Vectorization for Accelerated Gather/Scatter and Multibyte Data Formats

by

Andrew Anderson

Dissertation

Submitted to the School of Computer Science and Statistics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Science)

School of Computer Science and Statistics

TRINITY COLLEGE DUBLIN

January 2016

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

.....

Andrew Anderson

Dated: January, 2016

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this Dissertation upon request.

.....

Andrew Anderson

Dated: January, 2016

Abstract

SIMD extensions to the instruction sets of general purpose processors have become widespread, and SIMD width and the capabilities provided by the hardware are steadily increasing with newer processor generations.

This thesis tackles two challenges faced by compilers when generating code for modern SIMD extensions: SIMD code generation for interleaved memory access patterns, and SIMD code generation for custom types not provided by hardware.

On the topic of SIMD code generation for interleaved memory access, we address strided array access specifically, and propose and evaluate a technique for the generation of SIMD code to gather and scatter data elements between memory and SIMD registers. Our technique extends a prior state of the art technique to support a wider class of interleaved memory access.

On the topic of SIMD code generation for custom datatypes, we propose and evaluate a vectorized code generation approach which supports reduced-precision floating point number formats along a continuum between native types.

We demonstrate that compilers can generate efficient SIMD code for both challenges using modern SIMD extensions, without requiring special hardware support beyond the general-purpose data movement and reorganization features already present in a variety of modern SIMD-enhanced general purpose processors.

Acknowledgements

Many thanks are due first to family and friends for their support and encouragement during my studies.

While working on my PhD I spent much time on several very fulfilling teaching positions, which I enjoyed greatly. Thanks are due in particular to David, Andrew, Jonathan, and Glenn for making this possible.

Thanks are especially due to my colleagues who made the lab both an entertaining and an intellectually stimulating environment in which to work: Servesh, Aravind, Martin, Mircea, Shixiong, and others.

Much of the research and investigation I carried out during the first two years of my PhD was in collaboration with Avinash, who was never hesitant to discuss ridiculous ideas which couldn't possibly be made to work.

Finally, it is my great pleasure to thank David, my supervisor, without whose guidance this thesis would not have been possible. David was always available to discuss hare-brained schemes and divine which components were actually useful and interesting, in addition to being a boundless source of advice and knowledge on many topics, both research related and not. Thanks to his patience and understanding, my journey to PhD was smoother than it had any right to be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SIMD (Single Instruction Multiple Data) execution is a mode of parallel execution where a processor executes a single instruction that operates on multiple values at once. These values are held in wide registers (often referred to as *vector registers*), with each value notionally occupying a specific *lane*, representing the location of the value within the register. Using SIMD instructions, all the values stored in a register can be operated on simultaneously. SIMD instructions are often referred to as *vector* instructions, and in this thesis, the two terms are used interchangably.

For example, on a 4-way SIMD machine, a single SIMD add instruction will simultaneously perform four additions, adding together pairs of values which occupy the same lane in their respective registers to produce four results. SIMD execution is an efficient way to exploit fine-grained parallelism at the instruction level to speed up the execution of programs. The degree of parallelism exploited by a program using vector instructions is known as the *vectorization factor*, written `VF` for short.

Both x86 and ARM [66] processor families provide extensions with a set of instructions supporting short vector parallelism. Intel's **S**treaming **S**IMD **E**xtensions (SSE) for x86 and the ARM NEON instruction set both operate on 128-bit vector registers which can be treated as a number of 8-, 16-, 32-, or 64-bit *lanes*. Intel's **A**dvanced **V**ector e**X**tensions (AVX) introduced 256-bit vector registers [25], doubling the potential vectorization factor which can be achieved

over SSE. Intel's forthcoming AVX-512 extensions will support even longer 512-bit vector registers.

1.1 Challenges

This thesis tackles two challenges faced by compilers when generating code for modern SIMD extensions. First, in Chapter 3, we address the problem of SIMD code generation for interleaved memory access patterns with strides other than powers of two. Second, in Chapter 4, we address the problem of SIMD code generation for datatypes whose width is not a power of two. Chapter 2 explores some motivation and background for supporting these memory access patterns in vectorized code.

1.2 Thesis Structure

The remainder of this thesis is structured as follows.

In Chapter 2, we present an overview of the capabilities of present-day SIMD hardware, and discuss the area of automatic vectorization in compilers. We are particularly interested in the generation of code for vector memory access. We also explore the relationship between SIMD and approximate computing, with a particular focus on memory access in variable-accuracy algorithms.

In Chapter 3, we propose a new approach to automatic vectorization for non-consecutive memory access patterns. Our approach extends the state of the art, and we achieve significant performance gains on two widespread modern SIMD architectures.

In Chapter 4, we propose a novel, vectorized approach to reduced-precision representation of floating point data. We propose a set of low-precision storage formats, and show that they can be implemented using current SIMD extensions, without requiring specific hardware support. Several formats do not correspond to any available machine type, but we demonstrate that they can be manipulated very efficiently in a vectorized fashion. In our experimental eval-

uation, we find that these formats can be supported with little to no overhead compared to native formats, and in some cases can even result in performance gain. We conclude in Chapter 5, discuss our findings at a more general level, and present some interesting directions for future work.

1.3 Contributions

The principal contributions of this thesis are as follows. We propose a general scheme for the generation of vectorized code for a class of non-contiguous data access which we term *Static Affine Interleaving*. This class of non-contiguous data access encompasses all dense array accesses which have a compile-time constant stride and offset. The state-of-the-art prior work of Nuzman et al. developed a method for the vectorization of the subset of accesses where the stride of access is a power of two [53]. We extend and generalize their approach to support the case of arbitrary compile-time constant strides.

We demonstrate that significant speedup can be obtained from *automatic* vectorization of interleaved accesses at a variety of strides, many of which have previously been considered to require irregular or hand-tailored solutions.

We also propose a scheme for the vectorized implementation of customized multibyte data formats for storage of results with reduced accuracy. Our scheme performs computation using native data formats, and inserts format conversion operations during loading and storing of data. Recent work in the area of reduced-precision representations [33] identified a low-level vectorized approach as likely to have significant value.

We show that efficient vectorized code can be generated to accelerate the conversion between natively supported datatypes and custom multibyte data formats which lack native support. Although conversion of data between formats has significant overheads, we demonstrate that a vectorized implementation of reduced-precision floating point storage formats can achieve speedup even at small scales well outside the supercomputing context where previous results were demonstrated.

1.4 Collaborations and Published Work

A modified version of Chapter 3 has been accepted for publication in ACM Transactions on Architecture and Code Optimization.

Andrew Anderson, Avinash Malik, and David Gregg. "Automatic Vectorization of Interleaved Data Revisited". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.4 (2015), p. 50

This work arose out of a collaboration with Dr. Avinash Malik on a domain-specific language and compiler designed to support image processing applications. As part of that work, we realized that there was no all-purpose solution to the problem of generating vectorized code for strided memory access. Avinash provided many helpful comments and lent his expertise in analysis and program transformation to initial discussions of the approach. In addition, he incorporated an early implementation of the techniques into a compiler he was developing, and provided feedback on their operation. We also collaborated on the text of the paper accepted by ACM TACO. However, the structure of the techniques and the associated optimizations presented in Chapter 3 are my own work.

Chapter 2

Background & Literature Review

Automatic vectorization is a family of compiler techniques within the more general field of automatic parallelization. An auto-vectorizing compiler attempts to identify statements or operations in a program which can be executed in parallel using SIMD instructions, and then to transform the program so that those statements or operations are actually performed in parallel using SIMD hardware.

Automatic vectorization in one form or another has been incorporated in compilers for many years, beginning with compilers for vector supercomputers [3, 4, 90]. Starting with the Hewlett-Packard PA-RISC processor family [46], and Sun Microsystems UltraSPARC [36], commodity computer hardware began to include some SIMD capabilities, by extending existing instruction sets with SIMD instructions. Wider availability of SIMD capabilities in commodity hardware [60, 61] has made automatic vectorization an important concern for optimizing compilers.

Many of the challenges of generating vector code stem from the design and implementation of SIMD extensions in general-purpose processors. With this in mind, we have structured this chapter as follows. We first review some literature describing important features and limitations of SIMD hardware, and the context in which SIMD extensions were developed. We then discuss a selection of key publications on automatic vectorization techniques, which have influ-

enced the work presented in this thesis. Finally, we summarize and discuss the specific challenges tackled by this thesis in the context of prior work.

2.1 Review of Features of SIMD Hardware

Many microarchitectures in widespread use today provide short vector instructions as an extension of the scalar instruction set to support applications with fine-grained data parallelism [25]. Typically, short vector extensions are implemented in separate functional units, with their own pipeline and register files. They provide parallel versions of scalar arithmetic, data movement, and miscellaneous other functions.

SIMD extensions to the instruction sets of general purpose processors (GPPs) were first developed in response to the evolving need for acceleration of rich multimedia applications, which incorporated computationally heavy tasks such as audio and video decoding [46] and computer vision [22]. As general purpose processors have become faster and more capable, new application areas which benefit from acceleration with SIMD continue to develop. A good example of such an area is computational photography [65].

Diefendorff and Dubey give an early overview of the challenges posed by these multimedia applications, and describe how they were expected to influence processor design [23]. Though almost two decades have passed since the publication of their review, their predictions have largely been borne out. Multimedia applications remain widespread, and general purpose processors have had more and more SIMD capabilities included with each processor generation. Many of the challenges they discuss, such as data reorganization, still pose problems for vectorizing compilers today.

More recently, Talla et al. discuss the design of SIMD extensions and architectural enhancements [79]. They discuss the overheads associated with the implementation SIMD extensions in general purpose processors, and investigate the causes of bottlenecks in vectorized programs. They find that memory

access patterns in programs are one of the primary impediments to efficient utilization of SIMD hardware. From Talla et al.:

> [...] even though GPPs are enhanced with SIMD extensions to extract DLP in multimedia programs, there is a mismatch between the requirements of media applications (for address generation and nested loops) and the ability of GPPs with SIMD extensions.

2.1.1 SIMD-Enhanced GPPs versus Traditional Vector Machines

Arguably the most important practical difference between SIMD-enhanced general purpose processors and traditional vector machines is in the functionality provided by the memory subsystem.

Traditional vector processors provided rich data access instructions for gather, scatter, and strided memory operations. However, modern SIMD extensions typically provide very poor support for non-consecutive memory access. SIMD instructions on GPPs work best on data that is contiguous in memory, and operating on non-contiguous data requires the generation of additional instructions to perform *data layout transformation* — packing and unpacking data elements between memory and vector registers.

However, real-world programs often manipulate data stored non-consecutively in memory, and have nested loops [79]. To parallelize these programs using SIMD extensions, the compiler must be able to transform the program to map these structures on to the restricted capabilities provided by the hardware. Nuzman and Henderson describe the challenges of multi-platform automatic vectorization in the GCC compiler [52].

Additionally, the length of vector registers in a general purpose processor is typically fixed at a short size. For example, the size of a vector register is 128 bits on Intel SSE and ARM NEON.

2.1.2 SIMD versus Out-of-Order Superscalar Execution

Out-of-order superscalar processors dynamically exploit parallelism in the program while it is executing, by finding independent instructions in the program and executing them in parallel. The parallelization is accomplished entirely in hardware, using complex logic which inspects instructions which are about to be executed, and finds those which can execute in parallel, using the different functional units within the processor. This is in contrast to SIMD execution, where the compiler is responsible for transforming the input program to extract parallelism by mapping multiple scalar operations to a single SIMD operation.

Lee and DeVries, and later, Lee and Stoodley, investigated the merits of SIMD parallel execution compared to out-of-order superscalar execution [44, 45]. As the degree of parallelism present in a program increases, they found that SIMD execution is able to exploit the parallelism more efficiently than out-of-order superscalar execution, while not requiring any more silicon area. From Lee and DeVries [[44]]:

> There are two reasons why vector processors are no more area-intensive than superscalar processors. One is that partitioning the register file into vector registers and lanes provides tremendous bandwidth without incurring a large area penalty. The second reason is that the control logic for issuing vector instructions is only slightly more complex than the control logic for scalar issue. In contrast, instruction control in superscalar processors is significantly more complex and thus requires substantially more area to implement.

Kozyrakis and Patterson additionally compare a SIMD architecture with both superscalar and VLIW architectures [37]. They find that for multimedia applications on embedded systems, a SIMD architecture has advantages in performance and power consumption, and has reduced design complexity versus superscalar and VLIW machines. They also found that applications parallelized with SIMD instructions typically have reduced code size compared to superscalar and VLIW parallelism.

2.2 Automatic Vectorization on General Purpose Processors

While a fully comprehensive overview of published work on SIMD is impossible, several review works have been published concerning automatic vectorization using short vector extensions specifically. In an early work,

Cheong and Lam describe the design and implementation of an optimizer which targets the UltraSPARC VIS instruction set [20], and provide a general review of the challenges faced by automatic vectorization.

Krall and Lelait give a high-level overview of compilation techniques for so-called *multimedia processors* (processors incorporating SIMD extensions for the purpose of accelerating multimedia applications) [38].

Ren et al. also discuss the vectorization of applications using SIMD extensions targeted at multimedia applications [67]. Many techniques applied to vectorize programs using short vector extensions were initially developed for traditional long-vector machines [4].

2.2.1 Capabilities of Vectorizing Compilers

Maleki et al. describe an evaluation of production compilers which perform automatic vectorization, aiming to investigate what sorts of programs are difficult for compilers to automatically vectorize, and what the reasons for these difficulties are [49].

Their evaluation covers the open-source GCC compiler, as well as Intel's ICC, and IBM's XLC. They provide a thorough review of the issues that limit automatic vectorization by these compilers in a range of programs drawn from different benchmark suites. A key issue is that of memory access patterns, in particular, programs with non-unit stride memory access.

From Maleki et al.:

> Our results show that after 40 years of studies in auto vectorization, today's compilers can at most vectorize 45-71% of the loops in our

synthetic benchmark and only 18-30% in our collection of applications.

Bik et al. provide a detailed overview of the automatic vectorization methods used in the Intel C++/Fortran compiler [12]. They also note that vectorization of non-unit-stride memory references poses significant challenges. According to Bik et al. elaborate instruction sequences may be used to implement non-unit stride memory references, but this method usually does not yield much speedup.

Two reviews have been published which specifically discuss the automatic vectorization capabilities of the GCC compiler: first in 2004 [51] and again in 2006 [54]. Again, non-consecutive memory access patterns are noted as a concern. The initial review notes that non-unit stride memory references incur considerable overhead, and, because of this, the access patterns supported by the vectorizer are limited to consecutive access only. The later review was performed just after the publication of a technique by the same authors which generates efficient code for power-of-two strided access [53]. This technique has remained the state-of-the-art for many years, and at the time of writing is the default method of vectorization for non-consecutive memory accesses in GCC. The work of Nuzman et al. is particularly relevant to this thesis, and is discussed in detail in Chapter 3.

2.2.2 Trends in SIMD Width and Memory Access

There is a global trend towards increasing SIMD width on general purpose processors, exemplified by the introduction of progressively wider vector units in successive generations of processors in the x86 family [25].

Schaub et al. recently investigated the impact of increasing SIMD width on control flow and memory divergence in vectorized code [72]. They find that as SIMD width increases, across a wide range of representative benchmark programs, memory access becomes less uniform and less consecutive. If vectorizing compilers are to be able to exploit the trend in increasing SIMD width for

actual performance gain, one of the key challenges is efficient, flexible SIMD code generation for non-consecutive memory access.

2.2.3 The Problem of Non-Consecutive Memory Access

SIMD instructions on modern processors operate on data packed consecutively into vector registers, and typically only provide instructions to load and store data from consecutive memory locations. However, a common data access pattern is where data items that are processed in consecutive operations in a program are stored non-consecutively in memory. Strided array access is a common example of non-consecutive data access.

When automatic vectorization techniques group together operations which use data that is not consecutive in memory, and the processor does not support SIMD operation on non-consecutive data, the compiler must generate code to gather different data elements together into a vector register for reads, or scatter the contents of a vector register over memory, for writes. We refer to this as the problem of non-consecutive memory access.

Due to the difficulty of dealing with memory access in the generation of vectorized code, Chang and Sung propose augmenting processors with special hardware to support irregular, misaligned, and strided memory access [17]. However, until robust hardware support for non-consecutive access with SIMD extensions becomes widespread, the task of generating fast code for non-consecutive memory access falls to the compiler.

While some automatic vectorization techniques have been proposed which can deal with non-consecutive data access, they typically work by attempting to circumvent the lack of hardware support for non-consecutive data access. Some techniques attempt to reorganize the program so that access becomes consecutive [42], but this is not always possible.

Other techniques insert extra instructions in the program to rearrange data in memory so that the data elements operated on by SIMD instructions are stored consecutively, but this has significant overhead [48].

Because of the widespread lack of hardware gather/scatter support, many vectorization techniques operate with the assumption that all data access is consecutive. A broad selection of these techniques are discussed in Section 2.3. If a region of the program contains non-consecutive data access, these techniques are unable to vectorize that region. A paper at the 2004 GCC Developers Summit [51] describes the status of automatic vectorization in the open-source GCC compiler. At that time, the compiler would only vectorize regions with consecutive data access.

In 2006, Nuzman et al. described a technique by which the compiler could generate efficient SIMD instructions to perform strided array access, without requiring hardware support for gathering or scattering operations. The technique has been the state-of-the-art approach for loop vectorization, and is the default method for vectorization of strided array access in GCC at present. However, the technique can only generate SIMD code for power-of-two strided access [53].

2.3 Overview of Automatic Vectorization

Automatic vectorization relies on many pieces of compiler technology in addition to the techniques which actually perform the translation of programs to vector form. In this section, we review the literature on supporting technologies for vectorization. These include program analysis, theoretical models of computation, and code and data restructuring techniques. We also discuss some optimization challenges which are particularly important for vectorized programs, such as optimization of data permutations and locality of data access.

Over the years, many approaches to automatic vectorization have been proposed. So many, in fact, that recent work has applied machine learning techniques to the problem of simply *choosing* which vectorization techniques to apply to particular programs [78]. Broadly speaking, the approaches to auto-

matic vectorization which are implemented in compilers for general-purpose programming languages such as C and C++ belong to one of two families.

2.3.1 Loop Vectorization

One family of approaches deals with cyclic control structures which repeatedly iterate a set of operations. Multiple instances of the same operation are grouped together from successive iterations of the control structure to form a single SIMD operation. We refer to this family of approaches as *loop vectorization*.

Loop vectorization targets loops specifically, with the aim of performing multiple loop iterations in parallel by transforming the instructions in the loop into vectorized form. Early vectorizers focused exclusively on vectorizing loops [3], and the area is still the subject of much work.

For nested loops, the innermost loop is the usual target for vectorization, but vectorization of outer loops can sometimes yield better results [85]. The topic of vectorization of outer loops is revisited by Nuzman and Zaks for modern short vector extensions [55].

Karrenberg and Hack propose a holistic approach to vectorization of whole functions, encompassing control and data flow [35]. Their approach can be seen as an evolution of loop vectorization. However, their approach, like each of the others cited, assumes consecutive memory access.

2.3.2 Superword Level Parallelism

Another family of approaches deals with straight-line code, which performs a set of operations only once. Multiple *isomorphic* operations in a subgraph of the program, which perform the same computation but work on different data elements, are selected. These are grouped together into a single SIMD operation. We refer to this family of approaches as *superword level parallelism* or SLP.

Vectorization techniques based on SLP [42] incorporate a tactic which can handle a restricted case of non-consecutive memory access. SLP attempts to transform programs so that memory accesses *become* consecutive, for example, by searching for an unrolling factor for scalar loops which results in a dense memory access pattern in the vectorized loop. It is not always possible to transform the program in such a way.

For loops, for example, an unrolling factor that makes strided access consective can only be found if the scalar version of the loop touches every element of each accessed array region within `VF` loop iterations [42].

Park et al. [59] propose a "SIMD Defragmentation" approach which tries to extract parallelism at the level of subgraphs of operations within the program, similar to SLP. However, their approach does not address non-consecutive memory access beyond what was proposed in Larsen and Amarasinghe's original SLP paper.

Very recently, Porpodas et al. introduced a flexible version of SLP which can parallelize subgraphs which are not perfectly isomorphic by introducing extra operations to make them so [64]. However, like Park et al., they do not address non-consecutive memory access beyond what was proposed in the original SLP paper.

2.3.3 Analysis and Program Transformation

Automatic vectorization usually proceeds from the structure of data dependences within the program. A dependence analysis is used to detect operations or regions in the program which can run in parallel [57]. Many classical code optimization and reorganization techniques can applied before vectorization to make the program easier to vectorize [50, 5]

The polyhedral model is an important framework for this kind of analysis [21]. In the polyhedral model, data dependences between statments in loops are modeled as a dependence polyhedron, that is, by relations between sets of points in an integer vector space where each point represents a loop iteration (the iteration space). This representation is transformed and manipulated to

reorganize loops so that parallelism is exposed and locality is improved. Optimizers based on the polyhedral model are used in LLVM [30] and in GCC [63, 77, 81]. Vectorization can be represented in the polyhedral model, and Trifunovic et al. apply the model to generate vectorized code for loop nests [80]. However, the focus of their work is high-level optimization of the control structure, and the generation of code for the data access in the reorganized loops is not addressed.

2.3.4 Vectorization of Control Flow

Auto-vectorization approaches can also be applied to vectorize control flow, such as conditional branches generated by `if`-statements, within a program. Previous work extends SLP [75] to enable it to vectorize control flow by converting conditional expressions in the scalar code into predicated vector expressions (if-conversion).

Later work by Shin et al. introduced an approach which takes advantage of a particular type of SIMD instruction called BOSCC (Branch On Superword Condition Code) to represent control flow in vectorized programs [76].

2.3.5 Domain-Specific SIMD Languages and Compilers

A common tactic to ease the task of producing SIMD code is to use domain-specific languages which either provide explicit-SIMD programming constructs, or programming constructs that can be mapped to efficient SIMD code by the compiler.

SPL [89] is a domain-specific language for describing transformations used in digital signal processing. The associated compiler system, SPIRAL [27], generates fast SIMD code to implement the transformations. Research on the SPIRAL system has resulted in several advances in SIMD code generation, particularly in the area of SIMD permutations [28].

