into one optimized one. However, if code is duplicated such that there is a disjoint path for mixed masks, both techniques can be combined. For example, in the graph on the right of Figure 7.2, the middle path could be subject to merging both paths instead of executing them one after the other.[1]

[1]Note, however, that branch fusion is not applicable to this particular example since one path includes a loop.

8 Evaluation

In this chapter, we present a thorough evaluation of our Whole-Function Vectorization implementation in various scenarios: First, in *WFVOpenCL*, a prototypical OpenCL CPU driver that was implemented to showcase the benefits of WFV for a data-parallel language [Karrenberg & Hack 2012]. Second, in *AnySL*, a shading system aimed at highly efficient code without loss of portability and flexibility [Karrenberg et al. 2010]. Third, in *Noise*, a compiler that allows the user to specify which optimizations should be run on what parts of the code. In addition, we quantify the impact of the key techniques that exploit the results of the analyses described in Chapter 5.

8.1 Benchmark Setup and Reproducibility

In order to get realistic, unbiased results, every measurement was done "cold," i.e., there were no warm-up phases for specific applications as far as possible without restarting the entire machine. Every result presented in this chapter is the median execution time of at least 200 individual executions. In addition, we denote the minimum and maximum execution times that were measured as error bars in our graphs. All of this ensures that the reported results are as reproducible as possible by an arbitrary execution of the application in a normal environment [Touati et al. 2010]. When computing average speedups to compare different configurations or drivers, we use the geometric mean instead of the arithmetic mean to correctly weigh speedups and slowdowns [Fleming & Wallace 1986].

8.2 WFVOpenCL

We integrated WFV into the code generation pipeline of an OpenCL CPU driver [Karrenberg & Hack 2012]. We use the driver to compare different WFV configurations to each other. In addition, a comparison to the most important mainstream OpenCL CPU drivers is made: the proprietary implementations by Intel and AMD, and the open source implementation POCL.[1]

[1] `pocl.sourceforge.net`, version 0.8

Our driver is on par with the latest Intel driver and clearly outperforms the AMD and POCL implementations.

Although we focus on OpenCL, the presented techniques are also applicable to similar languages like CUDA. A data-parallel program is written in a scalar style. It is then executed in *n instances* or *work items* on a computing device.[2] To a certain extent, the order of execution among all instances of the program is unspecified to allow for parallel or sequential execution as well as a mixture of both. Every instance is identified with a *thread identifier* `tid`. Usually, the data-parallel program uses the `tid` to index data. Hence, every instance can process a different data item. In OpenCL, the `tid` is queried by calling the function `get_global_id`.

Instances are combined into *work groups* (also called *blocks*) that are important for synchronization. A kernel can use a `barrier` statement to enforce that no instance of a work group continues executing any code beyond the barrier before all instances have reached the barrier. There is no restriction for the behavior of instances in different work groups. GPUs have dedicated hardware support to implement barrier synchronization. On CPUs, barriers need to be implemented in software. Scalar implementations use the support of the runtime system and the operating system (e.g. Clover) which boils down to saving and restoring the complete state of the program for each instance. More sophisticated techniques use loop fission on the abstract syntax tree to decompose the kernel into separate pieces that are executed in a way such that all barriers are respected [Stratton et al. 2008]. However, this technique potentially introduces more synchronization points than needed. Section 8.2.3 describes an approach that generalizes the latter approach to work on control flow graphs (instead of abstract syntax trees) while not increasing the number of synchronization points.

In the following sections, we describe code-generation techniques to improve the efficiency of an OpenCL driver. Most importantly, we evaluate the impact of WFV. The compilation scheme of our driver looks like this:

1. Optimize scalar code as much as possible
2. Perform WFV (Section 8.2.1)
3. Implement barrier synchronization in software (Section 8.2.3)
4. Create loops for work group instances (Section 8.2.2)
5. Remove API callbacks such as `get_global_id` (Section 8.2.2)
6. Create wrapper for driver interface (omitted for brevity)

[2]Instances in the OpenCL/CUDA context are sometimes called *threads*, however this is not to be confused with an operating system thread.

The interface wrapper allows the driver to call the kernel with a static signature that receives only a pointer to a structure with all parameters. Pseudo-code for the responsible driver method `clEnqueueNDRangeKernel` and the modified kernel is shown in Listing 8.1 (before inlining and the callback optimizations described in Section 8.2.2).

8.2.1 WFV Integration

Enabling WFV in OpenCL can be summarized as follows: Vectorization is based upon changing the callback functions `get_global_id` to return not a single `tid` but a vector of W `tid`s whose instances are executed by the vectorized kernel.[3] The driver combines consecutive instances to exploit the fact that in most cases, consecutive instances will access consecutive memory. Thus, the Vectorization Analysis is initialized with all occurrences of `tid` to be `consecutive` and `aligned`. From there on, the kernel is vectorized as per the results of the Vectorization Analysis, converting control flow to data flow and vectorizing or duplicating values and operations as necessary.

8.2.2 Runtime Callbacks

OpenCL allows the user to organize instances in multiple dimensions (each instance is identified by an n-tuple of identifiers for n dimensions). Given a kernel and a global number of instances $N_0 \times \cdots \times N_n$ organized in an n-dimensional grid with work groups of size $G_0 \times \cdots \times G_n$, the driver is responsible for calling the kernel $N_0 \times \cdots \times N_n$ times and for making sure that calls to `get_global_id` etc. return the appropriate identifiers of the requested dimension. The most natural iteration scheme for this employs nested "outer" loops that iterate the number of work groups of each dimension ($N_0/G_0, \ldots,$ N_n/G_n) and nested "inner" loops that iterate the size of each work group ($G_0, \ldots, G_n$). Consider Listing 8.1 for some pseudo-code of the iteration scheme for two dimensions.

If the application uses more than one dimension for its input data, the driver has to choose one *SIMD dimension* for vectorization. This means that only queries for instance identifiers of this dimension will return a vector, queries for other dimensions return a single identifier. Because it is the natural choice for the kernels we have analyzed so far, our driver currently always uses the first dimension. However, it would be easy to implement a heuristic that chooses the best dimension, e.g. by comparing

[3]There is also `get_local_id`, which is handled analogously.

Listing 8.1 Pseudo-code implementation of `clEnqueueNDRangeKernel` and the kernel wrapper before inlining and optimization (2D case, $W = 4$). The outer loops iterate the number of work groups, which can easily be parallelized across multiple threads. The inner loops iterate all instances of a work group (step size 4 for the SIMD dimension 0).

```
cl_int
clEnqueueNDRangeKernel(Kernel kernelWrapper, TA arg_struct,
                       int* globalSizes, int* localSizes)
{
  int iter_0 = globalSizes[0] / localSizes[0];
  int iter_1 = globalSizes[1] / localSizes[1];
  for (int i=0; i<iter_0; ++i) {
    for (int j=0; j<iter_1; ++j) {
      int groupIDs[2] = { i, j };
      kernelWrapper(arg_struct, groupIDs,
                    globalSizes, localSizes);
    }
  }
}

void
kernelWrapper(TA arg_struct, int* groupIDs,
              int* globalSizes, int* localSizes)
{
  T0      param0 = arg_struct.p0;
  ...
  TN      paramN = arg_struct.pN;
  int     base0  = groupIDs[0] * localSizes[0];
  int     base1  = groupIDs[1] * localSizes[1];
  __m128i base0V = <base0, base0, base0, base0>;
  for (int i=0; i<localSizes[1]; ++i) {
    int lid1 = i;
    int tid1 = base1 + lid1;
    for (int j=0; j<localSizes[0]; j+=4) {
      __m128i lid0 = <j, j+1, j+2, j+3>;
      __m128i tid0 = base0V + lid0;
      simdKernel(param0, ..., paramN,
                 lid0, lid1, tid0, tid1,
                 groupIDs, globalSizes,
                 localSizes);
    }
  }
}
```

the number of memory operations that can be vectorized in either case, leveraging information from the Vectorization Analysis. The inner loop that iterates over the dimension chosen for vectorization is incremented by W in each iteration as depicted in Listing 8.1.

We automatically generate a wrapper around the original kernel. This wrapper includes the inner loops while only the outer loops are implemented directly in the driver (to allow multi-threading, e.g. via OpenMP). This enables removal of all overhead of the callback functions: All these calls query information that is either statically fixed (e.g. `get_global_size`, which returns the total number of instances) or only depends on the state of the inner loop's iteration (e.g. for one dimension, `get_global_id` is the work group size multiplied with the work group identifier plus the local identifier of the instance within its work group). The static values are supplied as arguments to the wrapper, the others are computed directly in the inner loops. After the original kernel has been inlined into the wrapper, we can remove all overhead of callbacks to the driver by replacing each call by a direct access to the corresponding value. Generation of the inner loops "behind" the driver-kernel barrier also exposes additional optimization potential of the kernel code together with the surrounding loops and the callback values. For example, loop-invariant code motion moves computations that only depend on work group identifiers out of the innermost loop. This would not be possible if those loops were implemented statically in the driver instead of generated at compile time of the kernel.

8.2.3 Continuation-Based Barrier Synchronization

OpenCL provides the `barrier` statement to implement barrier synchronization of all instances in a work group. A barrier enforces all instances of a work group to reach it before they can continue executing instructions behind the barrier. This means that the current context of an instance needs to be saved when it reaches the barrier, and restored when it continues execution. Instead of relying on costly interaction with the operating system, the following code transformation can be used to implement barrier synchronization directly in the kernel.

Let the set $\{b_1, \ldots, b_m\}$ be the set of all barriers in the kernel. The following, recursive scheme is applied: From the start node of the CFG, start a depth-first search (DFS) which does not traverse barriers. All nodes reached by this DFS are by construction barrier free. The search furthermore returns a set of barriers $B = \{b_{i_1}, \ldots, b_{i_m}\}$ which it hit. At each hit barrier, the live variables are determined and code is generated to store them into a

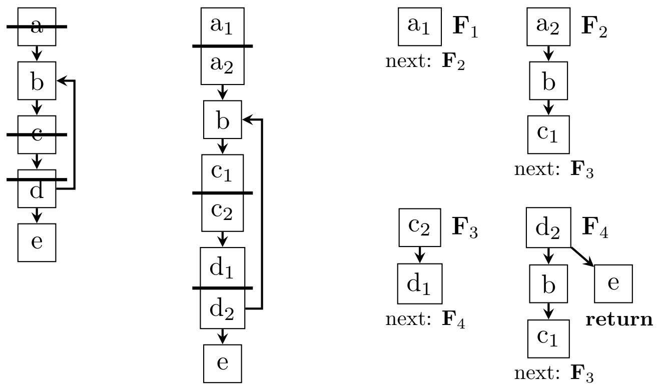


Figure 8.1: Example CFG of a kernel which requires synchronization (the barriers are indicated by the bars crossing the blocks), the CFG after splitting blocks with barriers, and the resulting set of new functions $\{F_1, \ldots, F_4\}$.

structure. For every instance in the group, such a structure is allocated by the driver. In front of each barrier, a return with the identifier of the hit barrier is placed and the block is split at that location. Now, all traversed blocks are extracted into a separate function, and the instructions $b_{i_1}, \ldots, b_{i_m}$ are taken as start points for the next m different functions. For each of these functions, we apply the same scheme until there are no more functions containing barriers. Figure 8.1 gives an example for this transformation.

Finally, we generate a wrapper that chooses the generated function to be executed next by testing the last returned barrier identifier (see Listing 8.2). The resulting code can be seen as a state machine: Each state corresponds to a part of the original kernel that is executed by each instance of the group. Each transition corresponds to crossing a barrier in the original kernel. If multiple barriers were reachable from another one, the state that corresponds to this code region has multiple outgoing transition edges. A transition is triggered only after all instances of the work group have reached the corresponding program point.

Note that the semantics of OpenCL require all instances to hit the *same* barrier, otherwise the program's behavior is undefined. Hence, if not all instances return the same barrier identifier, the kernel is in a bad state anyways. In such a case, we opt to use the identifier returned by the last executed instance.

Listing 8.2 Pseudo code for the kernel of Figure 8.1 after implementation of barriers and before inlining and optimization (1D, computations of `tid` etc. are omitted). The value of `liveValSize` is the maximum size required for any continuation, `data` is the storage space for the live variables of all instances.

```
void
newKernel(T0 param0, ..., TN paramN, int localSize, ...) {
  void* data[localSize/W] = alloc((localSize/W)*liveValSize);
  int next = BARRIER_BEGIN;
  while (true) {
    switch (next) {
      case BARRIER_BEGIN:
        for (int i=0; i<localSize; i+=W)
          next = F1(param0,...,paramN,tid,...,&data[i/W]);
        break;
      case B2:
        for (int i=0; i<localSize; i+=W)
          next = F2(tid, ..., &data[i/W]);
        break;
      ...
      case B4:
        for (int i=0; i<localSize; i+=W)
          next = F4(tid, ..., &data[i/W]);
        break;
      case BARRIER_END: return;
} } }
```

8.2.4 Experimental Evaluation

Our OpenCL driver is based on the LLVM compiler framework [Lattner & Adve 2004]. We use the front end of the AMD APP SDK (version 2.1) to produce LLVM IR. We did not attempt to implement the full OpenCL 1.2 API rather than a sufficiently complete fraction to run a variety of benchmarks. In some cases (denoted “Scalar”), we modified the benchmarks to only use scalar values instead of the OpenCL built-in vectors to allow for automatic vectorization. All experiments were conducted on an Intel Xeon E5520 at 2.26 GHz with 16 GB of RAM running Ubuntu Linux 13.04 64 bit. The vector instruction set is Intel’s SSE 4.2, yielding a SIMD width of four 32 bit values. The machine ran in 64 bit mode, thus 16 vector registers were available.

We report kernel execution times of our driver in different configurations and compare kernel execution time and total wall clock time to the latest Intel, AMD, and POCL CPU drivers.

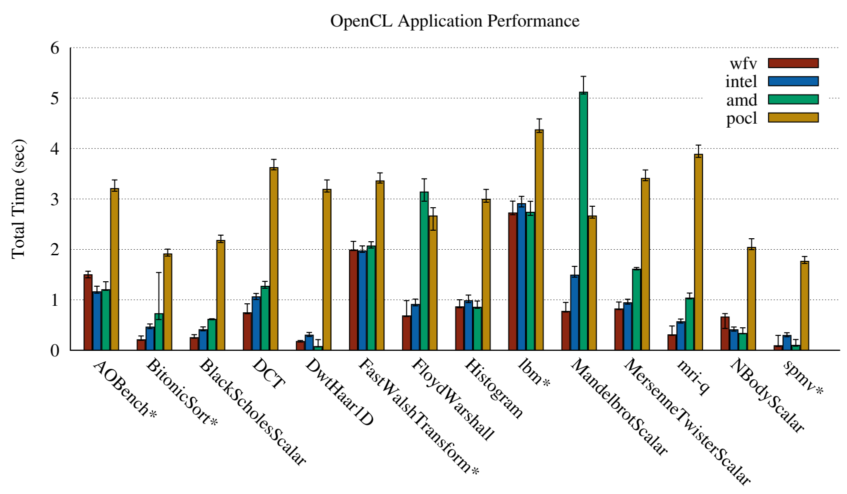


Figure 8.2: Performance comparison of total runtime, which includes everything including OpenCL API calls, optimization, WFV, and code generation. Values are median execution times of 200 individual runs without warm-up phase, minimum and maximum execution times are shown with error bars.

8.2.4.1 Benchmark Applications

The benchmark applications are selected from the AMD APP SDK[4], the Parboil benchmark suite [Stratton et al. 2012], and AOBench[5]. To cover a diverse set of real-world problems ranging from sorting algorithms over stock option estimation to physics simulations and computer graphics, we chose the following applications: `AOBench` (1K×1K pixels), `BitonicSort` (1M elements), `BlackScholesScalar` (16K elements), `DCT` (8K×8K elements), `DwtHaar1D` (30K elements), `FastWalshTransform` (50M elements), `FloydWarshall` (1K elements), `Histogram` (16K×16K elements), `MandelbrotScalar` (8K×8K pixels), `MersenneTwisterScalar` (32K elements), `NBodyScalar` (8K elements). The Parboil benchmarks `lbm`, `mri-q`, and `spmv` were set to use the default inputs. Barrier synchronization is used in the applications `BinomialOptionScalar`, `DCT`, `DwtHaar1D`, `Histogram`, and `NBodyScalar`.

[4]`developer.amd.com/sdks/AMDAPPSDK`, version 2.8.1
[5]`code.google.com/p/aobench`

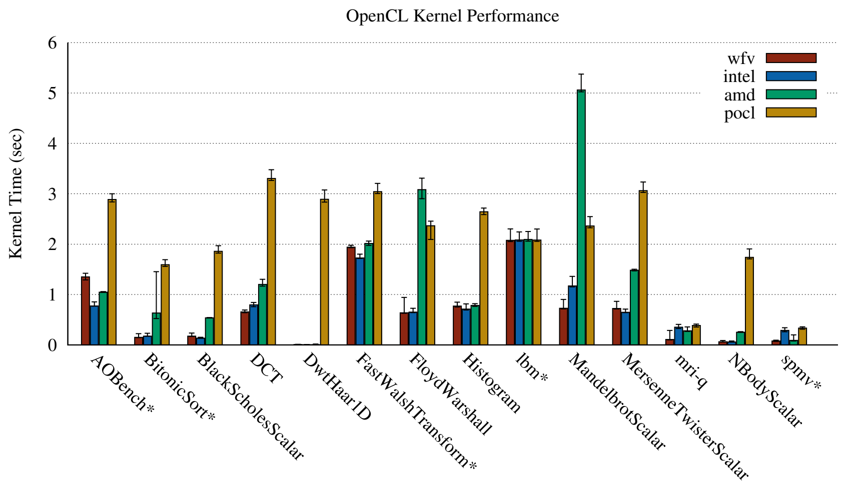


Figure 8.3: Kernel performance comparison of our WFV-based OpenCL driver, the proprietary Intel and AMD drivers, and the POCL driver. Values are median execution times of 200 individual runs without warm-up phase, minimum and maximum execution times are shown with error bars.

8.2.4.2 Comparison with other CPU Drivers

For a fair comparison to other available drivers we implemented a naïve, unoptimized multi-threading scheme that uses OpenMP. Figure 8.2 shows application performance of the entire driver, including calls to the OpenCL API, optimization, WFV, and code generation for the kernel. It is important to say that the WFVOpenCL driver does not implement the full OpenCL API, and thus is likely to benefit from reduced overhead when compared to the other drivers. Thus, we now focus on the kernel execution time to discuss the effects of WFV and the different analyses.

Figure 8.3 and Table 8.1 show that our custom driver significantly outperforms the AMD and POCL drivers in almost all scenarios (geometric mean speedups of 1.87 and 6.0). The picture is different when comparing to the state-of-the-art Intel driver, which is on par with our implementation—each driver is faster than the other for 7 out of 14 benchmarks, the geometric mean of the speedups favors our implementation with an average of 1.03.

It is important to note that in spite of including a WFV implementation, the Intel driver *refuses* to vectorize the kernels of the `AOBench`, `BitonicSort`,

Table 8.1 Median kernel execution times (in milliseconds) of WFVOpenCL (vectorized and multi-threaded) compared to the Intel, AMD, and POCL drivers. "Speedup" compares our driver to the Intel driver. Note that the Intel driver does not vectorize kernels marked with an asterisk. These are the numbers for our driver in scalar mode: `AOBench` 2,123 (0.36×), `BitonicSort` 166 (1.04×), `FastWalshTransform` 1,628 (1.06×), `spmv` 65 (4.42×).

Application	WFV	Intel	AMD	POCL	Speedup
`AOBench*`	1,350	773	1,047	2,886	0.58×
`BitonicSort*`	149	173	634	1,592	1.16×
`BlackScholesScalar`	173	134	538	1,857	0.77×
`DCT`	653	794	1,200	3,304	1.22×
`DwtHaar1D`	6	1	2	2,891	0.17×
`FastWalshTransform*`	1,927	1,726	2,014	3,042	0.90×
`FloydWarshall`	636	651	3,082	2,364	1.02×
`Histogram`	762	712	782	2,643	0.93×
`lbm`	2,062	2,084	2,093	2,082	1.01×
`MandelbrotScalar`	724	1,170	5,059	2,361	1.62×
`MersenneTwisterScalar`	719	648	1,483	3,062	0.90×
`mri-q`	108	351	279	385	3.25×
`NBodyScalar`	60	57	252	1,737	0.95×
`spmv*`	75	287	89	334	3.83×
Average WFV Speedup	-	1.03×	1.87×	6.0×	1.03×

`FastWalshTransform`, and `spmv` benchmarks. This means that Intel's heuristics deem the code to not benefit from vectorization, although this is not true for `AOBench` and `BitonicSort`.