Some approaches propose to extend C and similar languages with explicit SIMD language features [15]. These include Intel's `ispc`, which is a language

and compiler system for SIMD [62] that also extends C with SIMD language features. The IVL language and compiler is another explicit-SIMD programming system based on `ispc` [47].

The Halide language [65] is a project that differs from other approaches by decoupling the specification of algorithms from their schedule. Halide allows programmers to explicitly request vectorized execution of regions of their program, in addition to other methods of parallelization.

All of these approaches aim to improve the performance of SIMD programs by giving the programmer more ways to structure their code to explictly take advantage of SIMD hardware.

2.4 Memory Access with Multimedia Extensions

There are several concerns specifically with memory access in programs which make use of multimedia extensions. In this section we review some approaches to concerns such as alignment, locality of access, and data layout transformation.

2.4.1 Spatial Locality of Access

Spatial locality of access is a major concern in optimized code generation, particularly on microarchitectures with an extensive cache. When an access causes data to be fetched into the cache, subsequent accesses very close to the first are likely to result in cache hits, speeding up the program by eliminating costly main memory accesses. To exploit this arrangement, compilers optimize for spatial locality by rearranging the program so that data frequently accessed together in the program are stored closer together in memory.

Shin et al. discuss optimizing for spatial locality in vectorized code specifically [73], and propose some techniques to optimize locality using vector registers [74]. The polyhedral model also deals very comprehensively with locality concerns at a more general loop level [14, 13].

2.4.2 Alignment

A common restriction of SIMD load and store instructions on modern processors is that access to data which is not aligned with vector registers is subject to a performance penalty compared to accessing aligned data.

Several works have addressed the issue of aligned access with vectorization. Eichenberger et al. and Wu et al. propose several heuristics which can deal with alignment issues while vectorizing, either statically [24] or at runtime [87]. Subsequently Fireman et al. present two algorithms which are optimal for special cases of the alignment problem [26]. However, all three assume that memory access, though it may be misaligned, is still consecutive.

2.5 Approximate Computing

Approximate computing is an emerging area of research [31]. The general idea is to design applications and hardware so that they can tolerate a loss of quality or accuracy in their results to improve performance or energy efficiency.

Some existing work has integrated approximate computing concepts into compilation systems. Petabricks is a language and compiler which supports variable accuracy algorithms [7]. Programmers provide multiple implementations of algorithms, and with the use of benchmarking and auto-tuning, the system selects the appropriate implementation for a new platform. Programmers can also provide accuracy constraints for the implementation of an algorithm using annotations, which the compiler can incorporate into the code generation process, producing faster but less accurate code. Green is another compiler system which supports approximate computing, which is targeted specifically at reducing the energy used by programs [9].

Chapter 4 introduces a method of generating low-level vectorized code to implement customized data storage formats, which are designed to represent reduced-accuracy results produced by approximate algorithms. The idea is to save space in memory (and reduce memory bandwidth) by using formats which have lower precision than natively supported types.

2.5.1 Floating Point Programs with Reduced Accuracy

Exploiting reduced accuracy requirements in floating point computations to accelerate applications is the topic of several related works. Some, like Buttari et al. [16], choose to reduce precision in increments of the available native types on the platform. In the case of Buttari et al., their approach performs computationally expensive portions of numerical algorithms using single precision arithmetic, and less expensive portions using double precision arithmetic. They note that the choice of single precision arithmetic leads to speedups of up to 2x versus double precision for applications which are memory bound, due to the 2x reduction in the amount of data transferred through the memory bus.

This approach is also used by Rubio-Gonzalez et al. [71] who present an automated system which finds a compile-time instantiation of the types of floating point variables in the program which improves performance. Their approach is subject to accuracy constraints which are specified by the programmer via annotations in the source code. Like Buttari et al. [16] they consider only the available native machine types.

Lam et al. [41] also pursue this approach, by using binary instrumentation and translation to modify existing binaries and automatically find a satisfactory mixed-precision version of a program. Input programs are constrained to use double precision arithmetic only, and the system searches for program variables which can have their precision lowered without adversely affecting the computed results. Again, only natively supported types are considered for replacement.

2.5.2 Customized Data Storage Formats

Programs which produce approximate results with reduced accuracy often do not require the full precision of native types to represent those results. In this situation, the use of full precision native formats can lead to inflated storage requirements and excess memory traffic. Using a specialized data storage format can reduce the amount of memory traffic created by the program, with associated gains in energy efficiency and overall runtime.

Using vectorization to support specialized storage formats is particularly attractive. Vectorized execution provides a low-cost means of accelerating data reorganization and conversion by exploiting fine-grained data parallelism across multiple accesses. Vectorized implementations of specialized storage formats can make use of special vector reorganization instructions provided by modern processors, which are not available for scalar registers. In addition, vectorized execution is already the norm for many numerical applications.

2.5.3 Multibyte Floating Point

Jenkins et al. [33], propose to accelerate I/O performance of applications by reducing the resolution of the data. However, although the goal of that work is similar, there are several important differences with the approach we propose.

In terms of practical differences, an important distinction between the two pieces of work is that our approach parallelizes at the level of loop iterations, using vector memory access and vector reorganization to achieve a parallel speedup. Jenkins et al. evaluate their low-resolution scheme at extreme scale using thread-level parallelism via MPI, and using MPI to perform the data layout transformations. They note that a low-level vectorized approach appears to be promising future work given that data reorganization using MPI incurs significant overhead. In addition, the work of Jenkins et al. only deals with reads, whereas we propose an end-to-end approach which can deal with writes of reduced precision data in addition to reads.

Another important refinement over the work of Jenkins et al. is the topic of *rounding*, which is not addressed by that work. Jenkins et al. use simple truncation to obtain low-resolution data. This approach causes a measure of avoidable error which can be reduced significantly by performing correct rounding to select the nearest representable value in the target format for a given input value. In Chapter 4 we present a scheme for supporting custom floating point storage formats with correct rounding of data.

2.6 Summary and Directions for Improvement

Chapters 3 and 4 propose methods which generate SIMD code to efficiently implement memory access patterns which lack good hardware support – either by virtue of accessing data non-consecutively, or by accessing data stored in a format which does not correspond to native machine types.

In Chapter 3 we propose a generalization of the technique of Nuzman et al. [53] which removes the power-of-two constraint allowing vectorized code to be generated for a gathering or scattering operation with any compile-time constant stride.

Since difficult memory access patterns are often an impediment to vectorization [49], there appears to be significant value in a more general approach which can effectively vectorize a wider class of non-consecutive data access. Direct performance improvements aside, such an approach could act as an enabling technique for many of the vectorization tactics reviewed, allowing them to be applied to a wider range of programs by providing a way to generate SIMD code for more cases of non-consecutive memory access.

Previous work [48] has demonstrated some performance gains from transforming the layout of data in memory with scalar code just before the execution of vectorized code which accesses that data. However, being able to synthesize vectorized code to perform interleaved reads and writes directly in a vectorized loop or basic block means that instruction-level parallelism between data movement and computation can offset the overhead of memory access. This on-the-fly transformation of data layouts is the recommended mode of operation to achieve good performance using Intel's SIMD extensions [1].

In addition to our work on non-consecutive memory access, in Chapter 4, we propose a novel, low-level vectorized approach to supporting custom data formats. This is intended as a supporting technique for the area of approximate computing, reviewed in Section 2.5. We aim to reduce storage requirements and improve memory transfer times for programs with reduced-accuracy results.

Chapter 3

Vectorization of Static Affine Interleaving

Automatically exploiting short vector instructions sets is a critically important task for optimizing compilers. Vectorizing compilers identify program regions with fine-grained data parallelism and attempt to exploit this parallelism by transforming the program to use vectorized execution. Automatic vectorization is employed by many compilers, including those from the GNU Compiler Collection, LLVM, ICC, and more. Vectorized execution offers parallel speedups without the complexity of executing multiple independent instructions in parallel, and is therefore usually efficient in the use of hardware resources and energy.

However, automatic vectorization presents many problems in dependence analysis, code restructuring and data access patterns [8, 86]. A common data access pattern is where data items that are processed in different ways are interleaved together in memory. Unlike traditional vector processors which provided rich data access instructions for gather, scatter, and strided memory operations, modern short vector extensions typically provide very poor support for non-consecutive memory access. Short vector instructions work best on data that is contiguous in memory and operating on non-contiguous data requires the generation of additional instructions to perform *data layout transformation* — packing and unpacking data elements between memory and vector registers.

The need to reorganize data in vector registers has led to the inclusion of many specialized instructions for data layout transformation in vector instruction sets. For example, processors supporting Intel's SSE instruction set have the special `shufps` instruction, which selects even or odd indexed lanes (depending on the instruction mask) from two registers, and combines them in one register. ARM's NEON instruction set contains a similar `vld2` instruction, as does the Altivec instruction set (`vperm`). Such instructions typically include permutation of vector registers where the order of the items in the lanes of the register is changed, and combination of registers where a selection of lanes from different register are interleaved together. The presence of these instructions means that vectorization is well positioned as a supporting technique for other approaches which need to perform data layout transformations.

3.1 Modelling Arrays and Array Access

A dense array can be modelled as an indexed set of elements, where any two consecutive indices specify consecutive array elements. Similarly, an iterated access to such an array can be modelled as an indexed set, where any two consecutive indices specify array elements accessed by consecutive *loop iterations*. However, consecutive elements of the access are consecutive in the temporal sense, and are not required to be stored consecutively in the underlying array. When this is the case, we say that the access is non-contiguous. When multiple non-contiguous accesses share an underlying array, we say that the access pattern (i.e. the union of all accessed elements) is *interleaved*.

The relationship between *arrays* and *accesses* is captured by the *access function* for each access. The access function translates a loop iteration variable (which indexes elements of an access) so that it selects elements of an array. The access function can be any function of the loop iteration variable and potentially many more parameters. Many audio and video processing applications have access functions which are nonlinear and are implemented using lookup tables, and implementations of signal processing algorithms such as

the fast Fourier transform often use exotic nonlinear access functions, such as bit-reversed ordering. Implementations of file systems, databases, and compression algorithms frequently use space-filling curves as access functions.

In this work we focus on the class of access functions which result in *Static Affine Interleaving*. This class encompasses all functions which are strictly affine transformations of the loop induction variable with respect to the arrays being accessed, where the values of all other parameters is statically known. Such functions have the following general form.

Definition 1 (Access) *We represent a strided access with iteration variable i as a function a of the form*

$$a(i) = b + u * (\texttt{stride} * i + \texttt{offset})$$

where b is the base address of the array, and u the unit size in bytes of an array element. The access is consecutive when $|\texttt{stride}|$ is 1, and nonconsecutive otherwise. Note that the sign of the stride or offset may be negative.

For example, if the same array element is accessed in every loop iteration, `stride` may be instantiated to zero, and `offset` to the index of the accessed element in the underlying array. To visit every array element in order, `stride` may be instantiated to 1. To visit the elements of the array in reversed order, `stride` may be instantiated to -1, and `offset` to the length of the array, and so on.

Our approach to vectorizing such accesses follows a simple, general scheme: we cover the memory range accessed with non-overlapping vector loads or stores, and map individual accessed elements to loaded or stored lanes. The problem of vectorizing a strided access then becomes the problem of composing an ordered subset of loaded or stored lanes to or from a single vector register. In this chapter, we develop our approach by first showing how to create a vectorized code sequence in a simple, canonical form, and by application of successive optimizations, refine it into the final form which will be emitted.

While our approach does not require aligned access, it may be desirable for performance reasons. Any implementation may apply a wide variety of possible techniques to ensure aligned access [24, 26]. One approach is to load extra lanes and discard those unused, treating the vector register file as a compiler-controlled cache [73]. However, this tactic has some corner cases: when the base address of the array is misaligned or the array does not contain enough data, an implementation using this tactic may have to apply array padding, or rely on masked vector loads or stores. Similarly, when loop trip counts are not a multiple of the vectorization factor (`VF`), extra iterations may need to be peeled and performed as scalar iterations.

3.1.1 Notation and Presentation

We specify our code generation in terms of a simple abstract instruction set. Similar to Wu et al., we use a representation with *virtual* vectors, and rely on a later native code generation phase to map these to the actual hardware [88].There are several motivations for this choice of instructions. Many architectures provide a large number of distinct specialized data reorganization instructions. Were we to state our code generation in terms of specialized instructions, it would restrict the applicability of our techniques to architectures with support for those instructions. The choice of a simple, generic form of permute and blend instructions ensures our approach is more widely applicable.

Second, the program transformations we propose in this chapter are quite succinct when expressed in terms of these simple instructions. Including many specialized data reorganization instructions would significantly complicate the presentation of our techniques.

Finally, it is important to note that on architectures which do have highly specialized data reorganization instructions, they do not go unused. Many multimedia architectures such as Intel SSE, AVX, AVX2, and ARM NEON provide such instructions in addition to generic instructions corresponding to those we include. However, many of the highly specialized native instructions for data reorganization can be expressed in terms of a short sequence of more

generic permutes and blends. In this scenario, traditional tree-parsing instruction selection techniques [2] are very effective at selecting highly-specialized native instructions to cover the sequences of simplified operations which we generate. We detail our approach to native code generation in Section 3.4.6.

Instruction	Arguments
`load`	(`Vector target`, `Pointer source`)
`store`	(`Vector source`, `Pointer target`)
`permute`	(`Vector source`, `Vector mask`, `Vector target`)
`blend`	(`Vector left`, `Vector right`, `Vector mask`, `Vector target`)

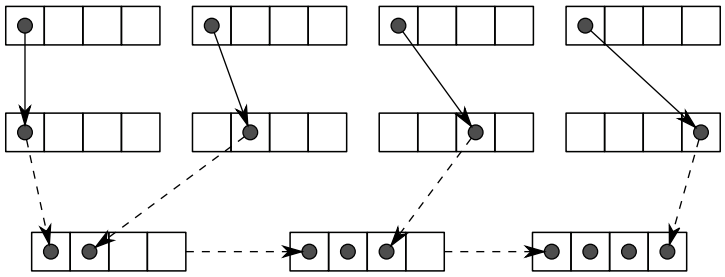
Figure 3-1: The target intermediate representation for code generation.

We assume the following informal semantics for the instructions in Figure 3-1. `load` and `store` are packed vector load and store instructions. `permute` and `blend` are masked permutation and blending instructions. `permute` moves the ith lane of the source to the lane of the target given in the ith lane of the mask. `blend` selects the ith lane of the target from the left source if the ith mask lane is `L`, or from the right source if it is `R`.

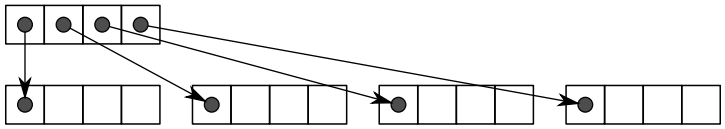
Presentation

We write mask literals as a list of lane values enclosed in angled brackets. Vectors described by these masks have a common element size and type. Lanes containing the special $*$ mask element indicate a don't-care output in the lane. For example, the instruction `permute a`, $\langle *, 0, *, 1 \rangle$, `b` moves lanes 0 and 1 of vector `a` to lanes 1 and 3 of vector `b`, and leaves lanes 0 and 2 in an undefined state. In addition to the $*$ element, masks for the `blend` instruction may contain only the two special values `L` and `R`, indicating left and right source register, respectively. In graphical figures where data movement is indicated with arrows, an arrow with a solid line represents data movement using the `permute` instruction, and an arrow with a dashed line represents data movement using the `blend` instruction.

3.2 Technique



(a) Algorithm 1. Nonconsecutive read of the form `a[4 * i]` with `VF = 4`. First we `permute` the load-mapped register set so that selected lanes will not collide under vertical composition with `blend`. We then `blend` the permuted registers together to form the packed vector corresponding to the access.



(b) Algorithm 2. Nonconsecutive write of the form `a[4 * i]` with `VF = 4`. We expand the register to be stored into a register set which shadows the array region to be written. This store-mapped register set is then written over the shadowed array region with predicated writes, or using a read-modify-write sequence.

Figure 3-2: Vectorizing interleaved access using mapped register sets.

3.2.1 Enabling Interleaved Access: Automatically Vectorizing a Single Strided Access

We vectorize a nonconsecutive read by first mapping the memory range accessed end-to-end into vector registers (the load-mapped register set). By permuting and blending this register set together, we can extract any subset of up to `VF` lanes into a single packed vector register. We vectorize nonconsecutive writes similarly, by first mapping a store-mapped register set to the memory region being written. A non-consecutive write of data in a packed vector register is performed by expanding the register into a register set with items at the correct locations, and then combining this register set into the store-mapped register set using the `blend` instruction. It is important to stress that these registers are only logically mapped to memory — a mapped register will result in

the generation of a memory operation only if one or more lanes in the register are active.

Any vectorized strided access may touch elements in the range of (`stride` $*$ `VF`) consecutive memory locations in one memory operation. Since this memory region is mapped by registers of length `VF` elements, it follows that a maximum of `stride` mapped registers are required for a single vectorized access. Figures 3-2a and 3-2b show graphically the action of this simple canonical technique, and Algorithms 1 and 2 contain the logic required to generate the depicted instruction sequences.

Our approach has two phases: one phase permutes the mapped register set to eliminate lane collision when interleaving or deinterleaving, and the other phase takes a collision-free register set and combines registers using the `blend` instruction to form a packed result. To see why lane collision is a problem, consider Figure 3-2a: we cannot directly `blend` the initial mapped registers together, because multiple elements occupy the same lane in their respective registers, and collide when using the `blend` instruction.

ALGORITHM 1: Generate code to vectorize a read access (canonical)

Input: A strided access `a`, load-mapped register set `p`
Output: `s`$[$`VF` $-$ `1`$]$ contains `VF` packed consecutive elements of `a`
$s \leftarrow$ allocate `VF` temporary registers;
for $i = 0$ *to* `VF` $- 1$ **do**
    `mask` $\leftarrow \langle$`VF` $\times *\rangle$;
    `mask`$[$`i`$] \leftarrow$ (`stride` $* i$) $+$ `offset`) $\mod$ `VF`;
    `r` $\leftarrow$ `p`$[$(`stride` $* i$) $+$ `offset`$)/$`VF`$]$;
    generate: `permute r, mask, s`$[$`i`$]$**;**
end
for $j = 1$ *to* `VF` $- 1$ **do**
    `left` $\leftarrow j$, `right` $\leftarrow$ `VF` $- j$;
    `mask` $\leftarrow \langle$(`left` $\times$ `L`) $+$ (`right` $\times$ `R`)$\rangle$;
    generate: `blend s`$[$`j` $- 1]$, `s`$[$`j`$]$, `mask`, `s`$[$`j`$]$**;**
end

ALGORITHM 2: Generate code to vectorize a write access (canonical)

Input: VF consecutive elements of a strided access a packed in register r
Output: Store-mapped register set p contains the VF elements of a in store order

```
pmask ← ⟨VF × ∗⟩;
bmask ← ⟨VF × L⟩;
pmasks ← stride copies of pmask;
bmasks ← stride copies of bmask;
s ← allocate stride temporary registers;
for i = 0 to VF − 1 do
    register ← (stride ∗ i)/VF;
    lane ← (stride ∗ i) mod VF;
    pmasks[register][lane] ← i;
    bmasks[register][lane] ← R;
    generate: permute r, pmasks[register], s[register];
    generate: blend p[register], s[register], bmasks[register], p[register];
end
```

3.2.2 Exploiting Spatial Locality: Grouping Multiple Interleaved Accesses

Multiple accesses to the same source or destination array can require overlapping vector loads or stores in a vectorized loop iteration if they share the same stride of access (shared-stride accesses). When this is the case, the accesses often exhibit spatial locality which can be exploited to reduce the number of memory operations in the vectorized program. Using the simple canonical approach from Section 3.2.1, we might generate loads and stores of the same data more than once. Similar to Nuzman et al. [53], we exploit this spatial locality by allowing multiple accesses to share mapped register sets when interleaving/deinterleaving, reducing the number of memory operations in the vectorized loop.

The degree of spatial locality present between any two shared-stride read or write accesses to the same array in vectorized loop iteration $(i/\texttt{VF})$ depends on the distance between accessed array elements in scalar loop iteration i, that is, the absolute difference between the `offset` of the two access functions. For any two shared-stride accesses with distinct offsets, three scenarios are possible.

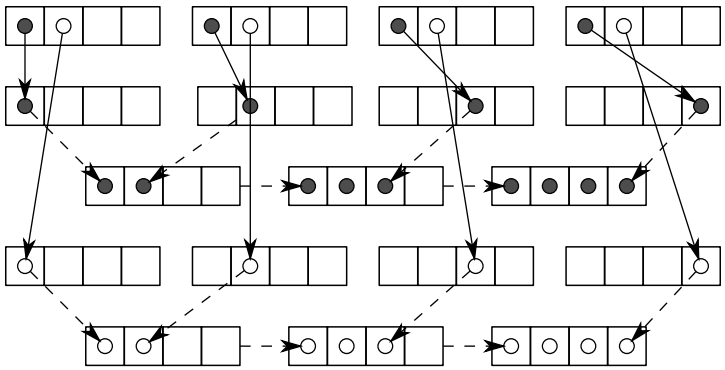
(1) *No locality* — When the distance between offsets is greater than or equal to $(\texttt{stride} * \texttt{VF})$, the accesses do not overlap vector loads or stores in a vectorized loop iteration.

(2) *Partial locality* — When the distance is strictly less than $(\texttt{stride} * \texttt{VF})$ and greater than or equal to `stride`, there is partial reuse — some elements of the first access will map to the same loaded or stored registers as elements of the second.