8.2.4.3 Impact of Analyses & Code Generation

To assess the impact of our analyses and code generation techniques, we compare the OpenCL driver in different configurations that enable or disable certain analyses and optimizations of WFV. If *all* analyses are disabled, all properties are set to the least informative values of their corresponding lattices. This effectively results in W-fold splitting of each statement without usage of vectors or vector operations. Since this is not very meaningful for a vectorization algorithm, the default mechanism when all analyses are disabled (`naïve`) is to initialize all program points with `unknown`. This means that no optimizations that are based upon `uniform` values have any effect, but only those operations that have to are split and/or guarded. Summarized, these are the tested configurations:

- The `scalar` configuration does not perform WFV.

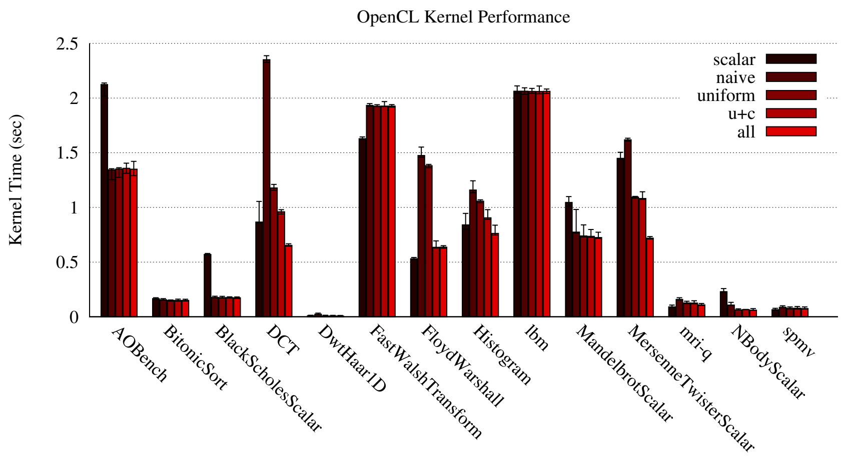


Figure 8.4: Kernel performance comparison of different configurations of the WFV implementation. Values are median execution times of 200 individual runs without warm-up phase, minimum and maximum execution times are shown with error bars.

- The `naïve` configuration initializes all values with `unknown`. This disables all optimizations that exploit `uniform` values and `consecutive` memory access operations. Also, control flow is fully linearized since the Rewire Target Analysis is disabled.
- The `uniform` configuration initializes all values with `uniform/aligned`, but everything that would be marked `consecutive` is automatically lifted to `unknown`. This enables usage of `uniform` values.
- The `uniform+consecutive` (`u+c`) configuration employs `uniform`, `consecutive`, and `aligned`, i.e., it optimizes memory access operations.
- The `all` configuration finally enables all analyses, i.e., it now also employs the Rewire Target Analysis to retain structure of the CFG.

Figure 8.4 shows an overview of the performance of the different WFV configurations. The concrete numbers can be obtained from Table 8.2.

The overall observation is that performance improves with the addition of analyses and optimizations. In the following paragraphs, we assess the impact of gradually enabling more parts of our analyses. We chose to use an accumulative scheme since the analyses build on top of each other. For

Table 8.2 Median kernel execution times (in milliseconds) of our OpenCL driver in different configurations for different applications ($W = 4$). `u+c` stands for `uniform+consecutive`. The column "Speedup" shows the effect of our optimizations, comparing `all` to `naïve`. The average speedup denotes the geometric mean of the speedups compared to the next left configuration (e.g. `all` vs. `u+c`).

Application	`scalar`	`naïve`	`uniform`	`u+c`	`all`	Speedup
`AOBench`	2,123	1,344	1,352	1,358	1,350	1.00×
`BitonicSort`	166	156	149	149	149	1.05×
`BlackScholesScalar`	570	177	175	175	173	1.02×
`DCT`	866	2,351	1,178	960	653	3.60×
`DwtHaar1D`	9	23	7	7	6	3.83×
`FastWalshTransform`	1,628	1,933	1,926	1,927	1,927	1.00×
`FloydWarshall`	530	1,475	1,382	635	636	2.32×
`Histogram`	840	1,159	1,057	905	762	1.52×
`lbm`	2,076	2,062	2,062	2,060	2,062	1.00×
`MandelbrotScalar`	1,046	774	739	735	724	1.07×
`MersenneTwisterScalar`	1,449	1,618	1,094	1,082	719	2.25×
`mri-q`	90	159	126	125	108	1.47×
`NBodyScalar`	229	107	63	65	60	1.78×
`spmv`	65	87	77	78	75	1.16×
Average Speedup	-	0.86×	1.28×	1.09×	1.11×	1.53×

example, it does not make sense to classify `consecutive` memory operations without determining which values are `uniform`.

Naïve Vectorization. The first observation is that even naïve vectorization can yield significant speedups. For example, `BlackScholesScalar` improves by a factor of 3.22, and `NBodyScalar` by a factor of 2.14. These are benchmarks that are dominated by arithmetic operations with only few different control flow paths. This makes them perfect targets for WFV.

However, as the geometric mean of the speedups of 0.86 shows, naïve vectorization is more often inferior to scalar execution than it improves the performance. This is the case for 8 out of the 14 benchmarks chosen, for example for the `DCT`, `DwtHaar1D`, and `Histogram` applications. The reason for these slowdowns is that the vector code generated without any additional information has to be too conservative: All memory operations have to be executed sequentially, all operations which may have side effects have to be guarded by `if` statements, all control flow is linearized, and so on. This highlights the importance of additional analyses and optimizations.

Uniform Operations. Retaining `uniform` values proves to be effective in basically all of the cases with an average speedup factor of 1.28 compared to naïve vectorization. For some benchmarks like `DwtHaar1D`, `DCT`, or `NBody`, the impact of classifying operations as `uniform` instead of `varying` has a huge impact. Their kernels exhibit large amounts of operations that can remain scalar, reducing the pressure on the vector unit. A big part of the improvement for `DwtHaar1D` is that there are calls to `sqrt` with a `uniform` argument. If this call is not marked `uniform`, as in naïve vectorization, it is lifted to `unknown`/`guarded`, and thus has to be executed sequentially and guarded by conditionals.

Optimized Memory Access Operations. Exploiting `consecutive` properties by using vector memory operations yields an additional average speedup factor of 1.09. This is especially effective for `DCT` and `FloydWarshall`. After exploiting `uniform` values, their kernels are dominated by memory access operations. Consider the FloydWarshall kernel: After common subexpression elimination, it only consists of 2 multiplications, 3 additions, 3 loads, and 2 stores. The Vectorization Analysis determines that the loads access `consecutive` addresses, so a single vector instruction can be used per load instead of W sequential loads followed by a merge operation. The store operations depend on control flow, so they still must be `guarded`, i.e., executed sequentially with each statement guarded by a conditional.

Retaining Uniform Control Flow. Finally, our last configuration, `all`, employs the Rewire Target Analysis. This enables the CFG linearization phase to retain parts of the control flow graph where it could be proven that the instances cannot diverge. This results in less code being executed, less register pressure, and less overhead for mask and blend operations. This also enables further optimization by allowing the Vectorization Analysis to be more precise. In Histogram, for example, there is a store operation inside a loop. Conservatively, it has to be executed sequentially and guarded to account for possibly diverging control flow. However, the loop is proven not to diverge, and furthermore to always be executed by all instances. This allows us to issue a vector store during instruction vectorization.

The effect of the Rewire Target Analysis can be best observed for `DCT`, `Histogram`, and `MersenneTwister`. These benchmarks profit most from the control-flow related improvements, with speedup factors of 1.47, 1.19, and 1.50. As expected, there is no effect on benchmarks that do not have any non-divergent control flow, such as `BitonicSort`, `FastWalshTransform`, or

`FloydWarshall`. The average speedup compared to the `uniform+consec-utive` configuration is 1.11. The average speedup compared to the `naïve` configuration is 1.53. The final speedup achieved over scalar execution is on average 1.36 for these benchmarks.

8.2.4.4 Applications Not Suited for WFV

Despite our efforts, there are still benchmarks where our static analyses cannot infer enough information to generate code that is more efficient than the original, scalar code. For example, `FastWalshTransform` is dominated by memory access operations. Although they access `consecutive` addresses in most cases, this is not *always* the case. Thus, WFV conservatively has to insert sequential operations, and stores in addition have to be guarded. This is a perfect example for approaches that exploit dynamic properties (see Section 8.2.5).

On the other hand, there are also applications that are generally not well suited for vectorization for current CPU architectures. For example, this is the case for the `FloydWarshall` kernel: While 2 of the 3 load operations can be vectorized, and the third can remain `uniform`, the store operations depend on control flow. The control flow in turn depends on the loaded values, so the branching behavior is input-dependent. This forces to issue guarded store operations, which result in overhead that outweighs the improved loads and arithmetic operations. It may again be beneficial to use dynamic variants to determine whether the control flow actually does diverge, but since the property entirely depends on the input, this is a hard task to do for a compiler without domain-specific knowledge or additional user input (e.g. annotations).

8.2.5 SMT-Based Memory Access Optimization

We evaluate the effect of our improved code generation for memory accesses for the applications that contain the kernels we discussed in Section 5.8: `FastWalshTransform` and `BitonicSort`. To this end, we employ our SMT-based approach to generate machine code.

In each kernel there are actually several memory operations, which happen to lead to the same satisfiability problems $\neg\varphi(w, a)$. It is worth noting that for the majority of kernels that we found in the AMD APP SDK, the memory address computations are so simple that the relevant equations are decided already via term simplification, i.e., without any non-trivial SMT solving. Nevertheless, our SMT-based technique is to our knowledge the

Table 8.3 Median of kernel execution times (in milliseconds) of 1000 executions with the SMT-optimized configuration of our OpenCL driver compared to the previously best configuration (`all`) and scalar execution (`scalar`). The Speedup column shows the effect of our SMT-based memory access optimization, comparing `all+smt` to `all`. The input sizes were set to 1M elements for both applications.

Application	`scalar`	`all`	`all+smt`	Speedup
`FastWalshTransform`	303	309	299	1.03×
`BitonicSort`	166	149	71	2.1×

first one that enables the compiler to generate better code in less simple cases such as `FastWalshTransform` and `BitonicSort`. Hence, if maximum performance of a kernel with complex memory operations is desired, our approach is the only currently available option.

In Table 8.3, we report kernel execution times of our SMT-enhanced driver in different configurations.[6] The results clearly demonstrate the applicability of our approach. For `FastWalshTransform`, this is the first time that we were able to beat the scalar implementation with the WFV-based one. It turns out, however, that the optimized code can be executed in only one out of W cases, which limits the speedup to 3 percent. The situation is different for `BitonicSort`. Here, the optimized code is executed in the majority of cases, which results in an immense speedup factor of 2.1.

As mentioned before, it is remarkable to note that the Intel driver refuses to vectorize either of the two kernels. The reasons are probably that their heuristics consider the memory operations to dominate the runtime and that they cannot be assumed `consecutive`.

[6]These experiments were conducted on a Core 2 Quad at 2.8 GHz with 8 GB of RAM running Ubuntu Linux 64 bit. The vector instruction set is Intel's SSE 4.2, yielding a SIMD width of four 32 bit values.

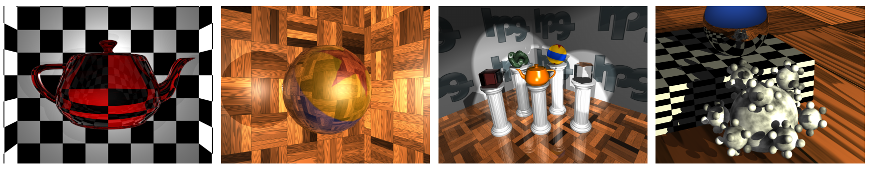


Figure 8.5: Scenes rendered with the ray tracers RTfact (first three) and Manta (rightmost). The surface shaders are written in RenderMan or scalar C++. The AnySL system loads, specializes, optimizes, and vectorizes the shaders and seamlessly integrates them into the renderer at runtime.

8.3 AnySL: Efficient and Portable Shading for Ray Tracing

In computer graphics, *shading* plays an important role: Graphics engines use programs called *shaders* to provide flexibility for certain aspects of the rendering process, such as the appearance of surfaces, the emission of light sources, or the displacement of geometry.

Often, shaders are the most performance-critical parts of a graphics engine. This is because they are called in the innermost loop of the renderer and have to simulate complex physical reflectance properties. At the same time, in professional environments such as special effects in the movie industry, game development, or photo-realistic rendering for product advertisement, shaders have to be written by artists rather than programmers. Therefore, so-called *shading languages* have been developed: domain-specific languages that facilitate the task of describing physical properties of a surface at a specific point, abstracting away from the underlying implementation. Popular examples for these languages are RenderMan [Apodaca & Mantle 1990], HLSL [Peeper & Mitchell 2003], or GLSL [Rost et al. 2004]. Another shading language that also employs automatic vectorization, albeit on the abstract syntax tree level, is RTSL [Parker et al. 2007].

AnySL [Karrenberg et al. 2010] is a shading system that aims at providing a flexible and portable, yet efficient way of shading and is easily integrated into an existing renderer. For this evaluation, we focus on the efficiency, for which WFV plays the major part: Shading languages use a sequential code model, in which a shader is written to evaluate a single point of a surface. Yet, the code is executed for every visible point of that surface independently.[7] This data-parallel execution can be exploited by evaluating multiple surface points in parallel with multi-threading and WFV. Figure 8.5 shows example images generated with AnySL.

8.3.1 WFV Integration

AnySL loads the scalar shader at runtime, compiles it to LLVM bitcode, specializes it to the shading system, and optimizes it. Then, WFV is used to produce a shader that evaluates multiple surface points at once. The whole procedure is efficient and allows for recompilation at runtime. For example, shader parameters can be modified at runtime, which invokes an immediate recompilation to get the best possible code.

Instead of shading each individual point immediately, the shading system collects information about W points before calling the vectorized shader. Thus, there is no `tid` as in OpenCL or CUDA, but vectorization is triggered by passing vectors of input values instead of scalar values. For example, the scalar shader receives scalar values for the coordinates of the surface point, the viewing direction, texture coordinates, and other parameters. The vectorized shader on the other hand directly receives W coordinates at once, which means the analysis will mark the corresponding input parameter `varying`. This still allows the use of `uniform` values. For example, in a ray tracer with a pinhole camera, all primary rays always have the same origin.

Function calls play a major role in shaders. For example, RenderMan uses a *direct* shading model, where the shader itself does all computations required to produce a final color value. In the case of ray tracing, this may include shooting additional rays with the function `traceRay`, e.g. for transparency effects or to determine whether light sources are obstructed. Shooting additional rays however is the task of the ray tracing engine, so the shader issues a recursive call to the renderer. In the vectorized context, this function call will often occur for many of the active instances that execute the shader. In the case of SIMD-based ray tracers such as RTfact or Manta, vectorized functions to trace multiple rays at once already exist in the renderer. Thus, WFV has to be made aware of the available mapping of the scalar `traceRay` function to its vector implementation. Also, if the call is inside some potentially diverging control flow statements, the vector implementation can provide a mask parameter. WFV automatically passes the active mask of that block and thus treats calls to `traceRay` as `varying` instead of `sequential` or `guarded`.

[7] Note that this is not generally true: There are shaders that can query information from neighboring surface points.

8.3.2 Experimental Evaluation

To evaluate AnySL with WFV, we integrated it into the real-time ray tracers RTfact [Georgiev & Slusallek 2008] and Manta [Bigler et al. 2006] and adapted shading language front ends for RenderMan and C++. We evaluated the performance of a diverse set of procedural shaders. AnySL achieves an average rendering speedup factor of 3.8 in RTfact thanks to the usage of WFV when compared to sequential shading.

8.3.2.1 Setup

All experiments were conducted on a Core 2 Quad (Q9550) with a SIMD width of 4 (SSE4.1) at a clock rate of 2.8GHz and 4GB of memory running Ubuntu Linux 9.10. The resolution was set to 512x512 pixels.

To improve the performance of complex procedural shaders that use a noise function [Perlin 1985; 2002], we implemented a variant of noise optimized for vectorization. The original noise function uses complicated branching patterns and repeated indexing into a permutation table, which requires splitting and reassembling of packets due to the lack of a scattered load instruction in SSE. Our optimized variant is entirely branch-free and generates the pseudo-random numbers on the fly instead of using a table, resulting in similar performance gains for shaders with noise components as for those without. This variant is similar to how the noise function is implemented on GPUs [Olano 2005]. Note that our noise function is written in scalar code and automatically vectorized together with the rest of the shader.

8.3.2.2 RTfact

We compare the rendering performance of automatically vectorized shaders to scalar shading where packets are split and the scalar shader is executed sequentially. This is really the only option for non-trivial shaders if automatic vectorization is not available. Because of the complexity of writing vectorized shaders by hand, we can only compare to a few simple, manually vectorized shaders. Note that these hand-written shaders are directly integrated into the rendering system, and thus allow for more efficient invocation compared to shaders that are dynamically compiled. Also, these shaders cannot be modified at runtime.

The vectorized versions of the shaders outperform their scalar counterparts by an average factor of 3.8 (see Table 8.4). Notice that we even achieve super linear speedups of factors up to 5.0. This has two sources: First, RTfact

Table 8.4 Performance of RTfact (in frames per second) for different RenderMan shaders in scenes with one shader on a (triangulated) sphere and two point light sources. Due to the difficulties of writing packet shaders by hand, we can only compare with two hand-optimized shaders, which show a performance difference of less than 10%.

Shader	Manual	Scalar	WFV	Speedup
Brick	-	8.8	31.4	3.6x
Checker	34.5	8.8	31.8	3.6x
Glass	-	0.9	4.5	5.0x
Granite	-	7.2	24.6	3.4x
Venus	-	7.6	25.7	3.4x
Parquet	-	4.3	18.6	4.3x
Phong	35.5	14.1	32.5	2.3x
Screen	-	4.6	22.7	4.9x
Starball	-	4.5	20.0	4.4x
Wood	-	4.4	19.1	4.3x
Average	-	6.5	23.1	3.8x

is based entirely on SIMD values. This means that when executing scalar shaders, the packets that hold the parameters for multiple rays first have to be extracted, and results have to be packed to vectors again. This overhead does not occur when using vectorized shaders. Second, cache coherence improves when shading multiple points at a time, especially for shaders with a lot of code.

8.3.2.3 Manta

Manta, although also being a packet ray tracer, internally differs largely from RTfact: The internal algorithms of RTfact are entirely based upon SIMD primitives and operations on those, whereas Manta only employs vectors and vector operations in certain parts of its algorithms and requires careful tuning by hand. This is especially visible for calls from the shader back to the rendering engine (such as `illuminance()`). These calls are highly performance-critical and are required to operate seamlessly on SIMD data types to expose the full potential of the vectorized shaders.

In combination with our limited knowledge of Manta, these issues do not make the renderer a choice as ideal for our automatically vectorized shaders as RTfact. Despite these drawbacks, Manta still reaches an average speedup of 1.6 with peaks up to 2.5 for some shaders in our experiments.

8.4 Noise: On-Demand Loop Vectorization

When code has to be tuned to achieve maximum performance, automatic optimization strategies such as "-O2" often do not produce the code that the programmer has in mind. This is especially true when it comes to automatically vectorizing loops. There are various reasons for this:

- Static analysis results may be too imprecise to ensure the correctness of the transformation.
- Some cost functions simply tell the compiler to *not* carry out the transformation.
- An optimization chosen by the compiler can, under certain circumstances, turn out to be detrimental.
- The compiler sometimes chooses a suboptimal order of optimizations, known as the *phase ordering problem* [Touati & Barthou 2006].

Therefore, programmers often try to outsmart the compiler by manually "optimizing" the code. This costs time, is error prone, and makes the code illegible and unmaintainable. Even worse, it does not scale with the number of target architectures because every one might need different optimizations. Hence, many variants of the same piece of code have to be maintained.

Noise is a language extension that allows a programmer to create custom optimization strategies and apply them to specified code segments.[8] This enables fine-grained control over the optimizations applied by the compiler to conveniently tune code without actually rewriting it. With Noise, the programmer can easily choose optimizations and their order without interfering with the standard optimizations being applied to the remaining program. In particular, it is possible to annotate loops that are known to be vectorizable. This is especially important for legacy code in the High-Performance Computing (HPC) environment but is also relevant in other performance-sensitive fields such as computer graphics.