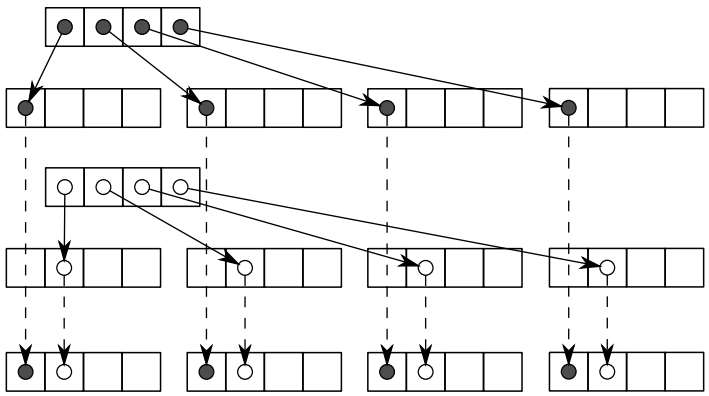
(3) *Full locality* — When the distance is strictly less than `stride`, there is full reuse in a vectorized loop iteration — all `VF` elements of each access map to the same set of vector loads or stores.

Note that our definition does not take into account temporal locality found along the backedge of the vectorized loop. Rather, we focus exclusively on exploiting spatial locality within `VF` iterations of the original loop.
Consider the pair of shared-stride memory accesses $\texttt{a}[4 * \texttt{i}]$ and $\texttt{a}[4 * \texttt{i} + 1]$, from Figure 3-3a or 3-3b. This pair of accesses require the same vector memory operations in every vectorized loop iteration — the grouping exhibits *full locality* as we have defined it. However, when we include an access $\texttt{a}[4 * \texttt{i} + 5]$, it will require an extra memory operation in every vectorized loop iteration, because the underlying memory regions do not completely overlap (*partial locality*). Including another access, $\texttt{a}[4 * \texttt{i} + 7]$, that access would exhibit full locality with access $\texttt{a}[4 * \texttt{i} + 5]$, but only partial locality with accesses $\texttt{a}[4 * \texttt{i}]$ and $\texttt{a}[4 * \texttt{i} + 1]$.

In order to minimize the number of vector memory operations for any number of shared-stride accesses, it is sufficient to consider only groups of maximal size, and load or store each resulting mapped register exactly once. However, minimizing the number of memory operations does not guarantee the generation of optimal vectorized code for a loop containing such accesses, which is a difficult optimization problem. Considerations include not only repeated memory operations, but also vector register spills and reloads due to register pressure and the scheduling of the instructions which perform the interleav-



(a) Algorithm 1. Two nonconsecutive reads of the form $\texttt{a}[4 * \texttt{i}]$ (dark dots) and $\texttt{a}[4 * \texttt{i} + 1]$ (light dots) with $\texttt{VF} = 4$, with shared spatial locality can be serviced from the same load-mapped register set using `permute` and `blend` sequences.



(b) Algorithm 2. Two nonconsecutive writes of the form $\texttt{a}[4 * \texttt{i}]$ (dark dots) and $\texttt{a}[4 * \texttt{i} + 1]$ (light dots) with $\texttt{VF} = 4$ are composed into a single store-mapped register set, reducing the number of stores required.

Figure 3-3: Sharing load-mapped/store-mapped register sets to exploit spatial locality.

ing/deinterleaving, which can exhibit significant instruction-level parallelism with the computation present in the loop. Barik et al. [11] demonstrate that good solutions to such problems require tight integration between vectorization, register allocation, and instruction scheduling during compilation.

We do not attempt to solve this optimization problem, but use the same simple, practical grouping heuristic for accesses with locality as Nuzman et al. [53] — group only those accesses which exhibit *full locality* as we have defined it. This approach yields significant speedup in practice, and does not require much engineering effort on the part of the compiler implementor.

In order to decide at compile time which shared-stride accesses can be serviced from a shared mapped register set, we introduce the analytic concept of an access group, which generalizes the similar concept of Nuzman et al. [53].

Definition 2 (Access Group) *Accesses to the same array with the same direction (read or write) may be grouped by mapping each access offset to some interval* $[k, k + (\texttt{stride} - 1)]$*, for* $k \in \mathbb{N}$*, where* $k \equiv 0 \mod \texttt{stride}$*. Each such interval, for any particular stride, defines a distinct access group at that stride. The size of an access group (written* n*) is bounded above by the shared stride of access of the group, and below by zero.*

Let us assume we are considering two accesses $a_0(i) = b_0 + u_0 * (\texttt{stride}_0 * i + \texttt{offset}_0)$ and $a_1(i) = b_1 + u_1 * (\texttt{stride}_1 * i + \texttt{offset}_1)$. These accesses can share the same load or store mapped register set iff

$$\texttt{stride}_0 = \texttt{stride}_1$$

$$u_0 = u_1$$

$$\lfloor (b_0 + u_0 * \texttt{offset}_0)/(u_0 * \texttt{stride}_0) \rfloor = \lfloor (b_1 + u_1 * \texttt{offset}_1)/(u_1 * \texttt{stride}_1) \rfloor$$

This formulation groups accesses where stride of access and unit size are equal, and the accesses are relatively aligned within `stride` elements of the shared unit size of access. These criteria are sufficient to ensure that grouped accesses exhibit *full locality*. To vectorize such an access group, we repeatedly apply

Algorithm 1 or 2 but use only a single, shared mapped register set. Composing reads and writes into a shared mapped register set reduces the number of memory operations required for any group of n accesses. Since n is bounded above by the stride of access, and the number of mapped registers is exactly equal to the stride of access (Section 3.3.2), this sharing represents a reduction from worst-case $O(n^2)$ memory operations considering individual accesses to $O(n)$ operations considering the access group.

3.2.3 Dealing with Store-Side Gaps

The work of Nuzman et al. [53] specifically excludes interleaved access patterns with store-side gaps. A *gap* is any unread or unwritten area of memory between elements of an interleaved access. Figure 3-3 displays two scenarios with gaps. In both examples in the figure, only two of the four lanes in each loaded or stored vector are used. While unused loaded lanes can simply be discarded, unused lanes in stores require the implementation to preserve the contents of memory in those lanes. As indicated in Figure 3-2, this may be achieved using predicated writes, or using a read-modify-write sequence. On both of our experimental platforms, predicated writes have very poor performance, while read-modify-write has excellent performance.

While predicated writes typically have the same semantics as the original scalar writes, any implementation using read-modify-write sequences may encounter race conditions due to the fact that a read-modify-write sequence modifies memory elements which were not modified by the scalar code. If the contents of memory corresponding to unused lanes changes between the read and write step of a sequence, data races and memory corruption can result. Broadly, there are three scenarios for such races: races between accesses in the same access group, races between different access groups, and thread-level races between multiple instances of the vectorized code operating on the same memory region.

For the class of (static-affine) accesses we consider, the technique of Chatarasi et al. [18] can be used to statically detect data races. The compiler may only

correctly parallelize statements where doing so would not create a data race. Our approach avoids the creation of races between accesses in the same access group by composing all the resulting memory reads or writes into a single mapped register set. Effectively, our technique coalesces vector memory operations in the compiler. Since our approach produces only one sequence of non-overlapping memory operations, we avoid introducing races between the accesses in any one access group.

To avoid inter-access group races, implementations which cannot use predicated writes may place read-modify-write sequences resulting from different access groups into atomic sections or use memory fences to ensure exclusive access to contested memory regions. The same approach may be used to avoid thread-level races. Where the hardware does not provide a way to ensure atomic execution of a group of instructions, or to create memory barriers, read-modify-write cannot always be used safely in a multithreaded context.

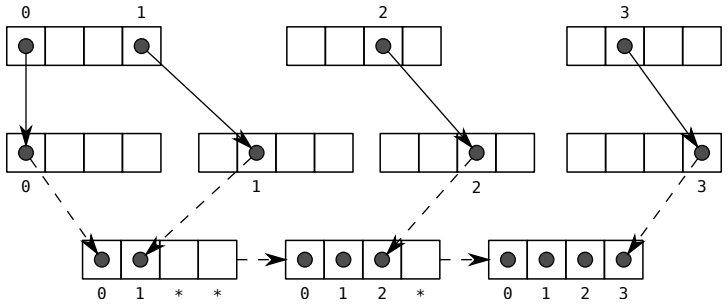
3.3 Optimization

There is significant scope for optimization of the instruction sequences generated by the approach outlined in Section 3.2.2. In this section, we present four optimizations which transform the `permute/blend` sequence programs generated by our technique.

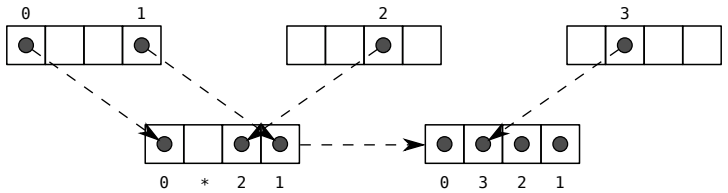
These optimizations can be broken down into two categories: either reducing the number of permutations in a program, or reducing the number of blends. The optimizations concerning permutation follow from the realization that we typically permute a register for one of two reasons: either we need to eliminate a lane collision for blending, or we need to enforce matching element order in two vector registers to ensure a legal vectorization. In both of these scenarios, we can eliminate permute instructions under some conditions. Concerning blends, we state a property of blend instructions assuming the informal semantics in Section 3.2 which allows us to merge multiple blend instructions into a single blend instruction. We also introduce a reassociation of blend in-

struction sequences to increase the count of mergeable instructions under this property.

3.3.1 Eliminating Permutations I: Executing Original Loop Iterations Out-Of-Order



(a) Algorithm 1. Memory access of the form $\texttt{a}[3 * \texttt{i}]$ with $\texttt{VF} = 4$. $*$ denotes a don't-care element in an output register. Elements are labelled with their index in original program order.



(b) Algorithm 3. Performing the deinterleaving step without first permuting the load-mapped register set to eliminate lane collision under vertical composition with the `blend` instruction. Memory access of the form $\texttt{a}[3 * \texttt{i}]$ with $\texttt{VF} = 4$. Elements are labelled with their index in original program order.

Figure 3-4: Effect of reordering original loop iterations to match data layout.

The lane-collision removing permutation stage of Algorithms 1 and 2 is unneccesary when the data layout in a mapped register set is already free of lane-collisions. When the data layout is collision-free, we can obtain a large savings on data reorganization by skipping the permutation stage and directly blending mapped registers together (Figure 3-4). However, this approach can cause packed results to be produced with elements out of order with respect to the original loop (Figure 3-4b). If each iteration of the original loop was independent, then it is permissible to reorder operations within a SIMD instruction.

To obtain a legal vectorization of the original program, care must be taken to maintain matching orders of operations within the SIMD instructions in the vectorized loop. To ensure a legal vectorization, we must introduce a permutation whenever the order of elements in different operands of the same SIMD instruction do not match. We refer the reader to Figure 3-5 for a detailed example.

Determining the order of execution of scalar loop iterations within a vectorized loop iteration to minimize the overall number of permutations is an optimization problem which appears hard. The number of possible iteration orderings is the factorial of the vectorization factor, and each ordering implies a (possibly identity) permutation of every register which is the target of a gather or the source of a scatter. We do not attempt to solve the problem in this chapter. Instead, to keep the number of permutations reintroduced small, we apply a simple, practical heuristic. We examine the data flow graph of the vectorized program, and choose the most commonly observed element order of deinterleaved data elements as the order in which to execute the loop iterations. We verify that this heuristic is sufficient to achieve significant speedup in practice (Section 3.4.7). Having chosen this shared order, we apply Algorithms 3 and 4 to generate vectorized interleaving or deinterleaving code for each access, then scan the vectorized program, reintroducing permutations where necessary to enforce the chosen order of operation and ensure a legal vectorization.

ALGORITHM 3: Generate code to deinterleave a strided read without permutation

Input: A strided access a, load-mapped register set p, a **collision free** at VF
Output: Register s contains VF packed consecutive elements of a (out-of-order)

```
mask ← ⟨VF × L⟩;
masks ← stride copies of mask;
for i = 0 to VF − 1 do
    maskIdx ← ((stride ∗ i) + offset) mod VF;
    laneIdx ← ((stride ∗ i) + offset) / VF;
    masks[maskIdx][laneIdx] ← R;
end
for i = 1 to VF − 1 do
    generate: blend s, p[i − 1], masks[i − 1], s;
end
```

ALGORITHM 4: Generate code to interleave a strided write without permutation

Input: `VF` consecutive elements of a strided access `a` packed in register `r`, `a` **collision free** at `VF`

Output: Store-mapped register set `p` contains the `VF` elements of `a` (in store order)

$\texttt{mask} \leftarrow \langle \texttt{VF} \times L \rangle$;
$\texttt{masks} \leftarrow$ `stride` copies of `mask`;
for $i = 0$ *to* $\texttt{VF} - 1$ **do**
  $\texttt{maskIdx} \leftarrow ((\texttt{stride} * i) + \texttt{offset}) \mod \texttt{VF}$;
  $\texttt{laneIdx} \leftarrow ((\texttt{stride} * i) + \texttt{offset}) \ / \ \texttt{VF}$;
  $\texttt{masks}[\texttt{maskIdx}][\texttt{laneIdx}] \leftarrow R$;
end
for $i = 1$ *to* $\texttt{VF} - 1$ **do**
  generate: $\texttt{blend p}[\texttt{i} - 1],\ \texttt{r},\ \texttt{masks}[\texttt{i} - 1],\ \texttt{p}[\texttt{i} - 1]$;
end

3.3.2 Eliminating Permutations II: Simultaneously Resolving Collisions for Multiple Accesses

When the data layout in a mapped register set is not free of collisions, we cannot apply the optimization detailed in Section 3.3.1 to skip the lane-collision removing permutation stage of Algorithms 1 and 2. However, if we must perform some permutations to remove lane collisions, we can avoid permuting the entire register set for each access, as in Algorithms 1 and 2.

The goal is to choose, for each colliding access, a unique lane number in the range of `VF` lanes for each of the `VF` elements of the access. Let us say that two registers collide if *any* access occupies the same lane in both registers. In order to remove all collisions resulting from a strided mask, we can logically rotate one register by increments until our unique lane numbering condition is achieved for every contained access. If we extend this transformation to the full register set, so that element-wise vertical collision between each register and all registers with a lower index is removed, we have eliminated lane collision for all accesses in the contained access group with only one rotation of each register. As long as the group contains fewer than or exactly `stride` ac-

cesses, each vector lane holds at most one accessed element, which ensures that the transformation is possible. This property is ensured by our definition of an access group from Section 3.2 which sets the upper bound on the number of grouped accesses to exactly `stride`. Logical rotation is achieved using the `permute` instruction with a mask which arranges elements in rotated order. Since the transformation ensures that the mapped register set is free of lane collisions, it is always possible to apply Algorithms 3 and 4 immediately afterwards. Algorithms 5 and 6 state this combined approach.

Figures 3-5 and 3-6 show graphically the action of Algorithms 5 and 6. If the accesses are reads, then we can simply rotate each load-mapped register immediately after it has been loaded. However, for writes, the transformation is a little more subtle. For writes, rather than simply transforming a register set from one order to another, we are creating a register set in *transformed* order by combining registers with the `blend` instruction. To interleave a register resulting from computation into the store-mapped register set, we blend it with each store-mapped register in turn, with each blend forming a new store-mapped register by inserting one or more elements of the compute register. Since we are composing registers with the `blend` instruction, we must ensure that the access does not have lane collisions, which would require multiple stored elements to map to single lanes of the compute register.

Our approach runs as follows. We first compute what rotation of each store-mapped register is required so that each element of every access occupies a unique lane in the store-mapped register set. This gives us the *rotated store order* we must produce. Next, for each access to be interleaved, we take the lane number in our rotated store order of each element of that access. This gives us the required order of elements in the source register so that it can be blended into the store-mapped register set without collision.

Finally, each register resulting from computation is permuted into the required order before applying the `blend` sequence which combines it with each store-mapped register. Figure 3-6 shows this graphically. Intuitively, blending together these permuted registers results in the generation of a rotated image

of the store-mapped register set. We then perform an inverse rotation of each store-mapped register before storing to achieve the target store order. This scheme guarantees the introduction of at most one permutation per mapped register, plus at most one permutation per compute register to be stored, as opposed to the naive approach presented in Algorithms 1 and 2, which permutes every mapped register at least once for every access which shares it.

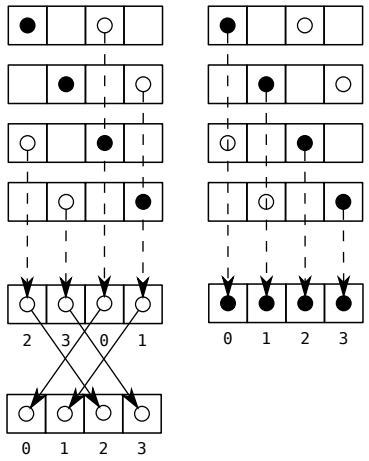


Figure 3-5: Algorithm 5. Our transformation enables vertical composition with the `blend` instruction of colliding memory accesses with a single transformation of the mapped register set. One access is out-of-order under vertical composition with `blend` after data layout transformation (left side). However, the vectorization can be legalized with a single `permute` as shown, to ensure the order of elements in each register is the same.

3.3.3 Statically Determining Lane Collisions

The presence or absence of lane collisions can be statically determined. To see why this is the case, consider any scalar access $a(\texttt{i})$: the memory region touched by the vectorized access is divided by our approach into buckets of `VF` consecutive elements. The extent of the memory region is $(\texttt{stride}{*}\texttt{VF})$ scalar elements, and the index within the region of element `i` of the scalar access is given by $(\texttt{stride} * i + \texttt{offset})$. The corresponding vector lane number of the element is found by floor modulo of this index by `VF`. It follows that the elements of any access will repeat vector lane numbers, and would collide when composed with the `blend` instruction, after $lcm(\texttt{VF}, \texttt{stride})$ scalar elements. Lane colli-

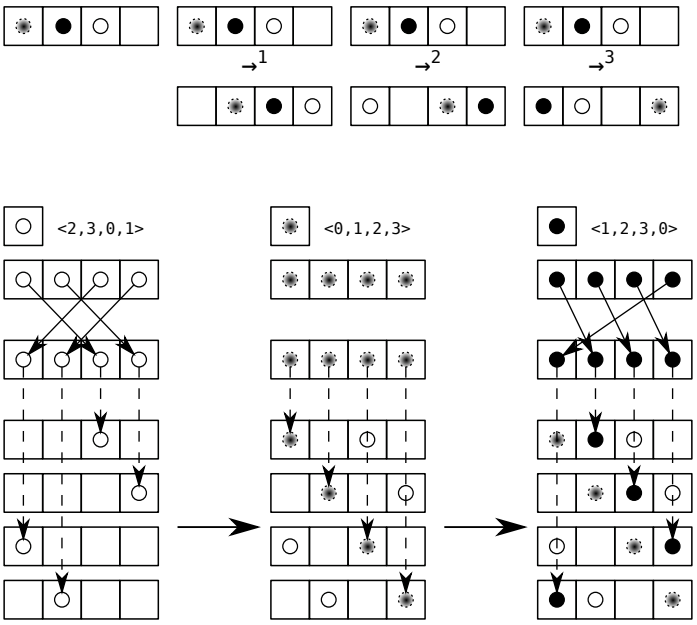


Figure 3-6: Algorithm 6. Accesses a[4 ∗ i + 2] (light dots), a[4 ∗ i] (patterned dots), and a[4 ∗ i + 1] (black dots). Desired store order at top with rotated store order underneath. Required permutation of each access to obtain rotated store order indicated with permute masks. Heavy arrows at bottom show the evolution of the store-mapped register set as each access is interleaved in. Final store-mapped register set before inverse rotation is shown at bottom right.

sion occurs when the length of the memory region touched by the vectorized access is larger than this quantity, i.e. when $(\texttt{stride} * \texttt{VF}) > lcm(\texttt{VF}, \texttt{stride})$. By definition, the condition is false when stride and VF are coprime.

3.3.4 Reassociation of Blend Instructions

Figure 3-8 demonstrates an important property of our blend instruction merging transformation. The effect of the transformation is greatest when the data dependence structure in the blend reduction sequence forms a *full* binary tree, where every node has either 0 or 2 children [70]. The tree structure of data dependence is determined by the associativity of operations in the vectorized progam.

However, the blend instruction sequences generated by the simple, canonical form of our technique are reduction sequences with a *linear* chain of dependence. That is, every instruction is of the form: `blend a, b, mask, a`. We

ALGORITHM 5: Vectorize a group of shared-stride reads with lane collisions

Input: Load-mapped register set p, rotations for each mapped register.
Output: Compute register set c produced, with some registers out-of-order

```
span ← stride ∗ VF;
lanes ← VF;
vectors ← span/lanes;
for i = 0 to vectors − 1 do
    mask ← {(x + rotations[i]) mod lanes | x ← [0..(lanes − 1)]};
    generate: permute p[i], mask, p[i];
end
foreach a in accesses do
    c_a ← p_0;
    for j = 1 to lanes − 1 do
        left ← j, right ← lanes − j;
        mask ← rotate(rotations[j], ⟨(left × L) + (right × R)⟩);
        generate: blend c_a, p[j], mask, c_a;
    end
end
```

ALGORITHM 6: Vectorize a group of shared-stride writes with lane collisions

Input: Register set c with n packed shared-stride accesses, with a common SIMD lane order
Output: Store-mapped register set p contains all accesses in rotated store order

```
span ← stride ∗ VF;
lanes ← VF;
vectors ← span/lanes;
foreach a in accesses do
    for i = 0 to vectors − 1 do
        mask ← ⟨lanes × L⟩;
        mask[(offset_a + i) mod lanes] ← R;
        generate: blend p[i], c_a, mask, p[i];
    end
end
for i = 0 to n − 1 do
    mask ← {(x + (lanes − i)) mod lanes | x ← [0..(lanes − 1)]};
    generate: permute p[i], mask, p[i];
end
```

begin with an empty register a, and accumulate lanes into it by repeatedly combining other registers with the accumulator. The tree structure of data dependences which results from this idiom has only a single operation at each level. However, we can exploit the associativity of the blend instruction to *algebraically reassociate* the blend operations in the vectorized program, trans-

forming the dependence structure so that the entire left and right subtrees of any one blend instruction are independent. Importantly, reassociation does not change the number of instructions. Algebraic reassociation is described in detail by Barik et al. [11]. Reassociation not only increases the count of blend instructions which can be merged, but also significantly increases the available instruction-level parallelism in blend reduction sequences.