Consider the example in Listing 8.3. Assume that the loop should be transformed as shown in Listing 8.4: fuse the loops, inline the function call, perform loop-invariant code motion, vectorize the loop and unroll it. It is safe to assume that off-the-shelf compilers such as Clang, GCC, or ICC fail to produce exactly the desired code. On the other hand, our Noise-enabled branch of the Clang compiler allows the programmer to achieve the desired result without rewriting the code manually. In fact, evaluating

[8] Noise is joint work with Roland Leißa, Marcel Köster, and Sebastian Hack for the ECOUSS project (`ecouss.dfki.de`).

Listing 8.3 Noise applied to a code region. Optimizations applied in order: Loop fusion, inlining of `bar`, loop-invariant code motion, loop vectorization, unrolling, standard optimizations for cleanup.

```
float bar(float x) { return x + 42.f; }

void foo(float x, float* in, float* out, int N) {
  NOISE("loop-fusion inline(bar) licm "
        "vectorize(8) unroll(4) -O2")
  {
    for (int i=0; i<N; ++i) {
      float lic = x * bar(x);
      out[i] = in[i] + lic;
    }
    for (int i=0; i<N; ++i) {
      out[i] *= x;
    }
  }
}
```

Listing 8.4 C++-Code showcasing the desired result of Listing 8.3. Noise produces equivalent assembly or LLVM IR.

```
void foo(float x, float* in, float* out, int N) {
  float  lic  = x * (x + 42.f);
  __m256 licV = _mm_set1_ps256(lic);
  __m256 xV   = _mm_set1_ps256(x);
  int i = 0;
  if (N >= 32)
  {
    for ( ; i<N; i+=32) {
      __m256* inV  = (__m256*)(in+i);
      __m256* outV = (__m256*)(out+i);
      outV[0] = _mm_mul_ps256(
                  _mm_add_ps256(inV[0], licV), xV);
      outV[1] = _mm_mul_ps256(
                  _mm_add_ps256(inV[1], licV), xV);
      outV[2] = _mm_mul_ps256(
                  _mm_add_ps256(inV[2], licV), xV);
      outV[3] = _mm_mul_ps256(
                  _mm_add_ps256(inV[3], licV), xV);
    }
  }
  for ( ; i<N; ++i) {
    out[i] = (in[i] + lic) * x;
  }
}
```

the performance and adjusting the optimizations that are applied is now possible without touching any of the actual code.

Other compilers offer features like `#pragma ivdep`, `#pragma simd`, or `#pragma parallel`. Our loop vectorization approach is basically the same as using `#pragma simd`, e.g. in OpenMP [Klemm et al. 2012], which forces loop vectorization by disabling alias and dependency analyses etc. However, these `#pragmas` are restricted to the loop vectorization available in the corresponding compiler, which are currently more limited than WFV, e.g. in their support of nested control flow, non-vectorizable operations, or function calls. Also, these approaches require explicit identification of reductions in the #pragma, i.e., the user has to annotate each reduction variable and the kind of reduction. This is not required when using Noise, which recognizes *loop-carried dependencies*, a more general class of reductions, automatically (Section 8.4.2).

Several compilers also offer `#pragma optimize` as a means to control the optimization level on a per-function basis. To our knowledge there are no implementations that allow to use it within a function body. Also, there is no fine-grained control over the optimizations or the order of optimizations applied. In contrast, Noise allows to annotate compound statements and loops as well, and provides the option to specify exactly which phases should be run.

We implemented Noise for C/C++ using attributes within the Clang front end for LLVM. In addition to exposing LLVM's internal optimization phases, Noise also provides special built-in transformations. This includes explicit inlining and loop unrolling as well as other custom loop transformations—in particular data-parallel loop vectorization using WFV.

8.4.1 WFV Integration

In addition to the LLVM-internal phases `bb-vectorize`, `slp-vectorize`, and `loop-vectorize`, Noise provides `wfv-vectorize`. This loop optimization can be used to vectorize data-parallel loops. It basically is a wrapper around WFV that performs loop-specific analyses and transformations before vectorizing the loop body. The wrapper employs similar analyses as LLVM's loop vectorizer to determine whether the loop is vectorizable, e.g. there has to be a loop induction variable which can be vectorized. It also introduces additional loops if the iteration count is not a multiple of the requested vectorization width or unknown. *Loop-carried dependencies* with certain restrictions that allow the loop to still be vectorized by classic loop vectorization approaches are called *reductions*. In contrast to previous approaches, `wfv-vectorize` cannot only handle reductions, but arbitrary loop-carried dependencies (see Section 8.4.2).

Another advantage of our approach is that, to our knowledge, WFV is currently more advanced than available loop vectorizers when it comes to the actual vectorization of the loop body. For instance, it can handle arbitrary control flow including loops with multiple exits, operations with side effects, or operations without vector equivalent.

8.4.2 Loops with Loop-Carried Dependencies

Definition 19 (Loop-Carried Dependency (LCD)) *A* loop-carried dependency *is a strongly connected component of a set of operations in a loop, i.e., the dependencies between the operations form a cycle.*

This means that the result of an operation o in iteration i is used in the next iteration $i + 1$ to compute the next result of o. Consider the simple example on the left of Listing 8.5: The variable r is updated in every iteration of the loop, referencing its own value from the last iteration in the "$+=$"-operation. In SSA, this code is represented by cyclic def-use chains with a ϕ-function in the header of the loop. For more intuitive presentation, we resort to non-SSA examples here, although the presented transformation directly works on the SSA CFG.

LCDs are problematic when it comes to loop vectorization or loop parallelization in general: The prerequisite that each iteration is independent does no longer hold. However, there is a common optimization that targets a subset of LCDs, which are called *reductions*:

Definition 20 (Reduction) *A* reduction *is a loop-carried dependency with the following properties:*

- *the involved operations all use the same operator,*
- *the operator is associative and commutative,*
- *there is no use of a value of the LCD inside the loop that does not belong to the LCD itself, and*
- *there is exactly one use of a value of the LCD that is outside the loop.*

Reduction operators that are usually recognized by compilers are `add`, `mul`, `min`, and `max`. If their type is `float`, compiler flags like `fast-math` or `associative-math` are required to allow their optimization. Current compilers only allow vectorization of a loop if all LCDs are reductions. If an LCD is a reduction, the loop including the reduction operations can be vectorized if

Listing 8.5 Left: A vectorizable loop with a reduction: a loop-carried dependence that can be vectorized using fix-up code. Right: All common compilers are capable of producing similar, vectorized code ($W = 4$).

```
float foo(float x)
{
  float arr[32];
  // initialize array
  // ...
  float r = x;
  for (int i=0;i<32;++i)
  {
    // vectorizable code
    // ...
    r += arr[i];
    // vectorizable code
    // ...
  }
  // code that uses r
  // ...
}
```

```
float foo(float x)
{
  float arr[32];
  // initialize array
  // ...
  float4 rv = <x, 0, 0, 0>;
  for (int i=0;i<32;i+=4)
  {
    // vectorized code
    // ...
    float4 l = load_vec(arr+i);
    rv = add_vec(rv, l);
    // vectorized code
    // ...
  }
  float r = horizontal_add(rv);
  // scalar code that uses r
  // ...
}
```

some *fix-up code* is introduced before and after the loop. Consider the code on the right of Listing 8.5 that employs a vectorized loop: The reduction variable is now a vector where one element is initialized with the initial value of the old reduction variable, and the other elements are initialized with the neutral element of the operation. The modified reduction operation updates each element of the vector individually. Behind the loop, the elements of the vector are combined with the same operation "horizontally." In the example, the elements of the vector are summed up. This scheme is valid because the reduction operations are associative and commutative.

It is easy to see that reductions are only a small subset of LCDs. Existing vectorizing compilers basically use hard-coded sets of patterns to identify reductions. These loop vectorization implementations are aimed at loops for which the operations of the reduction are important for the overall performance of the loop. However, applications may have loops with costly loop bodies where only a cheap LCD occurs. If the loop is not vectorized because that LCD is not a reduction, all performance potential due to vectorization of the rest of the body is lost. This is the case for example for the main loop in the molecular dynamics code evaluated in Section 8.4.3.

Listing 8.6 Left: A vectorizable loop with a non-vectorizable, loop-carried dependence (LCD). The LCD's strongly connected component contains different update operations, some of which are non-associative and non-commutative, intermediate results are used, and some operations depend on control flow. No current compiler can vectorize this loop. Right: Vectorized loop with unrolled LCD ($W = 2$). The unrolled LCD itself does not necessarily improve performance, but it enabled vectorization of the *rest* of the loop body (omitted code at A, B, and C).

```
float bar(float x) { /* ... */ }

float foo(float x,
          float* in,
          float* out)
{
  float r = x;
  for (int i=0; i<32; ++i)
  {
    // vectorizable code (A)
    // ...
    r += in[i];
    out[i] = r;
    // vectorizable code (B)
    // ...
    if (r<in[i]) r=bar(r);
    // vectorizable code (C)
    // ...
    ... = r;
  }
  return r;
}
```

```
float foo2(float x,
           float* in,
           float* out)
{
  float r = x;
  float2 rv;
  for (int i=0; i<32; i+=2)
  {
    // vectorized code (A)
    // ...
    r += in[i];
    rv[0] = r;
    out[i] = r;
    if (r<in[i]) r=bar(r);
    r += in[i+1];
    out[i+1] = r;
    rv[1] = r;
    if (r<in[i+1]) r=bar(r);
    // vectorized code (B)
    // ...
    // vectorized code (C)
    // ...
    ... = rv;
  }
  return r;
}
```

8.4.2.1 Vectorization of Loops with Arbitrary LCDs

The `wfv-vectorize` phase of Noise includes a transformation that allows to vectorize loops with arbitrary loop-carried dependencies. It works as follows: First, LCDs that are no reductions are separated from the rest of the loop body. This is achieved by extracting the code that corresponds to one LCD into a new function. It is necessary to separate the code since the reduction code has semantics that cannot be represented in WFV: It must not be vectorized because that would violate the cyclic dependencies. Second, the LCD is duplicated W times, where each duplicated region's inputs are connected to the previous region's output and function arguments. The inputs of the first region correspond to the arguments of the function, the output of the last region to the returned value. The call to the function has to be placed in such a way that all dependencies are obeyed: It has to be placed behind the definition of the last external input that is required for the LCD, but before the first external use. The function call's properties are set explicitly to prevent the Vectorization Analysis to mark it as `sequential` and `guarded`. Finally, the code is vectorized by WFV, and all LCD functions are inlined again. Listing 8.6 shows an example for vectorization of a loop with such a non-trivial LCD.