Formally, our blend instruction can be represented as a binary operator $\oplus_m$, denoting the result of blending left and right operands with the mask m. Under this definition, the expression $(((a \oplus_m b) \oplus_{m'} c) \oplus_{m''} d)$ with initial masks m, m', m'', is equivalent to the reassociated expression $((a \oplus_n b) \oplus_{n'} (c \oplus_{n''} d))$ for some reassociated masks n, n', n''. Reassociation causes blend masks to change because the operands of the individual blend instructions are exchanged. Figure 3-7 states the formal rewrite rule for blend instructions in the vectorized code, with computation of reassociated blend masks n, n', n''.

$$\frac{\frac{\frac{(((a \oplus_m b) \oplus_{m'} c) \oplus_{m''} d)}{n \leftarrow m, \quad n'' \leftarrow \{(i,\ L) \mid (i,\ x) \in rights(m')\} \cup \{(i,\ R) \mid (i,\ x) \in rights(m'')\}}}{n' \leftarrow \{(i,\ L) \mid (i,\ x) \in active(n)\} \cup \{(i,\ R) \mid (i,\ x) \in active(n'')\}}}{((a \oplus_n b) \oplus_{n'} (c \oplus_{n''} d))}$$

Figure 3-7: Rewrite rule for reassociation of `blend` instruction sequences. The logical operation $rights(m)$ for some mask m selects all mask lanes which contain the `R` selector. The logical operation $active(m)$ selects all mask lanes which contain either `L` or `R` selectors.

Reassociation of a blend reduction sequence transforms the dependence structure. The sequences initially produced by our approach use a single register as an accumulator, blending in lanes from one register at a time to form a packed result. This approach results in low register pressure, requiring only a single live register to accomodate intermediate results, but requires sequential execution even when blend instructions can be independent. Fully reassociated blend sequences contain the same number of instructions, but perform the work as a parallel binary reduction. This approach has a high degree of in-

struction level parallelism, but increased register pressure versus the sequential reduction.

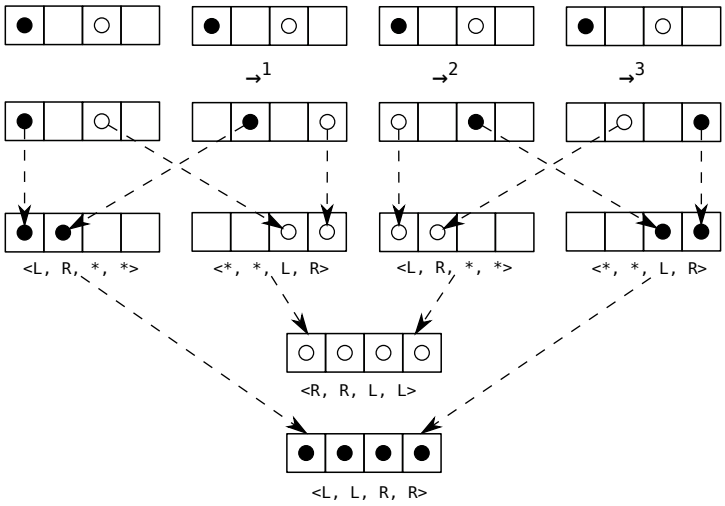
3.3.5 Eliminating Blends: Merging Multiple Blend Instructions

This section presents a novel optimization for blend reduction sequences of the sort generated by our technique. The key observation is that one vector register may hold the result of two different blend instructions if certain conditions are met. This allows us to merge multiple blend instructions into a single blend instruction, resulting in faster generated code with reduced register pressure. Intuitively, two blend instructions with identical left and right sources may be merged into a single blend instruction if the set of active output lanes of the two instructions are disjoint. The resultant register simultaneously carries the definition of both results of the initial pair of blend instructions. The resultant merged blend instruction may itself be merged with other blend instructions, and this merging may continue until all output lanes of the instruction are active. The merging can be defined as the result of a simple algebraic rewrite rule which may be applied repeatedly to the program to merge blend instructions.

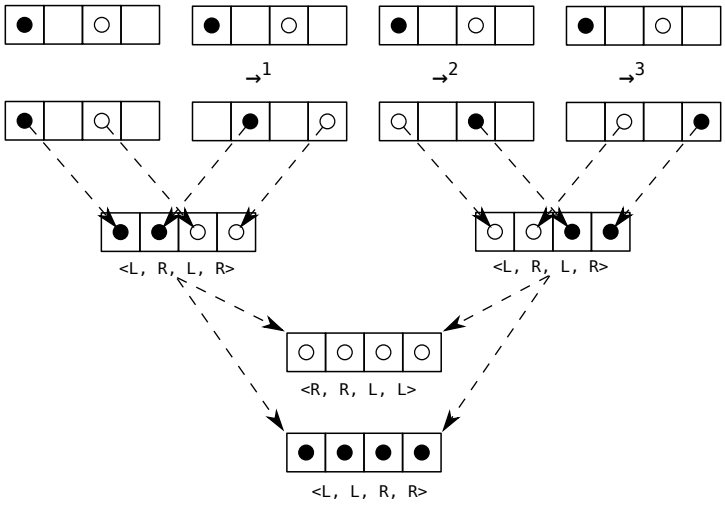
When stating program transformations as rewrite rules, the top line represents the initial program fragment, which is the pattern that must be matched to trigger the rule. The remaining lines represent necessary conditions which must hold for the rule to be applied. The final line represents the modified program fragment after the application of the rewrite rule. We represent masks in rewrite rules as indexed sets of mask elements. We say that two blend masks are disjoint iff everywhere there is an active lane in one, there is a corresponding don't-care lane in the other. Figure 3-8 shows graphically the action of this `blend`-merging rewrite rule, which is stated formally in Figure 3-9.

3.4 Evaluation

In this section we give an analysis of our technique, present a comparison with the technique of Nuzman et al., and present an experimental evaluation. In



(a) Before merging blends: 6 `blend` instructions, 4 temporary registers



(b) After merging blends: 4 `blend` instructions, 2 temporary registers

Figure 3-8: Graphical depiction of data flow before and after merging `blend` instructions. Two reads of the form $\texttt{a}[4 * \texttt{i}]$ (dark dots) and $\texttt{a}[4 * \texttt{i} + 2]$ (light dots). The load-mapped register set (top line of diagrams) is rotated as indicated to remove lane collisions (second line of diagrams). Masks for `blend` instructions are indicated below each result.

$$\frac{\texttt{blend r0, r1, mask1, r2} \qquad \texttt{blend r0, r1, mask2, r3}}{\frac{\{i \mid (i,\ x) \in mask1 \wedge x \neq *\} \cap \{j \mid (j,\ x) \in mask2 \wedge x \neq *\} = \emptyset}{\frac{mask3 \leftarrow (\{(i,\ mask2[i]) \mid (i,\ *) \in mask1\} \cup \{(i,\ mask1[i]) \mid (i,\ *) \in mask2\})}{\texttt{blend r0, r1, mask3, rNew}}}}$$

Figure 3-9: Rewrite rule for merging `blend` instruction pairs.

our experimental evaluation, we used a fixed phase ordering throughout. Our phase ordering was as follows:

1. Group accesses by the criterion of *shared spatial locality*
2. Hoist all invariants out of the control structure to simplify optimization
3. Vectorize the control structure using one of the approaches from Section 3.2, creating the program representation in our IR
4. Apply the `blend` reassociation transformation rule in Section 3.3.4 if requested
5. Perform dead code elimination and squash register copies using copy propagation
6. Repeated apply the blend-superimposition transformation from Section 3.3.5 until it is no longer applicable (the program converges to a fixpoint)
7. Generate native code and apply target-specific optimizations

3.4.1 Time Complexity of Generated Code

In Sections 3.2.1 and 3.3 we have presented three approaches for vectorization of interleaved memory reads and writes with arbitrary constant strides. Each approach results in the generation of a fixed number of `permute` and `blend` instructions. In order to facilitate compile-time decision making about which approach to use, we present an analysis of the time complexity of generated code in terms of the number of instructions generated by each of these techniques.

3.4.2 Simple Canonical Technique (Algorithms 1 and 2)

As described in Section 3.2.1, the size of the mapped register set for any access is at most `stride` vector registers. When `stride` $>=$ `VF`, at most `VF` registers are required to be combined to vectorize any one access. In this case, each active mapped register contains only a single data lane required by the vectorized access. Proceeding according to Algorithms 1 and 2, each vectorized access requires `VF` permutations (one per active mapped register). After reassociation, each packed register resulting from interleaving or deinterleaving is at the root of a full binary tree of blend operations with `VF` leaves. However, reassociation does not change the number of blend operations, which remains at `VF`-1 blends per access. The number of generated instructions for a single interleaved access is therefore 2`VF` $- 1$ operations using this approach.

3.4.3 Out-of-Order Technique (Algorithms 3 and 4)

When a strided access satisfies the alignment criterion of being collision free at `VF` (Section 3.3.1) we may apply Algorithms 3 and 4 to vectorize it. We exploit the independence of original loop iterations to change the scalar iteration order within a single vectorized loop iteration. This tactic allows us to choose an iteration order which reduces the overhead of data layout transformation. Following Algorithms 3 and 4, for any one strided access we generate the tree of `VF` $- 1$ blend operations to combine the elements of the access into a single register. We then inspect each packed register and determine the most common iteration order implied by the results. In the worst case, we reintroduce a permutation for every packed result to legalize the vectorization. The number of generated instructions is `VF` for each vectorized access.

3.4.4 Collision Resolving Technique (Algorithms 5 and 6)

Although the asymptotic complexities of the simple canonical approach and the out-of-order approach are both of order $O($`VF`$)$ for a single access, the number of generated permutes and blends for the out-of-order technique is approx-

imately half that of the canonical technique (VF versus $2\texttt{VF} - 1$). To reduce the number of generated instructions, the compiler should attempt to apply the out-of-order technique if it is applicable. Algorithms 5 and 6 introduce a method for resolving relative alignment constraints for any access conforming to Definition 1. As detailed in Section 3.3.2, the cost of the transformation is amortized when an access group contains more than one access. The total count of operations using Algorithms 5 and 6 for a group of n shared-stride accesses is `stride` permute operations to resolve lane collisions followed by VF operations per access to perform vectorization, for a total of $(n * \texttt{VF}) + \texttt{stride}$ operations to vectorize n shared-stride accesses. We summarize our analysis in Table 3.1.

Technique	Instructions Generated	Order
Algorithms 1 and 2	$n * (2\texttt{VF} - 1)$	$O(n * \texttt{VF})$
Algorithms 3 and 4	$n * \texttt{VF}$	$O(n * \texttt{VF})$
Algorithms 5 and 6	$(n * \texttt{VF}) + \texttt{stride}$	$O(n * \texttt{VF})$

Table 3.1: Time complexity (number of generated instructions) of SIMD interleaving and deinterleaving code generated by the proposed techniques, for a group of n accesses at a shared stride of `stride` elements.

Table 3.1 omits the effect of our blend merging transformation from Section 3.3.5. Arithmetic properties of the stride and offset of each access, and VF determine the contents of masks in the tree of `blend` instructions generated by our approach. Because of this, the effect of blend merging is highly dependent on the input program. However, accesses vectorized using our approach will often result in trees of blend instructions with a high degree of compatibility. These trees exhibit the property that `blend` instructions at corresponding locations in two trees are pairwise mergeable. In such cases, the original pair of trees can be merged up to the root instructions, which cannot be merged because they each produce a full output register after merging their subtrees. Figure 3-8 shows one of these programs. The original pair of trees have a combined count of $(2\texttt{VF} - 2)$ instructions before merging, and the merged tree contains VF instruc-

tions. Real-world benchmark program `cxdotp-2D` (Figure 3-15) exhibits this property, and blend merging has a pronounced effect.

It is not possible to state the effect of blend merging on time complexity for a single access, because blend merging amortizes the total cost of blending over a group of accesses. We can characterize the effect of the transformation on an access group with an extra assumption. When an access group is full, by definition, accesses with all n distinct offsets are present. When n is even, each tree of blend instructions can be merged with exactly one other tree, and $n/2$ blend sequences result from blend merging. When n is odd, one instruction tree cannot be paired, and $n/2 + 1$ sequences result. The number of blend instructions required to interleave or deinterleave the entire group of n accesses, for $n > 1$, is thus $(n/2) * \texttt{VF}$ for even n, and $(n/2 + 1) * \texttt{VF}$ for odd n. Blend merging does not change the asymptotic complexity, which remains of order $O(n * \texttt{VF})$.

3.4.5 Comparison with Nuzman et al.

The technique of Nuzman et al. [53] generates extremely efficient code for interleaved access with power-of-two strides. However, the approach can only handle powers of two — when the stride is not a power of two, the technique of Nuzman et al. is not applicable. We generalize the approach of Nuzman et al. to arbitrary constant strides. Nuzman et al. use an intermediate representation with a small number of primitives, shown in Figure 3-10. These primitives precisely express interleaving/deinterleaving where the stride is a power of two. Our representation uses more generic primitives, which can express interleaving/deinterleaving at any constant stride, but require a constant factor more operations for power-of-two stride. This constant factor is demonstrated by the direct correspondence between each of the primitives of Nuzman et al. and a short, fixed sequence in our representation. The sequence corresponding to each Nuzman primitive is indicated in Figure 3-10, for $\texttt{VF} = 4$. This correspondence between representations often leads to identical native code after instruction selection when the stride is a power of two, because the native in-

structions implementing the primitives of Nuzman et al. are also selectable for the corresponding sequence in our representation.

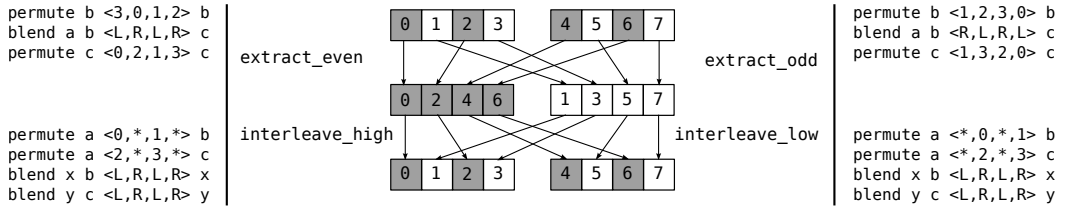


Figure 3-10: Primitive operations of Nuzman et al. `extract_even` and `extract_odd` extract sequential even or odd indexed lanes from two source registers. `interleave_high` and `interleave_low` perform the inverse data movements. Corresponding operations in our representation shown to the left and right of the diagram, for $\texttt{VF} = 4$.

The number of generated instructions in vectorized code from the technique of Nuzman et al. for power-of-two strided accesses is of the same order as our proposed techniques. As detailed in Nuzman et al. [53], their technique generates a perfect, complete binary tree of instructions of $log_2(\delta)$ levels for a vectorized interleaved memory access, where δ is the stride of access. The trees generated are perfect and complete because the technique only considers programs where δ is a power of two, and each level is formed by combining pairs of adjacent inputs from previous levels [70]. The $\delta - 1$ generated instructions correspond to the reassociated tree of blend operations produced by our approach. However, for any one vectorized access, the number of vector registers which must be combined is bounded above by `VF`, by the same argument as for our approach (Section 3.4.1). The maximal instruction count is thus $\texttt{VF} - 1$, making the asymptotic complexity of extraction of a group of n accesses order $O(n * \texttt{VF})$ using either technique.

3.4.6 Native Code Generation

In this chapter we address the problem of vectorizing interleaved access, and propose an approach which can generate vectorized code for accesses with arbitrary constant strides. However, generating correct vectorized programs is only of use if vectorization improves the overall performance of the program.

An important step in realizing a performance gain in practice is the generation of fast native code.

In our implementation, we use a modified version of the "Bottom-Up Tree Pattern Matching" approach [2, 10], with a simplified cost heuristic. For a subset of the available native data reorganization instructions on our two experimental platforms, we derived a table mapping each native instruction to the corresponding tree of abstract operations in our IR. Driven by this table, we perform the tree rewrite by greedily selecting the native instruction which covers the largest available subtree of our abstract operations at each step. It is possible to improve on this approach to instruction selection [29], particularly for vector instructions [11], but we found that even this simple approach was sufficient to realize a practical speedup from our techniques in experimental evaluation. Native instruction selection is a large topic, and is not the focus of this chapter, but future work could involve the use of an optimal instruction selection scheme to increase the performance of code generated by our approach. Possibilities in this direction are discussed in Section 3.6. In particular, GCC's instruction selection for the primitives of Nuzman et al. is very efficient, as can be seen by looking at the powers of two strides in Figures 3-11a and 3-11b. However, our simple scheme sometimes makes a better selection even for powers of two (Figure 3-11b, stride=4).

3.4.7 Experimental Evaluation

Our benchmarking was carried out on two experimental platforms: we used an Intel Core i5-2500 (Sandy Bridge) system with 16GB of RAM as our 128-bit "SSE" platform, and an Intel Core i5-4570 (Haswell) system with 32GB of RAM as our 256-bit "AVX2" platform. Experiments were run on Linux (kernel version 4.1). We followed the guidelines outlined in Paoloni [58] for benchmarking short programs on our experimental architecture. Our baseline scalar code was generated by running GCC on plain C code, with optimization level *-O3* and vectorization disabled. GCC implements Nuzman's algorithm for vectorization of interleaved access. For comparison with Nuzman, we generated

vectorized code using GCC *-O3*. We inspected the generated assembly and verified that GCC applied Nuzman's algorithm where the stride is a power of two. We implemented our vectorization techniques in a simple compiler that generates vector intrinsics, and compiled the resulting code with GCC *-O3*.

Figures 3-11 through 3-14 present the results of synthetic benchmarking of programs performing data movement on SSE. We generated programs which performed either a gathering operation (Figures 3-11 and 3-12) or a scattering operation (Figures 3-13 and 3-14). We present the speedup achieved by our simple, canonical approach using Algorithms 1 and 2 and our reordering approach using Algorithms 3, 4, 5 or 6 as appropriate. In all cases, the stride of access was swept through the range $[2, 16]$ — this choice was influenced by the experimental architecture (SSE), which has 16-byte vector registers. Where the stride of any individual gathering or scattering operation exceeds 16 bytes, it must perform at least as many vector memory operations as there were scalar memory operations in the original loop. On our SSE experimental platform, regardless of the technique employed, we would expect the performance of vectorized memory access to degrade as stride length increases.

3.5 Discussion of Results

3.5.1 Performance Limits

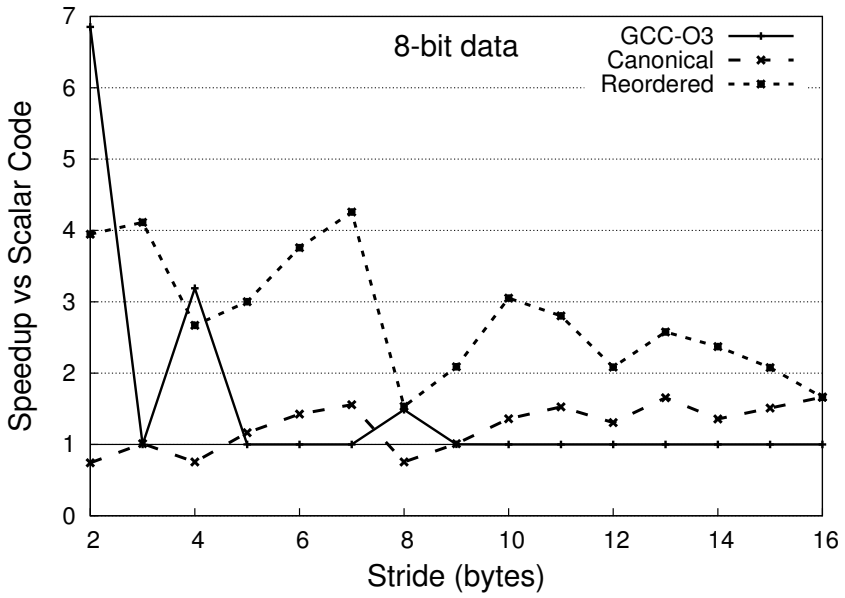
Performance degradation with increasing stride is visible in Figures 3-13 and 3-14, but only up to the length of an architectural vector register. Note that the stride of access in each graph is given in units of the word size, so a stride of 16 bytes is exceeded very quickly for wider types. In each case, when the stride exceeds 16 bytes, the performance of vectorized memory access is reduced to parity or near-parity with the scalar code, but does not further degrade with increasing stride, up to a stride of 128 bytes (the largest experimental value). Further, the experiments show that in general, large speedups are possible for single strided accesses where the stride is shorter than a vector register. The performance drop at strides longer than a vector register is significantly less

pronounced for gathering operations than for scattering operations. For example, our approach achieves 1.66x speedup versus scalar code performing a 64-bit stride 7 gather operation (Figure 3-12b). The stride of this operation is 56 bytes, much longer than a 16-byte vector register on our SSE experimental platform. The strategy of tiling a memory region with vector loads and composing required elements into results with SIMD instructions appears particularly effective on SSE. For 8-bit stride 3 gather operations (Figure 3-11a) our approach results in more than 4x speedup over scalar code, but GCC cannot vectorize the program (the approach of Nuzman et al. is not applicable). This case of data movement is ubiquitous in image and video processing applications, where formats using packed 8-bit triples are common.

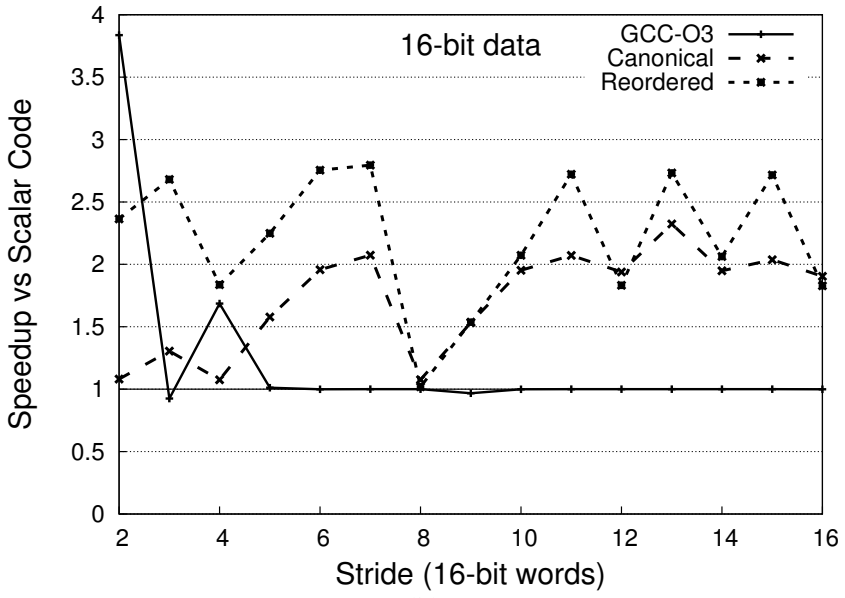
3.5.2 Effect of Reordering

The most pronounced difference between gathering (Figures 3-11 and 3-12) and scattering (Figures 3-13 and 3-14) results is in the effect of reordering loop iterations to reduce permutation overhead of interleaved access. When scalar data elements are positioned so that there is no lane collision while blending, we can apply Algorithms 3 and 4 and forego the permutation phase which removes lane collisions, resulting in a shorter program. Although analytically the reduction in number of generated instructions is equivalent for both Algorithm 3 and 4, a key semantic difference is that vector loads with unused lanes do not require different treatment from loads without, whereas vector stores with unused lanes must preserve the contents of memory between stored elements. We implemented such stores using a read-modify-write sequence, using our `load`, `blend` and `store` instructions. For stores, the relatively long latency of read-modify-write memory access acts to smooth the speedup obtained from improvements in data reorganization.

Figure 3-11: Synthetic benchmark results: 8-bit and 16-bit gathering operations

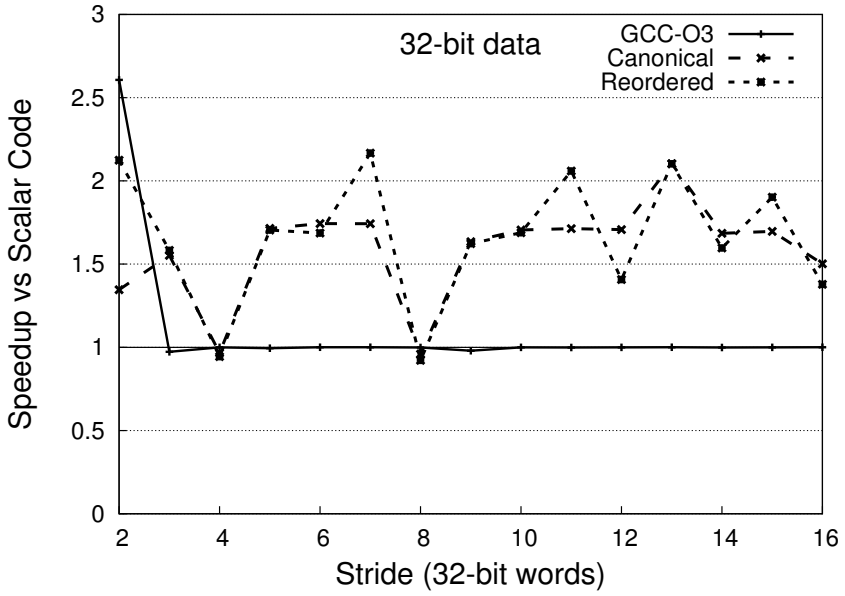


(a)

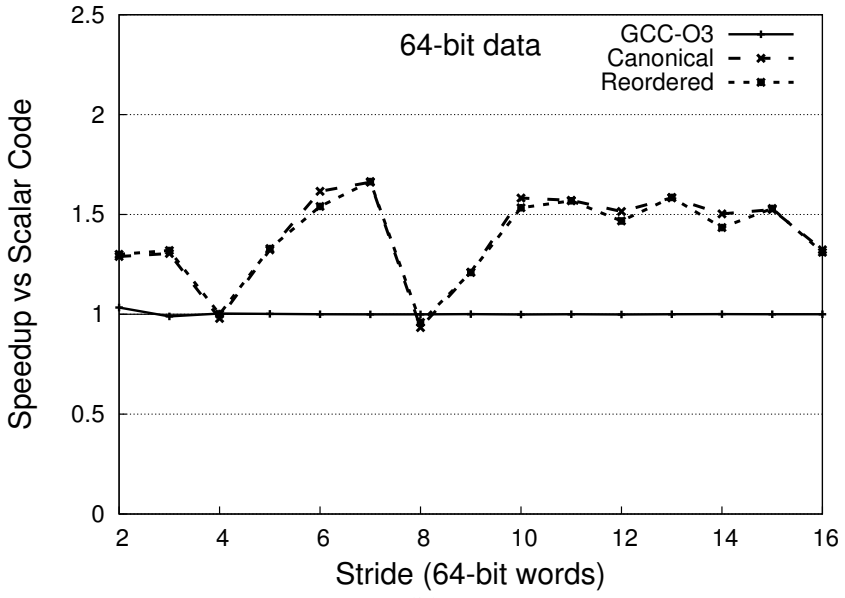


(b)

Figure 3-12: Synthetic benchmark results: 32-bit and 64-bit gathering operations

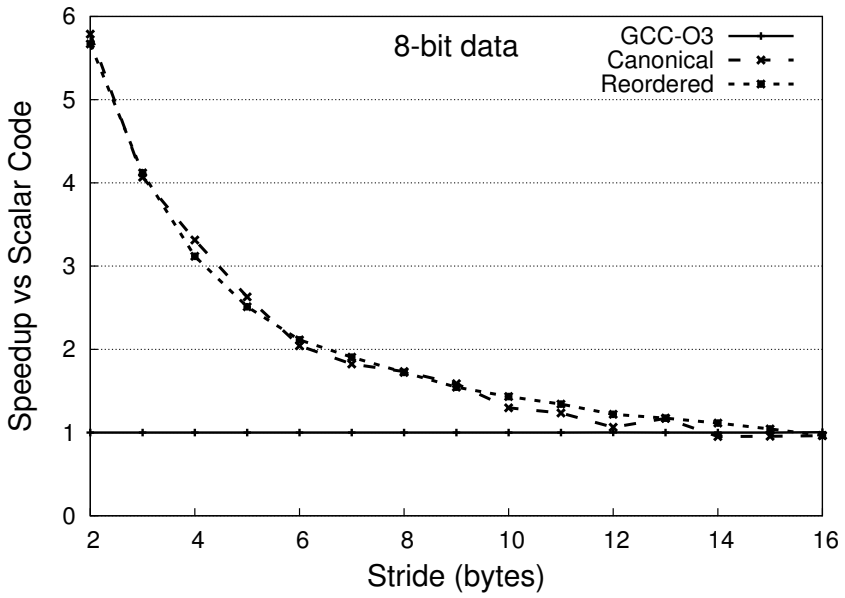


(a)

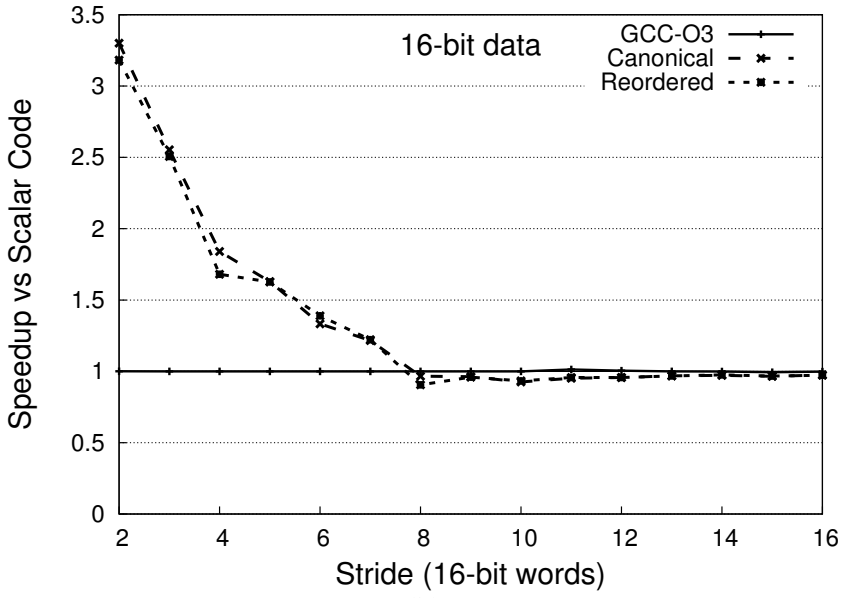


(b)

Figure 3-13: Synthetic benchmark results: 8-bit and 16-bit scattering operations

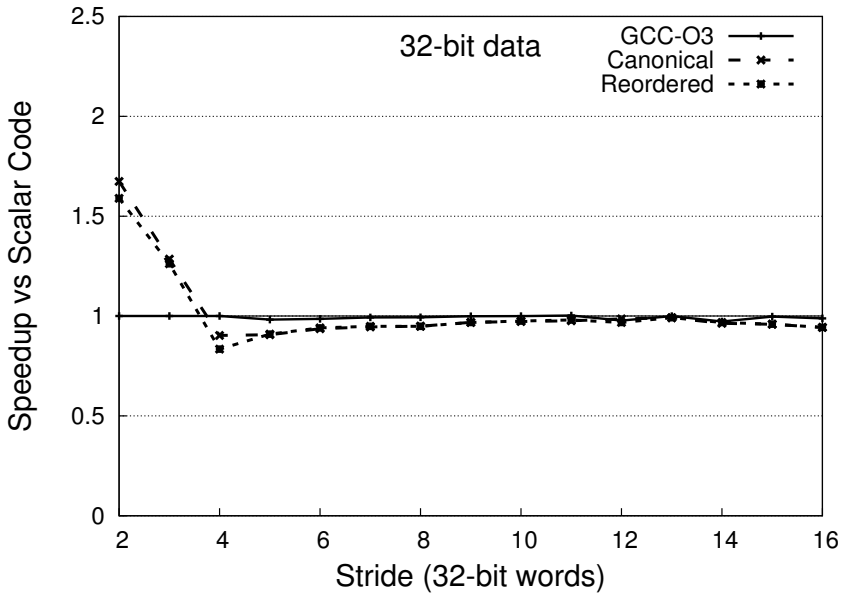


(a)

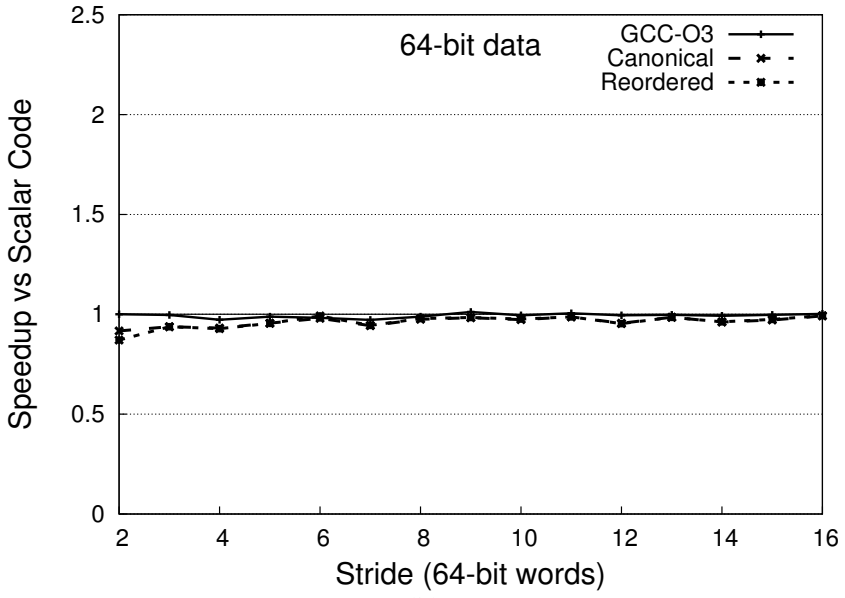


(b)

Figure 3-14: Synthetic benchmark results: 32-bit and 64-bit scattering operations



(a)



(b)

3.5.3 Relation of Speedup to Stride

When performing vectorized interleaved memory operations, there is often a pronounced difference between our simple canonical approach and reordered approach at neighbouring strides. At any fixed vectorization factor, accesses at a given stride will either exhibit lane collision or will not, determining whether permutations must be introduced to resolve these lane collisions. The presence of lane collisions can be statically determined — lane collisions are not present when `stride` and `VF` are coprime (Section 3.3.2). On our SSE experimental platform, the natural vectorization factors for the four machine types with distinct bit-width are powers of two ($\texttt{VF} = 16$ for 8-bit data, $\texttt{VF} = 4$ for 32-bit data, and so on), meaning that lane collisions are generally present at even but not odd strides, once `stride` exceeds `VF`. This pattern is observed in the simple oscillation of speedup across neighbouring strides in our synthetic benchmarks, though smoothed in the case of stores as previously noted.

3.5.4 Variability of Scalar Code

We would expect to see the oscillation previously mentioned throughout synthetic benchmarking, but it is often obscured by variability in the scalar code. The performance of the scalar code produced by GCC at any two neighbouring strides can be significantly different. In particular, GCC does very well when optimizing gathering operations for locality, and incorporates several patterns which produce fast code for common or idiomatic memory access patterns. In our synthetic benchmarking, GCC sometimes produces code which is faster than the best vectorized code produced from our approach. We investigated the performance difference and found that GCC chose to vectorize the data movement using Nuzman's algorithm. However, post-pass instruction selection emits scalar code for the abstract operations of Nuzman et al. Performing this devectorization step requires a very detailed cost model, and we did not attempt to replicate it in our experimental compiler. Apart from these cases, the performance of the best vectorized code produced by our approach matched or exceeded the performance of code generated by GCC at optimization level *-O3*.

In Figures 3-11 and 3-12, for some power-of-two strides, the speedups achieved by GCC and our approach are identical. In these cases, both our technique and the technique of Nuzman et al. result in identical native code after instruction selection.

3.5.5 Real-World Benchmarking (SSE)

Our real-world benchmarking uses a selection of BLAS Level 1 [43] routines with varying access patterns. Figure 3-15 (page 81) presents the results on our SSE experimental platform. We display speedup over scalar C code obtained by GCC using optimization level *-O3*, and also using each of the techniques we propose. Details of each benchmark program are listed in Table 3.2. We vectorize computation by simple scalar expansion.

BLAS L1 Routine			Benchmark Instantiation						
			Loads		Stores			GCC Applies	
Type	Name	Dim.	Stride	#	Stride	#	C Type	SIMD	Nuzman
cx	axpy	1D	2	4	2	2	`float`	Yes	No
cx	mul	1D	2	4	2	2	`float`	Yes	Yes
cx	dotp	2D	4	8	2	2	`float`	Yes	Yes
cx	dotp	3D	6	12	2	2	`float`	No	N/A
s	dotp	2D	2	4	1	1	`float`	Yes	Yes
s	dotp	3D	3	6	1	1	`float`	No	N/A
s	dotp	5D	5	10	1	1	`float`	No	N/A
s	norm	2D	2	2	1	1	`float`	Yes	Yes
s	norm	3D	3	3	1	1	`float`	No	N/A
s	norm	5D	5	5	1	1	`float`	No	N/A

Table 3.2: Summary of dimensions, strides, underlying C types, and vectorization realized for each instantiation of the BLAS Level 1 routines used in benchmarking in Figure 3-15. For example, cxdotp-3D is the dot-product of 3-dimensional vectors of complex numbers. The memory access pattern consists of 12 stride 6 loads, and 2 stride 2 stores, and the underlying C type is `float`. GCC does not vectorize this program, and the approach of Nuzman et al. is not applicable.

Of the 10 programs, 5 can be vectorized using the technique of Nuzman et al., and GCC applies it in 4 of 5 cases. In the case of `caxpy`, GCC uses a combination of classical optimizations to transform the program so that memory access

becomes consecutive, and vectorizes using scalar expansion. This results in extremely compact code, which achieves a speedup of over 2x versus scalar code. Applying the techniques we have described results in a speedup of 1.8x using reassociation and reordering.

The additional effect of applying reordering, reassociation, and blend merging is visible in many of the benchmarks, particularly `cxdotp - 2D`. Generally speaking, our simple canonical approach produces code that performs slightly worse than scalar code. However, for benchmarks dominated by computation, such as the `vnorm` programs, the overall speedup from vectorization is large, despite suboptimal data movement. For `vnorm - 3D` and `vnorm - 5D`, applying our simple canonical approach results in a program which is more than 2.5x faster than scalar code. GCC requires the *-ffast-math* option to vectorize the computation in the `vnorm` benchmarks, which contains a floating point square root operation. GCC generates a reciprocal square root operation followed by some iterations of the Newton-Raphson method for approximation of square roots. In order to fairly represent the effect of our techniques, we precisely duplicated this instruction selection. Applying our optimization techniques improves the speedup factor on data movement, bringing the overall speedup to more than 3x. In the `vnorm` programs, the interleaved access pattern is the principal impediment to vectorization. Once it is removed, significant performance gain is possible.

In each of the 4 cases where GCC applies the technique of Nuzman et al. — `cxmul`, `cxdotp - 2D`, `vdotp - 2D`, and `vnorm - 2D` — we generate programs which run faster on our SSE experimental platform, with the exception of `vnorm - 2D` where the error bars overlap. This gain is primarily due to our optimization techniques, particularly reordering, which can eliminate permutation instructions from the program. The effect of merging blends is particularly visible in `cxdotp - 2D`. The program contains 16 mergeable blend operations, each of which has two inactive lanes. Blend merging reduces this to 8 blend operations where every lane is active. Even though blend merging does not

change the asymptotic complexity of the generated code, it can lead to significant speedups in practice.

3.5.6 Real-World Benchmarking (AVX2)

Figure 3-16 (page 82) presents the results of real-world benchmarking on our AVX2 experimental platform. The figure demonstrates that our approach yields portable performance improvements across these two platforms. For most of the benchmarks, doubling the vectorization factor by moving from 128-bit to 256-bit vectorization yields a significant performance improvement. However, the benchmark `caxpy` is an exception. Our cost heuristic for instruction selection (Section 3.4.6) does not take into account fine-grained microarchitectural differences between the Haswell and Sandy Bridge platforms. Several 256-bit AVX2 versions of 128-bit SSE data reorganization instructions have either longer latency or require exclusive access to functional units where the SSE instruction does not. In addition, AVX2 instructions which reorganize data across the 128-bit boundary in a 256-bit register are subject to performance penalties relative to instructions which do not. In order to account for these differences, a detailed cost model would be required for instruction selection. However, for 9 out of 10 benchmarks, our technique results in efficient native AVX2 code. The `caxpy` benchmark exhibits a performance decrease relative to SSE in part because the instruction count is small, magnifying the effect of architectural differences.

Another significant difference from the SSE results is that the performance of the code generated by GCC is typically worse — GCC's code generation for AVX2 is not as mature as for SSE. On our SSE experimental platform, GCC achieves a geometric mean speedup of 1.43x over scalar code on this set of benchmark programs, but on AVX2 this is reduced to 1.30x. However, our approach achieves very good performance portability, represented by an increase in geometric mean speedup from a maximum of 1.77x on SSE to 2.53x on AVX2.

3.5.7 Comparison with Hand-Tuned and Reference BLAS

We performed some benchmarking of our generated code versus an open source reference BLAS implementation and Intel's MKL. The results are presented in Figure 3-17 on page 83. For small problem sizes (in the range of 1K to 64K data elements) which exhibit dense interleaved data access, the code generated by our approach significantly outperforms both BLAS implementations experimented with.

Single-core execution was used throughout. The performance gap begins to close only when the working set size grows so large that cache and memory effects come into play, i.e. at sizes which are ill-suited for single-core SIMD execution. Our approach could be used to produce optimized vector code for the individual single-core portions of a larger multi-core BLAS operation when the data access is interleaved.

Typically, BLAS implementations are tuned to take advantage of multicore parallelism and the memory hierarchy to achieve good performance when dealing with large amounts of data [82]. While BLAS implementations deal with both sparse and dense data representations, non-stride-1 (interleaved) dense data access is difficult to optimize in a library context.

It would be possible to provide a small number of hand-tuned kernels specialized for common strides, but doing so for every possible stride is implausible. Automated tuning systems such as ATLAS [84] perform install-time code generation to automatically create an optimized library for the target platform.

To avoid the need to generate an optimized kernel for every possible stride, a complementary compile-time specialization of operations using an approach similar to SPIRAL [27] could be used. This seems like a promising direction for future work.

3.6 Related Work

Much of the related work presented here is also discussed in our literature review in Chapter 2. Here we discuss more specific relationships between our work and related works in terms of the techniques presented in this chapter.

A key difference with much existing work discussed in Section 2.3 is that we do not take a fixed permutation and try to generate fewest instructions to perform it. Rather, our techniques synthesize SIMD instruction sequences to perform interleaved memory access, which may contain permutations. The most closely related work, that of Nuzman et al. [53] is discussed in depth in Section 3.4.5. Kudriavtsev and Kogge [40] propose to reorder operations within SIMD instructions to minimize the number of permutations in the program. Using their vectorization approach, permutations can occur when multiple scalar operations read a common subexpression, or as a result of permutation in the source program. While the aim of minimizing permutations is similar to the aim of our reordering approach (Section 3.3.1) the key difference is that Kudriavtsev and Kogge require memory access to be consecutive, and reorder operations to minimize permutations resulting from computation. We reorder operations specifically to minimize permutations resulting from interleaving/deinterleaving. Future work could consider a combined approach, but the resulting multi-objective optimization problem appears hard.

Ren et al. [68] optimize straight-line code by merging, propagating, and decomposing permutations within a basic block. Although their work does not address vectorization of interleaved memory access, they note that it often introduces permutations, using a power-of-two stride example which can be vectorized by Nuzman et al. [53]. They further note that producing optimal code (that is, with fewest permutations) for an arbitrary basic block maps to the NP-hard multiterminal cut problem, and propose a practical heuristic solution. The approach of Ren et al. could be applied to further optimize the permutations in our synthesized programs.