It is important to note that, although the LCDs can be of arbitrary shape, their dependencies may in some cases prevent vectorization. This is because our approach has to place the call to the function into which the LCD is extracted at some point where two constraints are met: First, all *operands* of any of the LCD's operations must be available. Second, all *uses* of the result of the current iteration or any intermediate results of the LCD's operations must be reachable, i.e., the call must not be placed below a use. These constraints may sometimes not be met by any location in the function body, which makes vectorization impossible.

8.4.3 Experimental Evaluation

A prototypical implementation of Noise with `wfv-vectorize` was evaluated on several benchmarks.[9] The machine used was an Intel Core i5-2500 with 4 cores running at 3.30GHz. For each benchmark, multiple variants were measured: First, the original code, compiled with Clang. Second, the manually optimized code, compiled with Clang. Third, the original code with Noise annotations that correspond to the manual transformations, compiled with our Noise-enabled Clang.

Figure 8.6 shows the performance of different versions of a *matrix multiplication* application with different loop orders. It can be observed that the manually unrolled code on this machine is usually a bit faster than using Noise. However, this comes at the expense of code that is much harder to maintain. With noise, only a single line has to be changed to modify which

[9]The *CG Solver* and *matrix multiplication* were developed at HLRS Stuttgart and evaluated by Yevgeniya Kovalenko. The *imd* code was developed by Cray, the manually vectorized variant was implemented by Mathias Pütz.

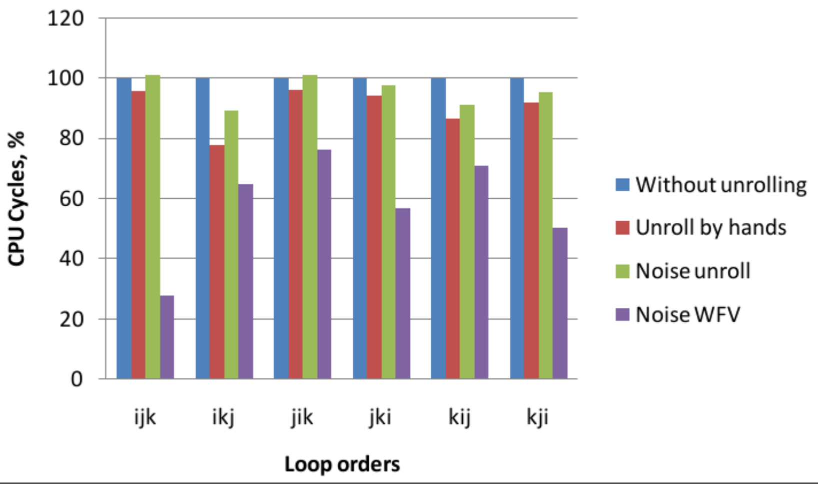


Figure 8.6: Performance effects of Noise `unroll` and `wfv` applied to a matrix multiplication function.[10]

loop should be unrolled how often. The histogram also shows the effect of using `wfv-vectorize` instead of `unroll`. For all loop orders, WFV clearly beats the unrolled versions.

The second application, *CG Solver*, is a conjugate gradient method for sparse matrices in *ellpack-r* format [Vázquez et al. 2011]. We applied the following optimizations both manually and with Noise: function inlining, loop unrolling, loop fusion, loop-invariant code motion, wfv-vectorization. The results are presented in Table 8.5. The manually tuned version runs approximately 11% faster than the original version compiled with "-O2." The Noise version reaches the same speedup without requiring to modify the actual code.

The CG Solver benchmark was also tested with different compilers: icpc 13.0, g++ 4.7.1, clang++ 3.4. None produced similarly efficient code for the most time consuming part of the code, a loop with a sparse matrix-vector multiplication which uses indirect addressing. As Table 8.6 shows, GCC produced the fastest code for the original implementation, followed by Clang and the Intel compiler (icpc). The manually improved code runs fastest when compiled with Clang, and slowest when compiled with the Intel compiler.

[10] Graph courtesy of Yevgeniya Kovalenko, HLRS Stuttgart.

Table 8.5 Execution times (in seconds) of two complex HPC applications. The column "Speedup" shows the effect of our Noise-based optimization, comparing Clang+Noise to Clang. Note that the manually optimized CG Solver does not include vectorization. Performance of Clang+Noise without `wfv-vectorize` matches the performance of Manual.

Application	Clang	Manual	Clang+Noise	Speedup
imd	60.1	39.7	41.5	1.45×
CG Solver	60.7	54.6	50.5	1.20×

Table 8.6 Execution times (in seconds) of the CG Solver application. The Speedup column shows the effect of our Noise-based optimization, comparing Clang+Noise to the other compilers at -O2.

Compiler	-O2	Manual	Clang+Noise	Speedup
Clang 3.4	60.7	54.6	50.5	1.20×
GCC 4.7.1	59.4	55.8	-	1.18×
Intel 13.0	63.8	59.8	-	1.26×

The Noise-improved variant outperforms the manually optimized variant compiled by g++ by 18% and icpc by 26%.

Notice that Clang+Noise includes `wfv-vectorize` for the loop with indirect addressing, while the manually optimized code was not vectorized due to the transformation proving too time-consuming. Without `wfv-vectorize`, the performance of Clang+Noise exactly matches the performance of the manually optimized code compiled with Clang.

The third application is *imd*, a complex molecular dynamics simulation. Table 8.5 shows a 1.45× speedup that is achieved by manual optimization. The main loop of the application was vectorized manually with significant help of domain knowledge: It is impossible for a compiler to determine that the loop can be vectorized because of multiple indirections in memory access operations that depend on input. Noise with `wfv-vectorize` allows to annotate a loop regardless of conservative analysis properties.

Additionally, the main loop exhibits a non-trivial, loop-carried dependency: The value that is updated is loaded from memory and may be a different one in each iteration. This makes it impossible to consider the LCD as a reduction because it is not possible to generate appropriate fix-up code. Indeed, classic approaches that only recognize reductions are unable to vectorize the loop because of this LCD. Since `wfv-vectorize` is able to handle arbitrary LCDs, Noise with WFV successfully vectorizes the loop.

However, this comes at a price: Because of the costly code generated to handle the complex LCD, the performance of the vectorized code does not improve over the original, scalar code.

The performance increase of the manually vectorized code is due to another optimization that requires domain knowledge: The update of the described LCD is a decrement in the `then` part of a `varying if` statement. This significantly complicates the code required to correctly handle the LCD, since the conditional has to be duplicated W times, the corresponding values of the condition have to be extracted, etc. However, the `if` statement can be safely removed: its only effect is to prevent the decrement in cases where the other value would be zero. In the original, scalar code, this increases the number of calculations in the loop body, which results in slightly decreased performance. In the vectorized code, however, control flow has to be linearized anyway. Thus, the removal of control flow does not influence performance negatively and only results in code that is easier to vectorize. In addition, since the branch is `varying`, its removal also improves the precision of the SIMD property analyses, which in turn results in even more efficient code.

The final performance is denoted in Table 8.5: The code generated by Noise performs similarly to the manually optimized variant. However, it is achieved by adding only a single line to the loop and removing an `if` statement. This is in sharp contrast to the significant effort of manually transforming the entire loop to vector code.

9 Conclusion

This thesis presented analyses and algorithms for automatic SIMD vectorization of control flow graphs in static single assignment form. The topic is motivated by the need for compiler techniques that exploit all available parallelism offered by today's hardware. Besides employing all available cores, SIMD instruction sets that are ubiquitous in current architectures also have to be used. The presented algorithm for Whole-Function Vectorization (WFV) is especially well suited for application to data-parallel applications such as particle simulations, stock option prediction, or medical imaging.

WFV performs SIMD vectorization of a function in the form of an arbitrary control flow graph in SSA form. The algorithm is based on a novel, *partial* control-flow to data-flow conversion and efficiently places blend operations for architectures without hardware support for predicated execution.

We furthermore presented a data-flow analysis that determines properties of instructions in the context of SIMD execution. An additional analysis captures behavior of control flow in a SIMD execution. The WFV algorithm leverages the results of these analyses to generate more efficient code than previous approaches.

A variety of case studies showcased the applicability of the system: First, an OpenCL CPU driver based on WFV is on par with the performance of the fastest available driver of Intel, and beats all other available CPU implementations. The optimizations based on the SIMD property analyses sum up to an average speedup of 1.53 compared to naïve vectorization. Second, a real-time ray tracer showed an almost linear average speedup of 3.8 on SSE due to WFV as compared to scalar execution. Finally, WFV was used to implement a loop vectorization transformation in a compiler that allows to specify which optimization phases to run on which code regions. It enabled vectorization of code that static analyses cannot prove safe for vectorization. At the same time, the produced code is as efficient as manually tuned code.

10 Outlook

The topics discussed throughout this thesis open up a plethora of possible directions for future work.

The CFG linearization phase has certain degrees of freedom in the chosen schedule: Consider the diamond-shaped CFG in Figure 10.1. The left CFG corresponds to the original, scalar code. The numbers on the edges represent the values that are live on that edge. Numbers next to blocks represent the maximum number of values that are live at any point in that block. The graph in the middle shows one way of linearizing, the graph on the right another one. Scheduling *c* first yields a maximum number of live values of 6 in block *b* and 9 in block *c*. Scheduling *b* first yields a maximum number of live values of 5 in block *b* and 8 in block *c*. Obviously, the second linearization makes more sense since it reduces the maximum number of values that are live in both blocks. Optimizing the block schedule in such a simple case is trivial. An open question is how to do this optimally in the general case of the Partial CFG Linearization algorithm shown in Section 6.3. This has two reasons: First, more complicated graph structures make the problem more complex, with sometimes more than two possible choices for block schedules. Second, the scheduling order of graphs with `optional` blocks has some constraints that have to be taken into account. These stem from the fact that the partially linearized graph has to obey the mandatory SIMD paths as detailed in Section 6.3.3. In addition, there are often multiple possibilities to obey the mandatory SIMD paths, and they may result in different CFG layouts. Further research is required to quantify the influence of these decisions and come up with heuristics to base the decision upon.