Karrenberg and Hack [35] propose a holistic approach to vectorization of whole functions, encompassing control and data flow. However, their approach

assumes consecutive memory access. If combined with the approach to vectorizing interleaved access which we propose, that restriction would be lifted for programs where the stride of access is known at compile time, enabling broader application of their approach.

Park et al. [59] propose a "SIMD Defragmentation" approach which tries to extract parallelism at the level of subgraphs of operations within the program, similar to SLP. Where vectorization using their approach results in interleaved memory accesses, our techniques could be applied to synthesize optimized SIMD code sequences to perform the access.

Barik et al. [11] propose an approach for efficient selection of vector instructions for straight-line code sections, which is tightly integrated with register allocation and instruction scheduling. Although their cost model formulation includes parameters for the cost of packing or unpacking data in vector registers, they do not propose a technique for generating the code which performs interleaved access.

Their experimental evaluation compares their approach to a prototype of SLP [42] using benchmark programs with restricted memory access patterns of the type discussed in Section 3.6.1. However, our techniques and their optimization approach are synergistic — if both were available in the compiler, their cost model could be extended to incorporate the costs of interleaved access vectorized using our approach. Similar to the work of Karrenberg and Hack [35], this would enable broader application of both techniques.

3.6.1 Superword Level Parallelism

Vectorization techniques based on SLP [42] incorporate a tactic which can handle a restricted case of interleaved memory access. SLP attempts to transform programs so that interleaved memory accesses become consecutive, by searching for an unrolling factor for scalar loops which results in a dense memory access pattern in the vectorized loop. Such an unrolling factor can only be found if the scalar loop touches every element of each accessed array region within a

fixed number of loop iterations, that is, if the access pattern of the loop has no gaps.

Previous work on SLP [75] has incorporated blending operations. The work extends SLP to enable it to vectorize control flow by converting conditional expressions in the scalar loop into predicated vector expressions (if-conversion), making use of blending operations to represent predicated results. However, we apply blending operations in a very different way, using them to perform interleaving/deinterleaving.

An recent extension to SLP [48] uses the polyhedral model [21] to generate a non-SIMD data movement phase which gathers nonconsecutive memory elements in compact temporary arrays in the prologue of the vectorized loop, allowing computations within the loop to access them as though they were consecutive in memory. This approach sidesteps the need to generate vectorized code to perform data movement, and is sufficient to achieve speedup over scalar code for some programs.

However, the approach assumes read-only array references, does not attempt to deal with interleaved writes, and requires accessed data to be copied to a temporary array before every loop iteration. Our approach synthesizes vectorized code to perform interleaved reads and writes in the vectorized loop, where instruction-level parallelism between data movement and computation can offset the overhead of memory access.

3.6.2 Alignment

Eichenberger et al. [24], propose an approach to solve alignment issues while vectorizing. However, the focus of that work is reducing the cost of misaligned vector accesses resulting from unit-stride code. The operation of the initial permutation phase in our approach is similar to realignment using the dominant shift policy of Eichenberger et al., but our objective is not to reduce the cost of realignment, but of interleaving/deinterleaving. Our approach locally realigns accesses within a set of vector registers acting as a compiler-controlled cache, reducing the number of instructions required for interleaving/deinterleaving.

However, because accessed data is cached in vector registers, the misalignment does not translate into misaligned memory accesses.

3.6.3 SPIRAL

The initial version of the SPIRAL system [27] vectorized certain classes of interleaved memory access by translation to C macros. A set of handwritten implementations of these macros using SIMD intrinsics was included for each target platform. However, the class of interleaved access vectorized by SPIRAL is distinct from that vectorized by our approach, which is specifically targeted at affine interleaving (Definition 1). Subsequently the authors proposed an approach to automatic generation of vectorized code for their class of interleaved memory access [28], but their approach differs from ours in two key respects.

The authors show that their technique produces a locally optimal code sequence for any one data permutation in a class they consider. However, locally optimal treatment of individual permutations does not guarantee a global optimum. In fact, both our work and the work of Nuzman et al. [53] have shown that optimizing multiple simultaneous permutations with spatial locality as a group allows further optimization and sharing of overheads.

Furthermore, the approach proposed [28] considers only shuffling operations. A key innovation of our approach is the decomposition of the problem into separate permutation and blending phases. We have shown in Section 3.3 that this leads to the amortization of overhead across multiple interleaved accesses, and exposes opportunities for optimizations like our blend merging technique (Section 3.3.5). Using blending operations can lead to the elimination of permutations entirely (Section 3.3.1).

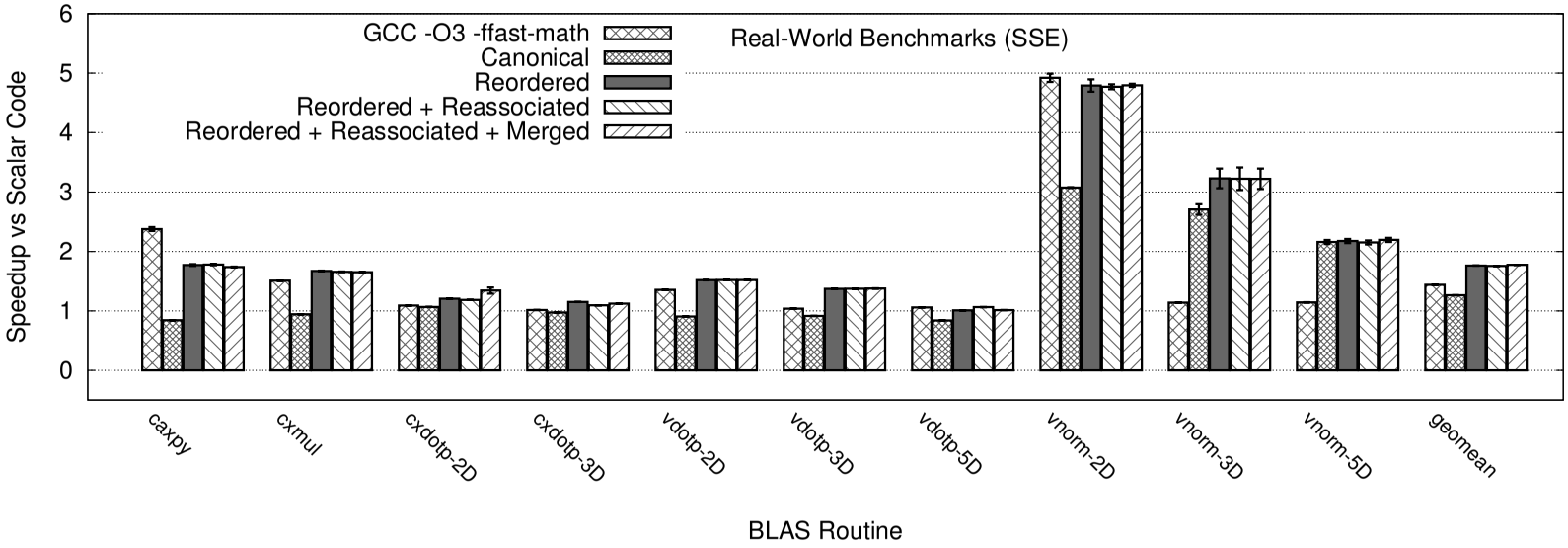


Figure 3-15: **SSE**: BLAS Level 1 operations exhibiting interleaved memory access optimized by GCC and using our approaches. Mean speedup over 10,000 runs of each program is plotted, with error bars showing a variance of one standard error about the mean. *geomean* is the geometric mean speedup of each approach over all benchmarks.

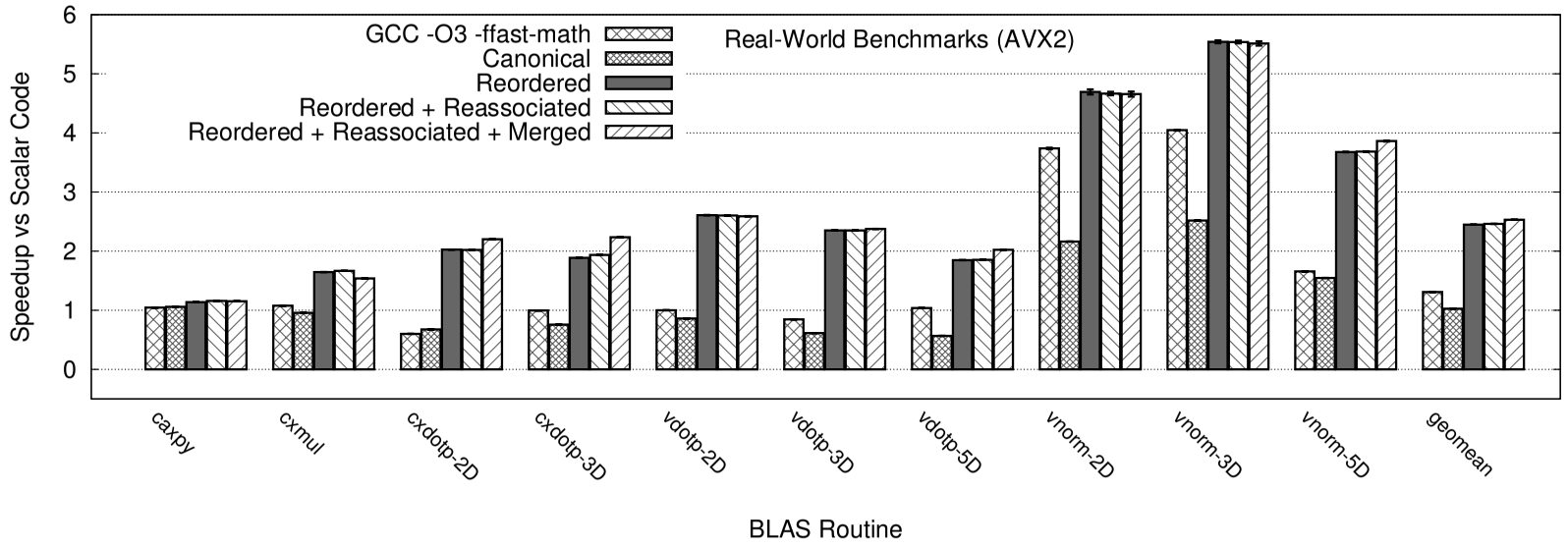


Figure 3-16: **AVX2**: BLAS Level 1 operations exhibiting interleaved memory access optimized by GCC and using our approaches. Plot parameters are the same as for the SSE graph (Figure 3-15).

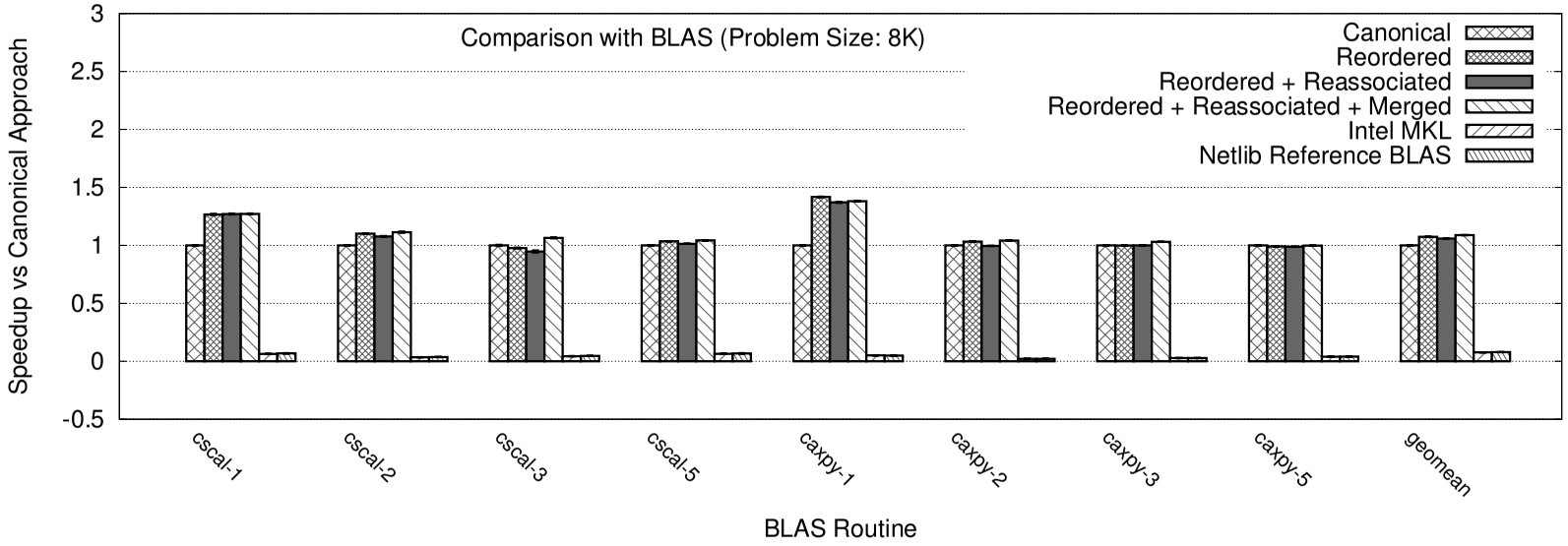


Figure 3-17: **SSE**: BLAS Level 1 operations `axpy` and `scal` with interleaved memory access. The number in x-axis labels indicates the value of the *increment* parameter in the BLAS operation. Problem size in the graph title indicates the length of arrays. BLAS operation type is `complex`.

Chapter 4

Vectorization of Multibyte Floating Point Data Formats

The IEEE Standard for Floating-Point Arithmetic (IEEE-754), standardized a computer representation of real numbers. The most recent version of the standard was published in 2008 [91]. The standard specifies a number of finite representations of real numbers to allow them to be stored in a computer memory, as well as operations on these representations. Since its initial publication in 1985, the majority of hardware implementations of floating-point arithmetic conform to the standard, providing a portable environment for floating-point computation.

To provide programmers with an interface to hardware facilities for floating-point computation, programming languages such as Fortan and C, among others, provide numeric types which correspond to the types defined in IEEE-754, as well as operations to manipulate the floating-point environment for computation. For example, the Fortran 90 type `REAL*4` and the C type `float` correspond to the IEEE-754 type `binary32`, which represents a real number as a 32-bit binary floating-point value.

4.1 IEEE-754 Floating Point

The IEEE-754 2008 standard [91] defines a number of finite binary representations of real numbers at different resolutions (16, 32, and 64 bits, among others). Each format encodes floating-point values using three binary integers: *sign*, *exponent*, and *mantissa*, which uniquely specify a point on the real number line following a general formula.

$$v = (-1)^s \times (1 + \sum_{i=1}^{M} (m_i 2^{-i})) \times 2^{e-bias}$$

v is the real number obtained by the evaluation of the formula for a *normalized* floating-point value with sign s, mantissa m of length M bits, and exponent e. The value $bias$ is an integer constant which differs for each format. The formats all use a single bit to represent the sign, but different numbers of bits for the exponent and mantissa components.

The exponent determines a power of two by which the rest of the number is multiplied, while the mantissa represents a fractional quantity in the interval $[1, 2)$ obtained by summing successively smaller powers of two, starting from 2^{-1} and continuing up to the length of the mantissa. If the ith mantissa bit is set, the corresponding power of two is present in the sum, and if it is unset, the power of two is not present. For normalized numbers, the leading 1 is not explicitly stored in the mantissa, but is implied.

A key feature of the representations defined in IEEE-754 is that their width is *fixed*. This can lead to scenarios where the full precision of a standard floating-point type is not required in the normal operation of the application.

4.2 Thought Experiment: 24-bit `float`

Consider an application which produces results in IEEE-754 single-precision binary floating-point format but requires only 15 stored mantissa bits to precisely represent any result. The `binary32` format provides 23 bits. An example

scenario might be the collision detection calculations in a video game where the player's movement is in coarse steps, and a distance calculation involving square roots is involved. Square roots can be calculated using an iterated algorithm which starts with a guess for the value of the square root, and repeatedly refines the guess, increasing the accuracy of the result. In order to speed up the calculation, the developer, knowing that distances need only be accurate to a few decimal places in this scenario, might use a faster root finding algorithm which performs fewer iterations, but gives a less accurate result.

Even though the results produced do not need the full precision of the `binary32` type, they must still be manipulated, loaded, and stored using that type, since it is the only corresponding floating-point type the machine provides. However, this leaves the developer in an unfortunate scenario: because their program does not require the full precision available, the last 8 bits of each floating point value resulting from a collision detection calculation will contain garbage. Effectively, one of every four bytes in the resulting data stream is unused. The application can therefore only properly utilize a maximum of 75% of the memory used for storing these values. Figure 4-1 shows the memory layout of floating point values indicating the unused area due to reduced accuracy of results.

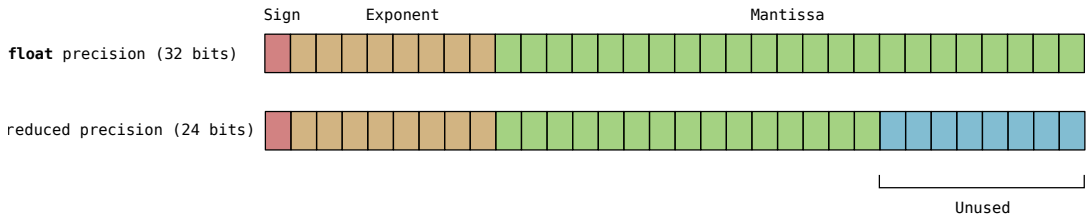


Figure 4-1: Unused bit positions in reduced-accuracy results

Eliminating this waste factor could be beneficial in several ways. On platforms with limited available memory, such as consoles and handheld games systems, a 25% reduction in memory required to store these reduced accuracy values may be valuable. It could allow the developer to increase the amount of memory allocated to other resources, such as higher-resolution textures, or

even reduce memory requirements so that the application can run in the first place.

In a more general computing context, since fewer bytes need to be transferred for any given number of data elements, the use of a lower precision representation can have several potential benefits due to reduced use of the memory subsystem. Data transfer typically consumes a significant amount of power, so a reduction in the application's memory bandwidth requirements can reduce the application's overall power consumption. Memory-bound parts of the application can also see a reduction in transfer latency since the quantities being transferred are now smaller.

Instead of being required to use the fixed-width types provided by IEEE-754, regardless of the accuracy of the results being represented, being able to choose the size of the representation along a continuum would be very helpful, particularly when developing applications for constrained platforms.

4.3 A Scheme for Reduced-Precision Floating Point Representation

The structure of the IEEE-754 encoding of floating point numbers means that a change in an exponent bit strongly influences the resulting value, while a change in a mantissa bit has less influence. Furthermore, a change in any bit of the mantissa has exponentially greater effect on the resulting value than a change in the next-least-significant bit. These observations lead naturally to a scheme for representing values at resolutions between those specified by the IEEE-754 standard: truncation of the low-order mantissa bits.

Our 24-bit floating point thought experiment (Figure 4-1) is one of these short representations. By getting rid of the unused low 8 bits of the mantissa, we can store low accuracy results in a compact representation and free up 25% of the memory which would have been used if we stored results in full IEEE-754 single-precision format. Figure 4-2 displays some possible reduced-precision representations for IEEE-754 32-bit floating point.

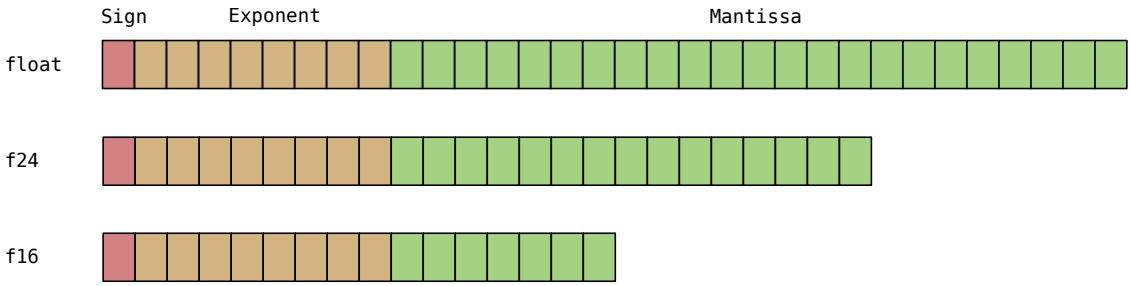


Figure 4-2: Layout of alternative multibyte floating point formats

A straightforward scheme for the use of reduced-precision storage formats is to convert values from the reduced-precision storage format into the appropriate IEEE-754 type when they are loaded. These values can then be used in computation as normal, producing results in a natively supported IEEE-754 format. If we need to store a value in reduced-precision format, we can perform a similar conversion while writing the data, encoding it in our desired storage format in memory. The principal difference between the reading and writing stages are that the datatype widens on reads, and narrows on writes.

To use this scheme in a practical application, some challenges need to be overcome. The native facilities provided by the hardware for floating-point computations will typically support only the IEEE-754 types `binary32` and `binary64`, which correspond to the C types `float` and `double`. Since we must use these types to perform computations, we have two challenges: reading and writing of our reduced precision representations.

4.3.1 Practical Multibyte Representations for IEEE-754

Our approach discards low order mantissa bits, which encode only a small portion of a represented value, keeping the sign and exponent intact.

By preserving the exponent of the existing value and adjusting only the mantissa, we can avoid having to perform a complicated conversion between number formats with different exponent sizes. Furthermore, while we could in theory discard any number of bits, by choosing to shorten the number format by multiples of 8 bits, we can represent floating point values with precision

along a continuum between the standard IEEE-754 formats without the use of complicated sub-byte data movement.

We propose a set of non-standard **fl**oating point multib**yte** types that we refer to as *flyte*s. Rather than emulating computation on *flyte*s in software, we convert them to/from the next largest natively supported type. Thus, a 24-bit *flyte* (`flyte24`) is converted to `binary32` before computation, and the `binary32` result is converted back to `flyte24` after. Table 4.1 summarizes our *flyte* storage formats and their memory layouts.

Table 4.1: Supported *flyte* storage formats for IEEE-754 types.

		Layout (bits)		
IEEE-754 format	Storage format	Sign	Exp.	Mant.
`binary32`	16-bit	1	8	7
.	24-bit	1	8	15
.	32-bit	1	8	23
`binary64`	40-bit	1	11	28
.	48-bit	1	11	36
.	56-bit	1	11	44
.	64-bit	1	11	52

4.3.2 Simple Scalar Code Approach

Figure 4-3 shows a simple implementation of the `flyte24` type in C++. It relies on the *bit field* facility in C/C++ to specify that the `num` field contains a 24-bit integer. It also uses the GCC `packed` attribute to indicate to the compiler that arrays of the type should be packed to exactly 24 bits, rather than padded to 32 bits. Figure 4-3 also shows a routine for converting from `flyte24` to `float` (i.e. `binary32`). The 24-bit pattern stored in the `flyte24` variable is scaled to 32 bits and padded with zeros. The resulting 32-bit pattern is returned as a `float`.
The code that is sketched in Fig. 4-3 can be used to implement programs with arrays of `flyte24` values, but it is very slow. Figure 4-6a shows a comparison of the execution time of several BLAS kernels using `flyte24` and other `flyte` and IEEE-754 types. The order of magnitude difference in execution time be-

```
class flyte24 {
private:
  unsigned num:24;
public:
  operator float() {
    u32 temp = num << 8;
    return(cast_u32_to_f32(temp));
  };
  ...
} __attribute__((packed));
```

Figure 4-3: Simple implementation of `flyte24` in C++

tween `flyte24` and `binary32` is the result of (1) the cost of converting between `flyte24` and `binary32` before and after each computation; and (2) the cost of loading and storing individual 24-bit values. In particular, storing data to successive elements of packed `flyte24` arrays can result in sequences of overlapping unaligned stores. Load/store hardware in GPPs is not designed to deal with such operations, which results in extraordinarily slow execution.

4.4 Reading from Reduced-Precision Representations

Reading from reduced-precision representations might be expected to incur a significant performance penalty due to the overhead of data format conversion, in addition to potentially awkward memory access to datatypes which do not correspond to any native word size.

For example, if the data format consists of packed 3-byte quantities, as in the 24-bit floating point format from our thought experiment, the sequence of operations required to load and convert one datum into a 32-bit register might run as follows.

First, load 4 consecutive bytes from the leading (byte) address of the datum, with the trailing byte belonging to the next datum. Next, shift the register if required, so that the 3-byte portion occupies the correct bit positions.

Finally, zero the trailing 8 bits of the register to encode the represented value in the corresponding IEEE-754 `binary32` number for computation. Note that

the loads will also most likely be unaligned, since the data format consists of packed 3-byte quantities.

While the overhead of conversion is indeed significant when performed in scalar code, a vectorized implementation benefits in several ways. Due to the width of vector loads, several reduced precision data values can be loaded with a single memory movement. Further, with vector reorganization instructions, we can both rearrange and zero-pad the loaded data values with very little overhead.

The scalar code versions of our BLAS Level 1 benchmarks which perform mainly reading operations, (Figure 4-6a) show the overhead incurred by reduced-precision reads in scalar code. However, as demonstrated in the vectorized versions of these benchmarks, (Figure 4-6b) the overhead of reading reduced-precision data using vector instructions to perform the conversion is of the order of a few cycles per data element.

4.5 Writing to Reduced-Precision Representations

The task of storing data to a reduced precision format is more complex than that of loading data from a reduced precision format, both in terms of the memory movement and due to the fact that while storing, we must convert values from a wider to a narrower representation. In this section, we first discuss the vectorization of the data movement, and then the task of data conversion, which chiefly involves the issue of *rounding* of values from larger to smaller number formats.

4.5.1 Vectorized I/O with Multibyte Representations

A vectorized floating point computation may require `VF` reduced-precision elements to be loaded into or stored from a single vector register. As previously outlined, these quantities may not correspond to any native type, but must be loaded from memory, converted, and packed consecutively in the lanes of a vector register for operation. Similarly, after computation has produced a

vector containing VF packed results, they must be rounded to the appropriate format and packed consecutively for storage.

Loading and storing of data in a multibyte representation which does not correspond to any native machine type presents some challenges. Typically, loads can be performed as normal, using the next largest native type, followed by the discarding of some loaded data, and the mapping of the remaining data to a legal value in the representation being used for computation. However, storing is a more difficult issue.

Since stored values do not correspond to native machine types, we cannot simply use native store instructions and reorganize later, as we can with loads, but must first *pack* the data to be stored into native types. This often leads to the splitting of data elements across separate store operations, where the leading bytes of an element are stored by one operation, and the trailing bytes are stored by the next.

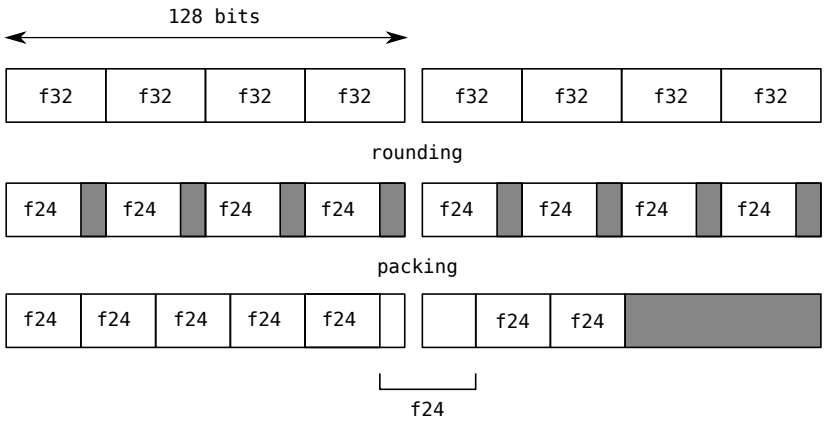


Figure 4-4: Layout of data in **SSE** vector registers before format conversion (top), after rounding to the desired size (center) and the desired layout in memory (bottom).

Figure 4-4 shows graphically the arrangement of data in registers in 32-bit floating point format after computation. The figure shows how data can be rearranged into compact 24-bit representation. Note that the desired memory layout requires data elements to straddle the boundary between vector registers.

Our approach to storing values in reduced precision format works by packing all the rounded data elements to be stored consecutively in a set of vector registers, which are mapped to a set of consecutive, *non-overlapping* vector memory movements. There are many approaches which can be used to accomplish this packing, but we follow in the vein of the previous chapter in using a two-phase `permute` and `blend` approach. This approach is shown graphically in the example in Figure 4-4. Vector permute instructions are used in the initial phase to arrange the data in each register into the required compact memory order. Next, the compacted vector registers are combined using vector blend instructions until a number of fully packed vector registers result. These registers hold the packed data to be written with non-overlapping vector stores. The trailing register of any such sequence of stores may not be full, in which case various methods such as predicated writes or a read-modify-write sequence can be used to ensure that the data is stored correctly.

4.5.2 Rounding

When numbers represented in IEEE-754 floating point format are used in computations (such as addition and subtraction), the natural result of computation is often a real number which is not exactly representable in the finite representation. In these cases, the standard specifies a way to round these numbers to the nearest representable value using the encoding scheme previously outlined.

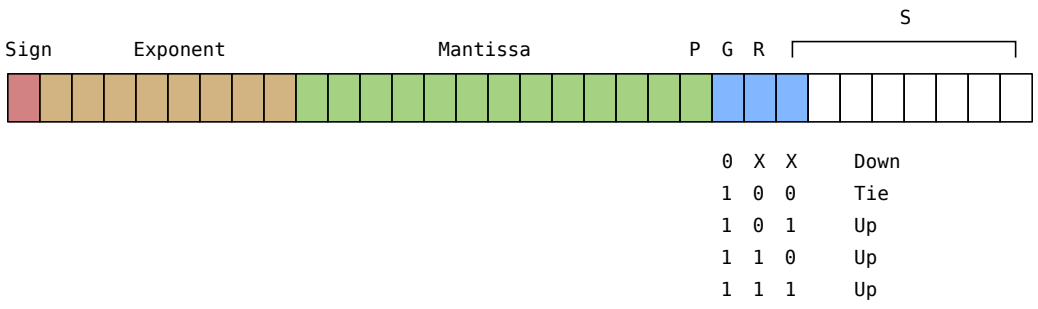


Figure 4-5: Rounding to nearest even. Marked bit positions correspond to **P**reguard, **G**uard, **R**ound, and **S**ticky bits.

Typically, floating point calculations are performed using more bits than are present in the respresentation, and results are rounded to fit within the constraints of the number format. With some care, we can exploit the native capabilities of the hardware to perform rounding of any number of bits.

Figure 4-5 shows a 32-bit floating point number being rounded to our 24-bit representation. As a measure to reduce error resulting from rounding of results, the IEEE standard specifies rounding of the mantissa to the nearest even number as the default rounding mode. Rounding to nearest even is expressed in terms of particular bits around the boundary where bits are discarded. The bit position which becomes the new least significant bit is known as the guard bit, while the first bit lost is known as the round bit.

When the guard bit is zero, the number for which it is the least significant bit is already even, and nothing needs to be done - the round bit and all less significant bits are discarded. When the guard bit is one, if either or both of the round or sticky bits are one, we must round up. The round bit is a single position, but the sticky bit is the logical OR of all lower bits. In the case where the round and sticky bits are both zero and the guard bit is one, we have a tie, and must inspect the bit preceding the guard bit to decide whether to round up or down.

This rounding mechanism is part of the IEEE-754 standard, and is therefore implemented in all IEEE-754 compliant hardware systems. While we could use a software implementation of this rounding approach, we can also utilize the existing hardware facilities to perform rounding of any number of bits by performing some simple arithmetic operations which have no effect other than causing the desired rounding to happen. Some of these operations are summarized in Appendix A.

Prior work on reduced-precision floating point data storage by Jenkins et al. 2012 does not perform an explicit rounding step, but directly truncates values. This is equivalent to another standard IEEE-754 rounding mode, namely, rounding towards zero. Jenkins et al. evaluate the error introduced by rounding towards zero and find it acceptable for their purposes. However, using

our approach, the rounding step in Figure 4-4 can be implemented in any way which is suitable for the needs of the application, including rounding towards zero, and rounding to nearest even.

4.5.3 Treatment of Special Values

The IEEE-754 floating point standard has special values and ranges as previously outlined in Section 4.1. Rounding interacts in different ways with these values and ranges, and behaviour which may be appropriate for some scenarios may not be for others.

NaN values

NaN values represent the result of expressions of *indeterminate form*, which cannot be computed, such as $\infty - \infty$. The expected behaviour of format conversion of a NaN value is a NaN value in the target format. However, NaN is not a singular value, but a value *range*. NaN values occupy a range of bit patterns distinguished by an exponent which is all 1s and any nonzero value in the mantissa. Since the mantissa is truncated by conversion to a shorter format, some NaN values cannot be represented after down-conversion.

NaNs may be either signalling or non-signalling. Signalling NaNs cause an invalid operation exception to be raised when they are produced, while quiet NaNs do not raise any additional exceptions. However, in IEEE-754 binary floating point formats, quiet NaNs are distinguished from signalling NaNs by the value of the *most significant bit* of the mantissa, which is preserved by truncation of up to $M - 1$ bits.

Infinities

Positive and negative infinities are encoded with an exponent which is all 1s and a zero mantissa. There are only two values in this class, which are distinguished by their sign. The expected behaviour of format conversion for infinities is a correctly signed infinity in the target format.

Subnormal numbers

Subnormal numbers are distinguished by a zero exponent, and lack the implied leading 1 in their mantissa. They represent numbers very close to zero. The expected behaviour of format conversion for subnormal numbers is slightly complicated due to the issues of overflow and underflow. The closest value in the target representation for a very large subnormal number may be a very small normalized number (overflow), while the closest value for a very small subnormal number may be zero (underflow). For subnormal numbers, rounding may validly cause the class of the value to change, either by underflow to zero, or by overflow to a small normalized number.

Normalized numbers

When a normalized number is being rounded, overflow to infinity occurs when the rounded value is so large that the exponent is all ones after rounding (overflow). This is a natural consequence of conversion from a larger to a smaller finite representation. However, when a normalized number is so small that its exponent is all zeros after rounding (underflow), it does not get rounded directly to zero, but instead to a subnormal number. Underflow is *gradual*, and a number will underflow to zero only when it is so small that both exponent and mantissa are all zeros after rounding. Normalized numbers may therefore naturally be rounded to several different classes of value. Very large positive or negative numbers may go to infinity when rounded, and very small numbers may become subnormal.

4.6 Performance Evaluation

This section presents the results of performance evaluation of vectorized code generated by the technique presented. The approach we proposed in this chapter for vectorized conversion to and from multibyte floating point representations was implemented in a library of pseudo-intrinsic functions, which internally use the platform's vector intrinsics to perform the conversions.

We vectorized a selection of BLAS operations by hand, and evaluated their performance when using reduced-precision memory representations versus native memory representations. Our objective is to assess the overheads of conversion to and from reduced precision formats, and determine whether they outweigh the overall performance gains from reducing memory traffic.

The rounding method used in these benchmarks was round towards zero.

4.6.1 BLAS Level 1 Evaluation

We vectorized several BLAS Level 1 programs using our approach to apply the BLAS kernel operation to data in each of the supported multibyte floating point representations presented in Table 4.1. Computation is performed using 32-bit `float` for formats `flyte16` and `flyte24`, and using 64-bit `double` for formats `f40`, `f48`, and `f56`. We measured the total running time of programs performing the same number of internal `float` or `double` operations, with conversion on loads and stores. In the figures, input size on the x-axis is the number of elements in a BLAS vector (not to be confused with SIMD vectors).

We produced two version of every program, one "library version" which is scalar code, and one "SSE version" which is vectorized using SSE operations. For `float` the vectorization factor is always 4, while for `double` it is 2. Both the scalar and vector versions of benchmarks perform exactly the same rounding operations. We would expect the vectorized `double` code to perform more slowly than the vectorized `float` code, due to the difference in the vectorization factor. This expectation is borne out in practice.

One feature which immediately becomes clear from experimentation is that conversion is very costly when performed in scalar code using scalar data movement (Figure 4-6a). For the scalar code versions of benchmarks, the overhead of conversion is large enough to completely negate the performance gains from the reduction in memory traffic for all the BLAS Level 1 kernels we used in experiments.

In-Place Operations

The BLAS Level 1 `scal` operation is an in-place transformation, which requires format conversion on both loads and stores when using our approach. However, when vectorized, the availability of vector data reorganization instructions gives a significant boost to the performance of format conversion operations. Figure 4-6b displays this clearly, with the performance of BLAS in-place vector scale on our `flyte16` representation very nearly matching the performance of BLAS vector scale on native `double` arrays.

Reductions

The `min`, `max`, and `magnitude` benchmarks perform a horizontal reduction. Due to the access pattern, which reduces an array to a single value, format conversion is only required on the load side, and the result is produced as a single `float` or `double` value. The `min` and `max` benchmarks display a clear separation between the performance of `float` and `double` operations in the vectorized code, which were only narrowly separated in the `scal` benchmark. In both of these benchmarks, the performance of the vectorized code for our `flyte16` and `flyte24` types exceeds the performance of native `double` operations, but is still slower than native `float` operations.

The `magnitude` benchmark computes the 2-dimensional Euclidean norm of the input vector, which is computationally a much heavier operation than `min` and `max`. This benchmark shows a very tight clustering of execution times for each of the representations corresponding to `double` and each of the representations corresponding to `float`, with error bars overlapping in many places. This kernel performs enough computation to effectively hide the overheads of format conversion, allowing us to perform the operation on arrays of `flyte16` and `flyte24` data as quickly as if they were stored in the native `float` representation.

The `dot − product` benchmark also performs a reduction, but reduces two arrays as opposed to the single array reduced in `min`, `max`, and `magnitude`. This causes the program to create twice as much memory traffic as the other re-

ductions. Even though the program is not computationally as heavy as the `magnitude` benchmark, the savings on memory traffic from the use of reduced precision data representations is enough that only a small penalty is incurred for `flyte16` and `flyte24` representations versus `float`. The penalty incurred for `double` representations is larger due to the increased overall memory traffic and the decrease in vectorization factor from 4 to 2 due to the constraints of the native hardware operations available.

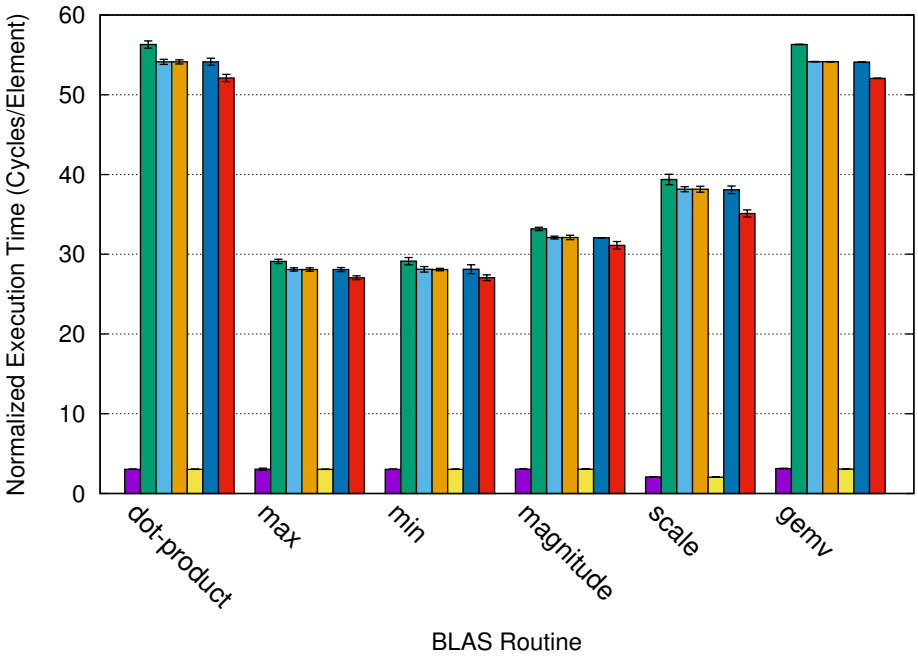
4.6.2 BLAS Level 2 Evaluation

We evaluated our approach when applying the BLAS Level 2 matrix-vector kernel `SGEMV` to data in our proposed multibyte floating point representations. Similar to the experimental data for the Level 1 kernels, when format conversion is performed in scalar code, the overheads very quickly offset the savings on memory transfer (Figure 4-6a).

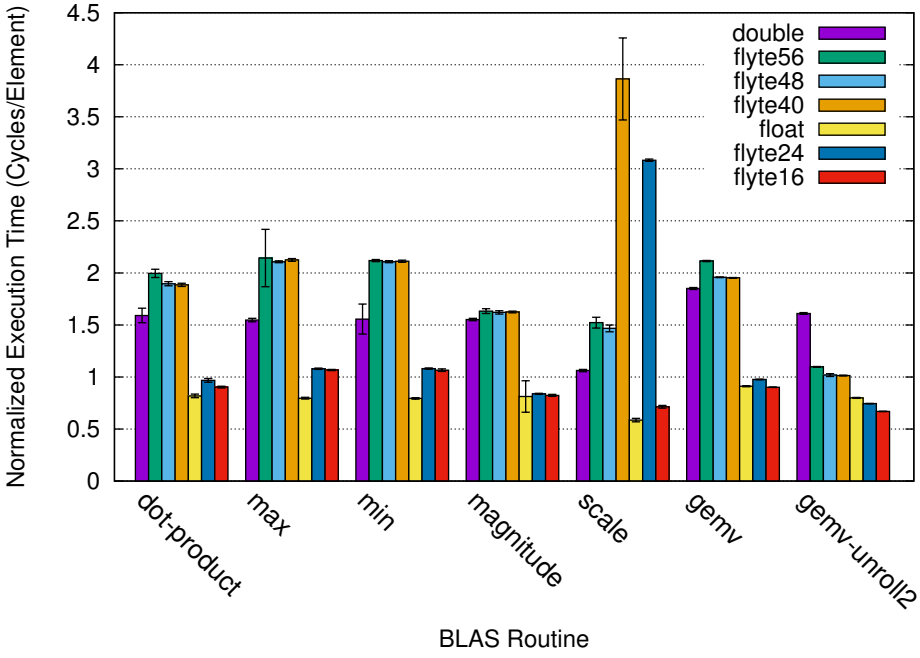
Using our vectorized approach, the performance gap is closed. Figure 4-6b shows that the data formats corresponding to `double` result in such a large savings on memory transfers that the application runs faster despite the overheads of format conversion. The data formats corresponding to `float` also show an improvement, although it is less pronounced due to the lower overall volume of data transferred.

Unrolling

Unrolling in combination with vectorization can be used to achieve even more performance gain when the vectorized program is not making full use of the memory system. Benchmark `gemv` $-$ `unroll2` from Figure 4-6b demonstrates the result of unrolling the `gemv` benchmark so that two vectorized loop iterations are performed together. The benchmark shows a clear separation between the datatypes, with the execution time for each datatype increasing with the size of the type. This demonstrates that the saving on memory transfer time is more than enough to compensate for the overhead of the format conversion operations.



(a) BLAS Level 1 Kernels, scalar code, problem size 8K



(b) BLAS Level 1 Kernels, vectorized (SSE), problem size 8K

Figure 4-6: Real-world benchmarks

4.7 Related Work

4.7.1 The Approach of Jenkins et al.

A variant of this scheme has previously been outlined by Jenkins et al. 2012, who introduce the notion of a *component vector* representation, which treats a floating point number as a vector of multibyte components, which, when concatenated, form the original full-precision value.

Jenkins et al. treat an array of floating point numbers as a *matrix*, where each row holds a single floating point number, and the columns of the matrix partition the value along byte boundaries. By storing the matrix in transposed order, *truncated* versions of every number are kept close together in memory, enabling faster access to reduced precision variants of each number.

If access to the full precision version of data is required, an inverse transpose operation can reconstruct the original data, though the cost of this operation grows with the data set size.

By varying the width in bytes of each matrix column, Jenkins et al. can statically vary the precision at which data is available. Dynamically altering the precision is possible, but expensive, requiring an inverse transpose of the data, followed by a subsequent transpose choosing different byte widths for each column.