The dynamic variants presented in Chapter 7 also demand further investigation. Most importantly, the development of suitable heuristics is a mandatory next step for each of the presented approaches to work in real-world scenarios. In this context, approaches based on machine-learning should provide a good starting point since there are many parameters to balance, such as the cost for dynamic checks, the cost of a code region, and the probability of execution.

Especially for the complex variants like WFV-SLP and Instance Reordering, the question is open whether they can be useful in a general setting. It

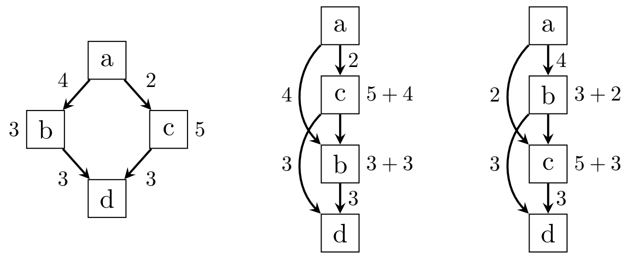


Figure 10.1: Linearization choices can influence register pressure. Numbers on edges represent live values on that edge, numbers next to blocks represent the maximum number of values live at any point in that block.

may be the case that the necessary data reorganization is just too costly for average applications. If this is the case, such transformations should remain an option for special purposes. As such, it would make sense to make them available via annotations that allow the user to guide variant generation explicitly. This could prove generally useful as a feature for compilers of data-parallel languages.

The application of WFV in a GPU context is also worth looking into. Many of the techniques presented in this thesis can be seen as software simulations of GPU hardware features. For example, GPUs have a mechanism called *coalescing* that automatically identifies memory access operations where all addresses are in a certain range and issues a single access operation. For this, the addresses do not even have to be consecutive. However, hardware development on the GPU side also explores different directions: Recent models from AMD also offer an explicit SIMD instructions set in addition to the massive multi-threading for which GPUs are known. WFV seems to be the ideal fit for architectures like this.

On the other hand, the development of CPUs is focusing more and more on features that are important for data-parallel languages. For example, the SIMD register size offered by Intel CPUs is still increasing, reaching 512 bit for commodity hardware with the introduction of AVX-512. Also, the instruction sets gain native support for scatter and gather operations, which greatly simplifies code generation. However, the static analyses presented in Chapter 5 are still required. This is because memory operations that access aligned, consecutive addresses are likely to remain more efficient than complex scatter/gather instructions. Another important aspect is the support of predicated execution by providing dedicated mask registers.

Although this is a very useful feature for data-parallel applications, it is also expensive. The AVX-512 instruction set architecture includes eight dedicated mask registers. Still, it remains to be seen whether future architectures will exhibit mask registers by default. If so, the mask generation phase of WFV does not have to generate explicit mask operations, mask phis, etc. anymore.

Another option is to use the scalar unit to store masks. This would decrease the pressure on the vector registers and use the scalar unit for mask operations. In general, this is already possible on SSE/AVX architectures, but due to a lack of dedicated instructions it involves significant overhead: A mask is "created" in a vector register, since vector comparison operations write to a vector register. The bits that correspond to one lane are set to 1 to represent `true` and to 0 to represent `false`. Moving a mask from a vector register to a scalar register is a simple `movemask` operation. This results in the first W bits of the scalar register being set to 1 if the vector element was non-zero, or 0 otherwise. However, the inverse operation currently does not exist for any of the common instruction sets, so either a lookup-table or an inefficient sequence of bit operations and move instructions has to be used.

Additional aspects of hardware support could be to provide means to test properties more easily, to choose different paths more easily, or to do data layout transformation more efficiently. For example, employing a special branch instruction which tests a mask and transfers control to different successors if the mask is either entirely `true`, entirely `false`, only has one active instance, or a mixture, would make variant generation more efficient. Data layout transformations from "array-of-struct" to "struct-of-array' layouts would certainly be useful, as well as facilities to reorganize instances as described in Section 7.5.

On the theoretical side, complex algorithms such as the Partial CFG Linearization in Section 6.3 require a better foundation. A complete semantics for SIMD execution is required to prove the correctness of such algorithms. The Operational Semantics presented in Section 5.4 may provide a valuable starting point for this.

Bibliography

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Cited on page 11.

Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. (1983). Conversion of control dependence to data dependence. In *POPL* (pp. 177–189).: ACM. Cited on pages 4 and 31.

Allen, R. and Kennedy, K. (1987). Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4), 491–542. Cited on page 31.

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *POPL* (pp. 1–11).: ACM. Cited on page 14.

Apodaca, A. and Mantle, M. (1990). RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, 10(4), 44–49. Cited on page 156.

Asher, Y. B. and Rotem, N. (2008). Hybrid Type Legalization for a Sparse SIMD Instruction Set. *ACM Trans. Archit. Code Optim.*, 10(3), 11:1–11:14. Cited on page 124.

Barik, R., Zhao, J., and Sarkar, V. (2010). Efficient Selection of Vector Instructions Using Dynamic Programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43 (pp. 201–212). Washington, DC, USA: IEEE Computer Society. Cited on page 32.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11 (pp. 171–177). Springer. Cited on page 76.

Benthin, C., Wald, I., Woop, S., Ernst, M., and Mark, W. (2012). Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(9), 1438–1448. Cited on pages 37 and 131.

Betts, A., Chong, N., Donaldson, A., Qadeer, S., and Thomson, P. (2012). GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12 (pp. 113–132). New York, NY, USA: ACM. Cited on page 36.

Bigler, J., Stephens, A., and Parker, S. G. (2006). Design for Parallel Interactive Ray Tracing Systems. *IEEE Symposium on Interactive Ray Tracing*. Cited on page 158.

Blelloch, G. E. et al. (1993). Implementation of a portable nested data-parallel language. In *PPoPP* (pp. 102–111).: ACM. Cited on page 33.

Boulos, S., Wald, I., and Benthin, C. (2008). Adaptive Ray Packet Reordering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (pp. 131–138). Cited on page 37.

Carter, L., Ferrante, J., and Thomborson, C. (2003). Folklore confirmed: reducible flow graphs are exponentially larger. In *POPL* (pp. 106–114).: ACM. Cited on pages 14 and 124.

Cheong, G. and Lam, M. (1997). An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*. Cited on page 32.

Cimatti, A., Griggio, A., Schaafsma, B., and Sebastiani, R. (2013). The MathSAT5 SMT Solver. In N. Piterman and S. Smolka (Eds.), *Proceedings of TACAS*, volume 7795 of *LNCS*: Springer. Cited on page 76.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the second International Symposium on Programming* (pp. 106–130).: Paris. Cited on page 45.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77 (pp. 238–252). New York, NY, USA: ACM. Cited on page 45.

Cousot, P. and Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79 (pp. 269–282). New York, NY, USA: ACM. Cited on page 45.

Coutinho, B., Sampaio, D., Pereira, F., and Meira, W. (2011). Divergence Analysis and Optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)* (pp. 320–329). Cited on pages 35, 36, 103, and 120.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadek, F. K. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 451–490. Cited on pages 12, 13, and 14.

Darte, A., Robert, Y., and Vivien, F. (2000). *Scheduling and Automatic Parallelization.* Birkhauser Boston. Cited on page 31.

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 (pp. 337–340). Springer. Cited on pages 72 and 76.

Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. (2010). Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10 (pp. 353–364). New York, NY: ACM. Cited on page 35.

Eichenberger, A. E., Wu, P., and O'Brien, K. (2004). Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04 (pp. 82–93). New York, NY, USA: ACM. Cited on page 43.

Fireman, L., Petrank, E., and Zaks, A. (2007). New Algorithms for SIMD Alignment. In S. Krishnamurthi and M. Odersky (Eds.), *Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science* (pp. 1–15). Springer Berlin Heidelberg. Cited on page 43.

Fleming, P. J. and Wallace, J. J. (1986). How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM*, 29(3), 218–221. Cited on page 141.

Fritz, N., Lucas, P., and Slusallek, P. (2004). CGiS, a New Language for Data-Parallel GPU Programming. In *VMV* (pp. 241–248). Cited on pages 33, 35, and 36.

Fung, W. and Aamodt, T. (2011). Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (pp. 25–36). Cited on page 47.

Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. (2007). Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40 (pp. 407–420). Washington, DC, USA: IEEE Computer Society. Cited on page 37.

Georgiev, I. and Slusallek, P. (2008). RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (pp. 115–122). Cited on page 158.

Grund, D. (2012). *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University. Cited on page 45.

Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. (2011). Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11 (pp. 62–73). New York, NY: ACM. Cited on page 77.

Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., and Zheng, B. (2010). Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors. In *PACT* (pp. 205–216). New York, NY, USA: ACM. Cited on page 35.

Hecht, M. S. and Ullman, J. D. (1972). Flow Graph Reducibility. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72 (pp. 238–250). New York, NY, USA: ACM. Cited on page 13.

Hormati, A. H., Choi, Y., Woh, M., Kudlur, M., Rabbah, R., Mudge, T., and Mahlke, S. (2010). MacroSS: macro-SIMDization of streaming applications. In *ASPLOS* (pp. 285–296). New York, NY, USA: ACM. Cited on page 33.

Jääskeläinen, P. O., de La Lama, C. S., Huerta, P., and Takala, J. (2010). OpenCL-based design methodology for application-specific processors. In *SAMOS'10* (pp. 223–230). Cited on page 35.

Janssen, J. and Corporaal, H. (1997). Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6), 1031–1052. Cited on pages 14 and 124.

Karrenberg, R. and Hack, S. (2011). Whole-Function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11 (pp. 141–150). Washington, DC, USA: IEEE Computer Society. Cited on page 10.

Karrenberg, R. and Hack, S. (2012). Improving Performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12 (pp. 1–20). Berlin, Heidelberg: Springer-Verlag. Cited on pages 10 and 141.

Karrenberg, R., Košta, M., and Sturm, T. (2013). Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages. In P. Fontaine, C. Ringeissen, and R. Schmidt (Eds.), *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science* (pp. 56–70). Springer Berlin Heidelberg. Cited on pages 10, 72, and 76.

Karrenberg, R., Rubinstein, D., Slusallek, P., and Hack, S. (2010). AnySL: Efficient and Portable Shading for Ray Tracing. In *Proceedings of the Conference on High Performance Graphics*, HPG '10 (pp. 97–105). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. Cited on pages 10, 141, and 156.

Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Cited on pages 11 and 15.

Kim, S. and Han, H. (2012). Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12 (pp. 55–64). New York, NY, USA: ACM. Cited on page 64.

Kim, Y. and Shrivastava, A. (2011). CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA. In *Proceedings of the 48th Design Automation Conference*, DAC '11 (pp. 128–133). New York, NY: ACM. Cited on page 36.

Klemm, M., Duran, A., Tian, X., Saito, H., Caballero, D., and Martorell, X. (2012). Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12 (pp. 59–72). Berlin, Heidelberg: Springer-Verlag. Cited on page 162.

Krall, A. and Lelait, S. (2000). Compilation Techniques for Multimedia Processors. *Int. J. Parallel Program.*, 28(4), 347–361. Cited on page 32.

Kretz, M. and Lindenstruth, V. (2012). Vc: A C++ library for explicit vectorization. *Software: Practice and Experience*, 42(11), 1409–1430. Cited on page 33.

Larsen, S. and Amarasinghe, S. (2000). Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not.*, 35(5), 145–156. Cited on pages 32, 131, and 132.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. Cited on pages 6, 16, 40, and 147.

Lee, Y., Krashinsky, R., Grover, V., Keckler, S., and Asanovic, K. (2013). Convergence and scalarization for data-parallel architectures. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on* (pp. 1–11). Cited on page 36.

Leissa, R., Hack, S., and Wald, I. (2012). Extending a C-like Language for Portable SIMD Programming. In *Principles and Practice of Parallel Programming*. Cited on page 33.

Leissa, R., Haffner, I., and Hack, S. (2014). Sierra: A SIMD Extension for C++. In *Proceedings of the 1st International Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14. Cited on page 33.

LENGAUER, T. AND TARJAN, R. E. (1979). A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1), 121–141. Cited on page 13.

LI, G. AND GOPALAKRISHNAN, G. (2010). Scalable SMT-Based Verification of GPU Kernel Functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10 (pp. 187–196). New York, NY: ACM. Cited on page 36.

LOWRY, E. S. AND MEDLOCK, C. W. (1969). Object Code Optimization. *Commun. ACM*, 12(1), 13–22. Cited on page 12.

LV, J., LI, G., HUMPHREY, A., AND GOPALAKRISHNAN, G. (2011). Performance Degradation Analysis of GPU Kernels. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly 2011*, EC2 '11. Cited on page 36.

MAGNI, A., DUBACH, C., AND O'BOYLE, M. F. P. (2013). A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13 (pp. 11:1–11:11). New York, NY, USA: ACM. Cited on page 32.

MOON, B., BYUN, Y., KIM, T.-J., CLAUDIO, P., KIM, H.-S., BAN, Y.-J., NAM, S. W., AND YOON, S.-E. (2010). Cache-oblivious Ray Reordering. *ACM Trans. Graph.*, 29(3), 28:1–28:10. Cited on page 37.

NEWBURN, C. J., SO, B., LIU, Z., MCCOOL, M. D., GHULOUM, A. M., TOIT, S. D., WANG, Z.-G., DU, Z., CHEN, Y., WU, G., GUO, P., LIU, Z., AND ZHANG, D. (2011). Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *CGO* (pp. 224–235). Cited on page 33.

NGO, V. (1994). *Parallel loop transformation techniques for vector-based multiprocessor systems*. PhD thesis, University of Minnesota-Twin Cities. Cited on page 32.

NUZMAN, D. AND HENDERSON, R. (2006). Multi-platform Auto-vectorization. In *CGO* (pp. 281–294). Cited on page 32.

NUZMAN, D., ROSEN, I., AND ZAKS, A. (2006). Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '06 (pp. 132–143). New York, NY, USA: ACM. Cited on page 64.

Nuzman, D. and Zaks, A. (2008). Outer-loop vectorization: revisited for short SIMD architectures. In *PACT* (pp. 2–11).: ACM. Cited on page 32.

Olano, M. (2005). Modified noise for evaluation on graphics hardware. In *HWWS '05* (pp. 105–110). New York, NY, USA: ACM. Cited on page 158.

Park, J. C. H. and Schlansker, M. (1991). *On Predicated Execution.* Technical Report HPL-91-58, Hewlett-Packard Software and Systems Laboratory, Palo Alto, CA. Cited on pages 3 and 31.

Park, Y., Seo, S., Park, H., Cho, H. K., and Mahlke, S. (2012). SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII (pp. 363–374). New York, NY, USA: ACM. Cited on page 32.

Parker, S. G., Boulos, S., Bigler, J., and Robison, A. (2007). RTSL: A Ray Tracing Shading Language. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07 (pp. 149–160). Washington, DC, USA: IEEE Computer Society. Cited on pages 34, 37, and 156.

Peeper, C. and Mitchell, J. L. (2003). Introduction to the DirectX 9 High-Level Shader Language. Cited on page 156.

Perlin, K. (1985). An Image Synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85 (pp. 287–296). New York, NY, USA: ACM. Cited on page 158.

Perlin, K. (2002). Improving Noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02 (pp. 681–682). New York, NY, USA: ACM. Cited on page 158.

Pharr, M. and Mark, W. R. (2012). ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing (InPar).* Cited on page 33.

Presburger, M. (1929). Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier congres de Mathematiciens des Pays Slaves* (pp. 92–101). Warsaw, Poland. Cited on page 72.

Ramalingam, G. (2002). On Loops, Dominators, and Dominance Frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5), 455–490. Cited on page 13.

Rhu, M. and Erez, M. (2013). Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13 (pp. 356–367). New York, NY, USA: ACM. Cited on page 37.

Rost, R. J., Kessenich, J. M., and Lichtenbelt, B. (2004). *OpenGL Shading Language*. Addison Wesley. Cited on page 156.

Rotem, N. and Ben-Asher, Y. (2012). Block Unification IF-conversion for High Performance Architectures. *Computer Architecture Letters*, PP(99), 1–1. Cited on page 120.

Sampaio, D., De Souza, Rafael, M., Collange, S., and Magno Quintão Pereira, F. (2013). *Divergence Analysis*. Rapport de recherche RR-8411, INRIA. Cited on page 35.

Sartori, J. and Kumar, R. (2012). Branch and Data Herding: Reducing Control and Memory Divergence for Error-tolerant GPU Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12 (pp. 427–428). New York, NY, USA: ACM. Cited on page 37.

Scarborough, R. G. and Kolsky, H. G. (1986). A vectorizing Fortran compiler. *IBM J. Res. Dev.*, 30(2), 163–171. Cited on page 32.

Seidl, H., Wilhelm, R., and Hack, S. (2010). *Übersetzerbau: Analyse und Transformation*. eXamen.press. Springer, 1st edition. Cited on page 46.

Shahbahrami, A., Juurlink, B., and Vassiliadis, S. (2006). Performance Impact of Misaligned Accesses in SIMD Extensions. In *Proceedings of the 17th Annual Workshop on Circuits, Systems and Signal Processing*, ProRISC (pp. 334–342). Veldhoven, The Netherlands. Cited on page 43.

Shin, J. (2007). Introducing Control Flow into Vectorized Code. In *PACT* (pp. 280–291).: IEEE Computer Society. Cited on pages 7, 36, and 138.

Shin, J., Hall, M., and Chame, J. (2005). Superword-Level Parallelism in the Presence of Control Flow. In *CGO* (pp. 165–175).: IEEE Computer Society. Cited on page 32.

Sreraman, N. and Govindarajan, R. (2000). A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.*, 28(4), 363–400. Cited on page 32.

Stanier, J. and Watson, D. (2012). A study of irreducibility in C programs. *Software: Practice and Experience*, 42(1), 117–130. Cited on page 124.

Stratton, J. A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., and Hwu, W.-m. W. (2010). Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10 (pp. 111–119). New York, NY, USA: ACM. Cited on page 35.

Stratton, J. A., Rodrigrues, C., Sung, I.-J., Obeid, N., Chang, L., Liu, G., and Hwu, W.-M. W. (2012). *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.* Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana. Cited on page 148.

Stratton, J. A., Stone, S. S., and Hwu, W.-M. W. (2008). MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In J. N. Amaral (Ed.), *Languages and Compilers for Parallel Computing* (pp. 16–30). Berlin, Heidelberg: Springer-Verlag. Cited on pages 35 and 142.

Tian, X., Saito, H., Girkar, M., Preis, S., Kozhukhov, S., Cherkasov, A., Nelson, C., Panchenko, N., and Geva, R. (2012). Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International* (pp. 2349–2358). Cited on page 33.

Timnat, S., Shacham, O., and Zaks, A. (2014). Predicate Vectors If You Must. In *Proceedings of the 1st International Workshop on Programming*

Models for SIMD/Vector Processing, WPMVP '14. Cited on pages 37 and 139.

Touati, S.-A.-A. and Barthou, D. (2006). On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06 (pp. 147–156). New York, NY, USA: ACM. Cited on page 160.

Touati, S.-A.-A., Worms, J., and Briais, S. (2010). *The Speedup Test.* Technical report, INRIA Saclay - Ile de France. Cited on page 141.

Tripakis, S., Stergiou, C., and Lublinerman, R. (2010). *Checking Equivalence of SPMD Programs Using Non-Interference.* Technical Report UCB/EECS-2010-11, EECS Department, University of California, Berkeley. Cited on page 36.

Vázquez, F., Fernández, J. J., and Garzón, E. M. (2011). A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 23(8), 815–826. Cited on page 167.

Wald, I. (2011). Active Thread Compaction for GPU Path Tracing. In *Proceedings of High Performance Graphics 2011.* Cited on page 37.

Weispfenning, V. (1990). The Complexity of Almost Linear Diophantine Problems. *Journal of Symbolic Computation*, 10(5), 395–403. Cited on page 72.

Wolfe, M. J. (1995). *High Performance Compilers for Parallel Computing.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Cited on page 32.

Yang, Y., Xiang, P., Kong, J., Mantor, M., and Zhou, H. (2012). A Unified Optimizing Compiler Framework for Different GPGPU Architectures. *ACM Trans. Archit. Code Optim.*, 9(2), 9:1–9:33. Cited on page 32.

Yang, Y., Xiang, P., Kong, J., and Zhou, H. (2010). A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10 (pp. 86–97). New York, NY: ACM. Cited on page 36.