Jenkins et al. implement their transposed approach to reduced-precision storage using MPI [32], and evaluate it in an extreme-scale computing context, using large-scale scientific applications GTS [83], S3D [19], and XGC-1 [39].

Our approach has some material differences from the approach proposed by Jenkins et al. First, we do not store data in column-transposed order, but instead store the reduced precision values packed consecutively in memory in the same order as in the original data layout.

Second, we support several different methods for reducing the precision of data. The truncation approach of Jenkins et al. is equivalent to a standard IEEE-754 round-towards-zero rounding mode. However, depending on the needs of the application, different rounding strategies may be required. For

example, fixed rounding (either always rounding up or rounding down) or other standard modes such as rounding to nearest even. Our approach allows the programmer to select the method for reducing precision according to their needs.

Third, and perhaps most importantly, our approach does not use MPI, but has a low-level vectorized implementation. Jenkins et al. identify a low-level vectorized approach as a promising direction for future work. We demonstrate that with this approach, we can obtain performance improvements from reduced precision representations using single-core SIMD execution on a general purpose processor.

4.8 Conclusion

In this chapter, we propose a method for conversion of floating-point data between different resolutions that uses vectorization to accelerate the conversion and amortize overheads across multiple data accesses. Our proposed approach supports the use of fine-grained multibyte data formats, and is not limited to the available native machine types. We also demonstrate that native hardware capabilities can be used to support format conversions on non-native multibyte data formats with very little overhead.

We show experimentally that reduced precision data representations can be leveraged to improve the performance of floating-point computations outside of a supercomputing context using single-core SIMD operation on modest shared memory multiprocessor systems. Our experimental evaluation is carried out using a modified implementation of BLAS Level 1 and 2 to support operation on reduced-precision floating point types. Internally, all operations are performed using regular IEEE-754 floating point types, but inputs are consumed and outputs produced in several reduced-precision representations. We find that this approach can be supported with very low overhead, and in some cases, can result in a speedup from the use of reduced precision types for storage.

Chapter 5

Conclusion and Final Thoughts

In this thesis, we have demonstrated that significant speedup can be obtained from automatic vectorization of interleaved accesses at a variety of strides, many of which have previously been considered to require irregular or hand-tailored solutions. We have also shown that efficient vectorized code can be generated to accelerate conversions between native datatypes and custom data formats. Both of these pieces of research demonstrate that careful vectorization of memory access can result in significant performance gain.

5.1 Future Work

Several directions for future work present themselves on foot of the research presented in earlier chapters.

5.1.1 Generalized Interleaved Access

Possible extensions to our approach to interleaved access include relaxing the constraint on equivalent unit sizes in an access group (Section 3.2). This would enable our approach when dealing with complex array-of-structures memory layouts where adjacent structure fields are of different widths. Analysis and code generation for this use case appears significantly more involved, but it seems plausible that performance gain in this scenario is possible. When dealing, for example, with complex audio and video data formats, a robust compile-

time mechanism for generating SIMD data layout transformations would relieve programmers of a lot of implementation effort and hand-optimization.

5.1.2 Runtime Code Generation

It would also be interesting to investigate the potential for using our techniques to generate code at run-time, for accesses whose stride is constant, but is not known at compile time. As more and more high-level interpreted languages add explicit SIMD features to their programming model [34], the demands placed on JIT compilers regarding vectorization will surely grow. Systems have already been proposed for generating vectorized code in JIT compilers [69, 56] which could be extended with a run-time implementation of our code generation techniques to generate SIMD code for accesses with arbitrary stride. Whether an implementation of the techniques proposed would be fast enough for the Just-In-Time context is an open question.

5.1.3 Finer Granularity of Storage Formats

Although for practical reasons we restricted the storage formats we consider in Chapter 4 to multiples of 8 bits in length, it is possible in principle to support formats where any number of bits have been truncated, from a single bit up to the length of the mantissa. It would be interesting to investigate support for bit-granularity storage formats to increase the flexibility of the approach. However, whether efficient vectorized code can be generated to support sub-byte precision formats remains an open question.

5.2 Final Thoughts

As the SIMD capabilities available on general purpose processors become more advanced, and the SIMD width increases, it will become even more important that compilers exploit these features to accelerate programs. In their review of the capabilities of vectorizing compilers, Maleki et al. lament the fact that many of the difficulties they encountered have published solutions, but those solutions are not all present in any one compiler [49]. Frameworks like the polyhedral model, which are designed to bring together many different transformations within a single representation, seem like a step in the right direction to reduce this fragmentation of techniques between different implementations.

Schaub et al. have found that as SIMD width increases, memory access, which is already expensive in terms of time and energy, becomes less uniform and less consecutive [72]. As processor manufacturers continue to add more SIMD features, and programming languages add more explicit-SIMD programming constructs, applications will place increasing demands on SIMD hardware. Although many patterns of memory access beyond strided array access are already handled by special-purpose compilation systems such as SPIRAL [27], given the findings of Schaub et al. and the global trend of increasing SIMD width, it seems likely that general-purpose vectorizing compilers will be expected to generate performant SIMD code for increasingly difficult memory access patterns.

Appendices

Appendix A

Identities for Rounding

We state several identities on floating point numbers represented in IEEE-754 binary formats which have the effect of rounding a floating point value v to the nearest representable value v_r with the same exponent, but a mantissa truncated by a given number of bits, b. These identities make use of the fact that an implementation of IEEE-754 naturally performs rounding after certain arithmetic instructions to obtain a representable result. Although algebraically they are identities, these computations have the side-effect of causing results to be rounded to a particular bit position.

None of the identities presented are totally correct, but the mode of their incorrect behaviour is described in the **Behaviour** subsection for each identity. All of the identities work correctly for regular normalized and subnormal numbers, but they may mistreat infinities and NaN values. In addition, some of the identities cause more overflows than others; this is also addressed in the description of the identities.

A.0.1 Arithmetic Rounding I

$$v_b = v \times 2^b$$

$$v_r = v + v_b - v_b$$

This identity hinges on the requirement that, specifically, the upper $M - b$ bits of the M-bit mantissa, which will form the truncated mantissa, are correctly rounded. The structure of the IEEE-754 binary floating point representation means that to multiply a number by a power of two, only the exponent of the value (which encodes a power of two by which to multiply the mantissa) needs to be changed. When two values are added which differ only in their exponent, the mantissa of the smaller value is effectively shifted to the right by a number of bits equivalent to the difference in exponents during addition. When the result is rounded, some bits are lost to round-off error. By choosing b as the power of two, we ensure that the addition $v + v_b$ loses at most as many bits as we plan to truncate. After addition, the correctly rounded, normalized value $v + v_b$ then has v_b subtracted from it, which has the effect of shifting the radix point to the left so that the high bits of the original mantissa, which are now correctly rounded, are back in their original place. We obtain a value close to the original value, but with b mantissa bits rounded off, and the remaining mantissa bits correctly rounded by the implementation.

Behaviour

While this approach handles subnormal numbers well, the multiplication by 2^b causes large positive and negative normalized numbers to overflow to infinity. Although correct, if overflows are a concern for the application, other approaches proposed may result in fewer overflows. The approach also causes infinities to be rounded to a NaN value, since the operation, when $v = \pm\infty$, causes the mantissa to become nonzero after rounding.

A.0.2 Arithmetic Rounding II

To reduce the space of values for which overflow occurs at the multiplication step, we can choose a different identity which multiplies by a much smaller number than 2^b.

$$v_b = v \times (1 + 2^{-b})$$
$$v_r = (v_b - v) \times 2^b$$

This approach exploits the fact that underflow is *gradual* in IEEE-754, meaning that a number which underflows will become subnormal, and finally underflow to zero, as opposed to overflow, which causes the result to immediately become infinity. It is acceptable that a very small number may underflow during rounding and be rounded to a large subnormal number. Such a number can still be used in computation to produce a result. However, a number which overflows to infinity during rounding cannot be used to produce a meaningful result from computation.

Behaviour

Although this approach causes fewer overflows at the multiplication step, due to the factor used being much smaller than 2^b, it still causes infinities to be rounded to NaN values.

A.0.3 Arithmetic Rounding III

To avoid the overflow of numbers entirely, we can make sure that we initially multiply by a number which is less than 1, and so the original value is never increased. This biases the rounding process towards underflow, and leads to higher underestimation error than the previously proposed approaches. However, we can avoid infinities resulting from rounding.

$$v_b = v \times (1 - 2^{-b})$$
$$v_r = (v - v_b) \times 2^b$$

Behaviour

Although this approach results in fewer overflows than the two approaches previously stated, it causes more underflows. In addition, it still mistreats infinities, causing them to be rounded to NaN values.

A.0.4 Arithmetic Rounding IV

A very simple rounding approach is to perform an *unsigned integer addition* of a constant consisting of $b-1$ 1s in the low order bits and 0 in all higher bits.

$$v_r = v + (2^b - 1)$$

Behaviour

This approach is attractive because of its simplicity. However, unlike the previously proposed approaches, we are not exploiting the native floating point hardware to perform rounding to nearest even, which results in ties being rounded down. In addition, infinities are still incorrectly rounded to NaN values by the approach.

Bibliography

[1] *3D Vector Normalization Using 256-Bit Intel®Advanced Vector Extensions (Intel®AVX)*. `http://software.intel.com/en-us/articles/3d-vector-normalization\\-using-256-bit-intel-advanced-vector-extensions-intel-avx`. 2013.

[2] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. "Code generation using tree matching and dynamic programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11.4 (1989), pp. 491–516.

[3] John R Allen and Ken Kennedy. *PFC: A program to convert Fortran to parallel form*. Rice University. Department of Mathematical Sciences, 1982.

[4] Randy Allen and Ken Kennedy. "Automatic translation of Fortran programs to vector form". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.4 (1987), pp. 491–542.

[5] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[6] Andrew Anderson, Avinash Malik, and David Gregg. "Automatic Vectorization of Interleaved Data Revisited". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.4 (2015), p. 50.

[7] Jason Ansel et al. "Language and compiler support for auto-tuning variable-accuracy algorithms". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 85–96.

[8] David F Bacon, Susan L Graham, and Oliver J Sharp. "Compiler transformations for high-performance computing". In: *ACM Computing Surveys (CSUR)* 26.4 (1994), pp. 345–420.

[9] Woongki Baek and Trishul M Chilimbi. "Green: a framework for supporting energy-conscious programming using controlled approximation". In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 198–209.

[10] A Balachandran, Dhananjay M. Dhamdhere, and S Biswas. "Efficient retargetable code generation using bottom-up tree pattern matching". In: *Computer Languages* 15.3 (1990), pp. 127–140.

[11] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. "Efficient Selection of Vector Instructions Using Dynamic Programming". In: *MICRO*. IEEE, 2010, pp. 201–212.

[12] Aart JC Bik et al. "Automatic intra-register vectorization for the Intel® architecture". In: *International Journal of Parallel Programming* 30.2 (2002), pp. 65–98.

[13] Uday Bondhugula et al. "A practical automatic polyhedral parallelizer and locality optimizer". In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 101–113.

[14] Uday Bondhugula et al. "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *Compiler Construction*. Springer. 2008, pp. 132–146.

[15] Patricio Bulić and Veselko Guštin. "An extended ANSI C for processors with a multimedia extension". In: *International Journal of Parallel Programming* 31.2 (2003), pp. 107–136.

[16] Alfredo Buttari et al. "Exploiting Mixed Precision Floating Point Hardware in Scientific Computations." In: *High Performance Computing Workshop*. 2006, pp. 19–36.

[17] Hoseok Chang and Wonyong Sung. "Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware". In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2008, pp. 167–176.

[18] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. "Static Data Race Detection for SPMD Programs via an Extended Polyhedral Representation". In: *IMPACT* (2016).

[19] Jacqueline H Chen et al. "Terascale direct numerical simulations of turbulent combustion using S3D". In: *Computational Science & Discovery* 2.1 (2009), p. 015001.

[20] Gerald Cheong and Monica Lam. "An optimizer for multimedia instruction sets". In: *Contract* 30602.95-C (1997), p. 0098.

[21] Albert Cohen, Sylvain Girbal, and Olivier Temam. "A Polyhedral Approach to Ease the Composition of Program Transformations". In: *Euro-Par*. Ed. by Marco Danelutto, Marco Vanneschi, and Domenico Laforenza. Vol. 3149. Lecture Notes in Computer Science. Springer, 2004, pp. 292–303.

[22] Robert Cypher and Jorge LC Sanz. "SIMD architectures and algorithms for image processing and computer vision". In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 37.12 (1989), pp. 2158–2174.

[23] Keith Diefendorff and Pradeep K Dubey. "How multimedia workloads will change processor design". In: *Computer* 30.9 (1997), pp. 43–45.

[24] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. "Vectorization for SIMD architectures with alignment constraints". In: *PLDI*. Ed. by William Pugh and Craig Chambers. ACM, 2004, pp. 82–93.

[25] Nadeem Firasta et al. "Intel avx: New frontiers in performance improvements and energy efficiency". In: *Intel white paper* (2008).

[26] Liza Fireman, Erez Petrank, and Ayal Zaks. "New algorithms for SIMD alignment". In: *Compiler Construction*. Springer. 2007, pp. 1–15.

[27] Franz Franchetti and Markus Püschel. "A SIMD vectorizing compiler for digital signal processing algorithms". In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE. 2001.

[28] Franz Franchetti and Markus Püschel. "Generating SIMD Vectorized Permutations". In: *CC*. Ed. by Laurie J. Hendren. Vol. 4959. Lecture Notes in Computer Science. Springer, 2008, pp. 116–131.

[29] Christopher W Fraser, Robert R Henry, and Todd A Proebsting. "BURG: fast optimal instruction selection and tree parsing". In: *ACM Sigplan Notices* 27.4 (1992), pp. 68–76.

[30] Tobias Grosser et al. "Polly-Polyhedral optimization in LLVM". In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Vol. 2011. 2011.

[31] Jie Han and Michael Orshansky. "Approximate computing: An emerging paradigm for energy-efficient design". In: *Test Symposium (ETS), 2013 18th IEEE European*. IEEE. 2013, pp. 1–6.

[32] Rolf Hempel. "The MPI standard for message passing". In: *High-Performance Computing and Networking*. Springer. 1994, pp. 247–252.

[33] John Jenkins et al. "Byte-precision level of detail processing for variable precision analytics". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE. 2012, pp. 1–11.

[34] Ivan Jibaja. *SIMD in Javascript*. 2014.

[35] Ralf Karrenberg and Sebastian Hack. "Whole-function vectorization". In: *CGO*. IEEE, 2011, pp. 141–150.

[36] Leslie Kohn et al. "The visual instruction set (VIS) in UltraSPARC". In: *compcon*. IEEE. 1995, p. 462.

[37] Christoforos Kozyrakis and David Patterson. "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks". In: *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press. 2002, pp. 283–293.

[38] Andreas Krall and Sylvain Lelait. "Compilation techniques for multimedia processors". In: *International Journal of Parallel Programming* 28.4 (2000), pp. 347–361.

[39] S Ku, CS Chang, and PH Diamond. "Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry". In: *Nuclear Fusion* 49.11 (2009), p. 115021.

[40] Alexei Kudriavtsev and Peter Kogge. "Generation of permutations for SIMD processors". In: *SIGPLAN Not.* 40.7 (June 2005), pp. 147–156. ISSN: 0362-1340.

[41] Michael O Lam et al. "Automatically adapting programs for mixed-precision floating-point computation". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 369–378.

[42] Samuel Larsen and Saman P. Amarasinghe. "Exploiting superword level parallelism with multimedia instruction sets". In: *PLDI*. Ed. by Monica S. Lam. ACM, 2000, pp. 145–156.

[43] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[44] Corinna G Lee and Derek J DeVries. "Initial results on the performance and cost of vector microprocessors". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society. 1997, pp. 171–182.

[45] Corinna G Lee and Mark G Stoodley. “Simple vector microprocessors for multimedia applications”. In: *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press. 1998, pp. 25–36.

[46] Ruby B Lee. “Accelerating multimedia with enhanced microprocessors”. In: *iEEE Micro* 15.2 (1995), pp. 22–32.

[47] Roland Leißa, Sebastian Hack, and Ingo Wald. “Extending a C-like language for portable SIMD programming”. In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 65–74.

[48] Jun Liu et al. “A compiler framework for extracting superword level parallelism”. In: *PLDI*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 347–358.

[49] Saeed Maleki et al. “An Evaluation of Vectorizing Compilers”. In: *PACT*. Ed. by Lawrence Rauchwerger and Vivek Sarkar. IEEE Computer Society, 2011, pp. 372–382.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[51] Dorit Naishlos. “Autovectorization in GCC”. In: *Proceedings of the 2004 GCC Developers Summit*. 2004, pp. 105–118.

[52] Dorit Nuzman and Richard Henderson. “Multi-platform auto-vectorization”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2006, pp. 281–294.

[53] Dorit Nuzman, Ira Rosen, and Ayal Zaks. “Auto-vectorization of interleaved data for SIMD”. In: *PLDI*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 132–143.

[54] Dorit Nuzman and Ayal Zaks. “Autovectorization in GCC–two years later”. In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, pp. 145–158.

[55] Dorit Nuzman and Ayal Zaks. “Outer-loop vectorization: revisited for short simd architectures”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 2–11.

[56] Dorit Nuzman et al. “Vapor SIMD: Auto-vectorize once, run everywhere”. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 151–160.

[57] David A Padua and Michael J Wolfe. “Advanced compiler optimizations for supercomputers”. In: *Communications of the ACM* 29.12 (1986), pp. 1184–1201.

[58] Gabriele Paoloni. “How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures”. In: (2010).

[59] Yongjun Park et al. “SIMD defragmenter: efficient ILP realization on data-parallel architectures”. In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. Ed. by Tim Harris and Michael L. Scott. ACM, 2012, pp. 363–374.

[60] Alex Peleg and Uri Weiser. “MMX technology extension to the Intel architecture”. In: *Micro, IEEE* 16.4 (1996), pp. 42–50.

[61] Alex Peleg, Sam Wilkie, and Uri Weiser. “Intel MMX for multimedia PCs”. In: *Communications of the ACM* 40.1 (1997), pp. 24–38.

[62] Matt Pharr and William R Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *Innovative Parallel Computing (InPar), 2012*. IEEE. 2012, pp. 1–13.

[63] Sebastian Pop et al. “GRAPHITE: Polyhedral analyses and optimizations for GCC”. In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, p. 2006.

[64] Vasileios Porpodas, Alberto Magni, and Timothy M Jones. "PSLP: padded SLP automatic vectorization". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 190–201.

[65] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[66] Venu Gopal Reddy. "Neon technology introduction". In: *ARM Corporation* (2008).

[67] Gang Ren, Peng Wu, and David Padua. "A preliminary study on the vectorization of multimedia applications for multimedia extensions". In: *Languages and Compilers for Parallel Computing*. Springer, 2004, pp. 420–435.

[68] Gang Ren, Peng Wu, and David Padua. "Optimizing data permutations for SIMD devices". In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 118–131.

[69] Erven Rohou et al. "Speculatively vectorized bytecode". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM. 2011, pp. 35–44.

[70] Kenneth Rosen. *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill, 2011.

[71] Cindy Rubio-González et al. "Precimonious: Tuning assistant for floating-point precision". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. 2013, p. 27.

[72] Thomas Schaub et al. "The impact of the SIMD width on control-flow and memory divergence". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015), p. 54.

[73] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures”. In: *IEEE PACT*. IEEE Computer Society, 2002, pp. 45–55.

[74] Jaewook Shin, Jacqueline Chame, and Mary W Hall. “Exploiting superword-level locality in multimedia extension architectures”. In: *J. Instr. Level Parallel* 5 (2003), pp. 1–28.

[75] Jaewook Shin, Mary Hall, and Jacqueline Chame. “Superword-level parallelism in the presence of control flow”. In: *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society. 2005, pp. 165–175.

[76] Jaewook Shin, Mary W Hall, and Jacqueline Chame. “Evaluating compiler technology for control-flow optimizations for multimedia extension architectures”. In: *Microprocessors and Microsystems* 33.4 (2009), pp. 235–243.

[77] Jan Sjödin et al. “Design of graphite and the Polyhedral Compilation Package”. In: *GCC Developers' Summit*. 2009.

[78] Kevin Stock, Louis-Noël Pouchet, and P Sadayappan. “Using machine learning to improve automatic vectorization”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8.4 (2012), p. 50.

[79] Deependra Talla, Lizy Kurian John, and Doug Burger. “Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements”. In: *Computers, IEEE Transactions on* 52.8 (2003), pp. 1015–1031.

[80] K. Trifunovic et al. “Polyhedral-Model Guided Loop-Nest Auto-Vectorization”. In: *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*. 2009, pp. 327–337.

[81] Konrad Trifunovic et al. “Graphite two years after: First lessons learned from real-world polyhedral compilation”. In: *GCC Research Opportunities Workshop (GROW'10)*. 2010.

[82] Endong Wang et al. “Intel Math Kernel Library”. In: *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[83] WX Wang et al. “Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments”. In: *Physics of Plasmas (1994-present)* 13.9 (2006), p. 092505.

[84] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. “Automated empirical optimizations of software and the ATLAS project”. In: *Parallel Computing* 27.1 (2001), pp. 3–35.

[85] Michael E Wolf and Monica S Lam. “A loop transformation theory and an algorithm to maximize parallelism”. In: *Parallel and Distributed Systems, IEEE Transactions on* 2.4 (1991), pp. 452–471.

[86] Michael Joseph Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[87] Peng Wu, Alexandre E Eichenberger, and Amy Wang. “Efficient SIMD code generation for runtime alignment and length conversion”. In: *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*. IEEE. 2005, pp. 153–164.

[88] Peng Wu et al. “An integrated simdization framework using virtual vectors”. In: *Proceedings of the 19th annual international conference on Supercomputing*. ACM. 2005, pp. 169–178.

[89] Jianxin Xiong et al. “SPL: A language and compiler for DSP algorithms”. In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 298–308.

[90] Hans P. Zima and Barbara M. Chapman. *Supercompilers for parallel and vector computers*. ACM Press frontier series. Addison-Wesley, 1990, pp. I–XV, 1–376.

[91] Dan Zuras et al. “Ieee standard for floating-point arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1–70.