A Compiler Approach for Exploiting Partial SIMD Parallelism

HAO ZHOU, UNSW Australia/NUDT, China
JINGLING XUE, UNSW Australia

Existing vectorization techniques are ineffective for loops that exhibit little loop-level parallelism but some limited superword-level parallelism (SLP). We show that effectively vectorizing such loops requires partial vector operations to be executed correctly and efficiently, where the degree of partial SIMD parallelism is smaller than the SIMD datapath width. We present a simple yet effective SLP compiler technique called PAVER (PArtial VEctorizeR), formulated and implemented in LLVM as a generalization of the traditional SLP algorithm, to optimize such partially vectorizable loops. The key idea is to maximize SIMD utilization by widening vector instructions used while minimizing the overheads caused by memory access, packing/unpacking, and/or masking operations, without introducing new memory errors or new numeric exceptions. For a set of 9 C/C++/Fortran applications with partial SIMD parallelism, PAVER achieves significantly better kernel and whole-program speedups than LLVM on both Intel's AVX and ARM's NEON.

CCS Concepts: • **Software and its engineering** → *Source code generation*

Additional Key Words and Phrases: Basic block vectorization, SLP vectorization, loop vectorization, partial vectorization, partial SIMD parallelism

ACM Reference Format:
Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. ACM Trans. Archit. Code Optim. 13, 1, Article 11 (March 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2886101

1. INTRODUCTION

Modern CPUs are equipped with SIMD units that operate on fixed-length short vectors, such as SSE/AVX for Intel CPUs and NEON for ARM CPUs, to enable high performance. However, programming SIMD units by hand is tedious and error prone, and produces nonportable code. Therefore, production-quality compilers such as ICC, GCC, and LLVM perform automatic vectorization by translating sequences of scalar operations, typically found inside loops, into vector instructions.

Two principal vectorization techniques exist for extracting the SIMD parallelism from a loop: (1) loop-level vectorization and (2) basic block or superword-level parallelism (SLP) vectorization. Loop-level vectorization, which employs technology previously developed for vector processors [Zima and Chapman 1991], exploits loop-level parallelism by combining multiple occurrences of the same scalar operation across consecutive loop iterations into a vector instruction. SLP vectorization [Larsen and Amarasinghe 2000] exploits SLP by replacing groups of isomorphic operations in a basic block with a vector instruction. In both cases, the central challenge is translating

This work is supported by the Australian Research Council under grant DP110104628.
Authors' addresses: H. Zhou, School of Computer Science and Engineering, UNSW Australia, NSW 2052, Australia and School of Computer, NUDT, Changsha, 410073, China; email: haozhou@cse.unsw.edu.au; J. Xue, School of Computer Science and Engineering, UNSW Australia, NSW 2052, Australia; email: jingling@cse.unsw.edu.au.

DOI: http://dx.doi.org/10.1145/2886101

SIMD resources into real application performance. In general, loops with regular computations and few dependences perform well on most SIMD architectures, as they exhibit an abundance of regular SIMD parallelism. Given an *m*-way SIMD unit, such loops can often be executed in terms of *m*-way, i.e., *full vector operations* on the entire SIMD datapath. Thus, the parallel resources of a SIMD unit are maximized.

In multimedia and scientific applications, however, there are many time-consuming, *hot* loops that exhibit little loop-level parallelism but some limited SLP with its degree being smaller than the SIMD datapath width. Existing techniques are ineffective in harnessing such partial SIMD parallelism, as they generate suboptimal code or give up vectorization entirely, leaving SIMD resources underutilized. Several obstacles that limit their capabilities are data dependences, nonconsecutive memory accesses (NMA), and complex control flow [Maleki et al. 2011]. Loop vectorizers [Bik et al. 2002; Eichenberger et al. 2004; Kong et al. 2013; Nuzman et al. 2006; Sreraman and Govindarajan 2000; Sujon et al. 2013] are usually unsuccessful, as the same scalar operation executed across consecutive loop iterations often touches nonconsecutive addresses with irregular strides. SLP vectorizers [Barik et al. 2010; Larsen and Amarasinghe 2000; Liu et al. 2012] also perform poorly, as the degree of SIMD parallelism considered is required to match the SIMD datapath width.

In this article, we introduce a new compiler technique called PAVER (PArtial VEctorizeR) to exploit partial SIMD parallelism effectively. One important special case is how to orchestrate 3-way operations of a given type (say, `double`) on a 4-way SIMD unit (say, with a 256-bit datapath). The novelty behind PAVER lies in several new techniques introduced to ensure a correct and efficient execution of partial vector operations on a SIMD unit. To handle partial vector loads and stores, we use vector instructions as wide as possible to maximize SIMD utilization while minimizing the overheads caused by memory access, packing/unpacking, and/or masking operations. In addition, we avoid introducing new memory (protection and corruption) errors. To handle partial vector computations, we avoid introducing new user-visible numeric exceptions in unused SIMD lanes. Thus, PAVER is a general compiler technique, as full vectorization is a special case of partial vectorization.

Specifically, this article makes the following contributions:

—We identify and investigate systematically for the first time how to vectorize partially vectorizable loops in multimedia and scientific applications. Such loops are hard to vectorize well unless their partial SIMD parallelism is exploited adequately.
—We introduce a new compiler technique, PAVER, for harnessing partial SIMD parallelism and formulate it as a generalization of the traditional SLP algorithm.
—We have implemented PAVER in LLVM and evaluated it on an Intel Haswell CPU (with a 256-bit AVX SIMD extension) and an ARM Cortex-A15 CPU (with a 128-bit NEON SIMD extension) using a set of 9 C/C++/Fortran applications that exhibit partial SIMD parallelism. PAVER achieves significantly better kernel and whole-program speedups than LLVM on both Intel's AVX and ARM's NEON.

The rest of this article is organized as follows. Section 2 examines some real-world programs with partial SIMD parallelism. Section 3 introduces PAVER. Section 4 describes its implementation in LLVM. Section 5 focuses on our experimental evaluation. Section 6 discusses the related work. Finally, Section 7 concludes the article.

2. MOTIVATION

In Section 2.1, we introduce a class of programs with partial SIMD parallelism. In Section 2.2, we explain why their hot loops cannot be vectorized well currently. In Section 2.3, we examine how production-quality compilers—ICC, GCC, and LLVM—vectorize a specific loop kernel. We then motivate the basic idea behind PAVER by describing how it exploits partial SIMD parallelism to vectorize this kernel effectively.

Table I. Nine Programs Containing 16 Time-Consuming Loops with Partial SIMD Parallelism

Program (Application)	Kernel (Containing Function)	Kernel Execution Time (%)	Description	Source
183.equake (Seismic Wave Propagation Simulation)	SMVP (smvp) SimLP1 (main) SimLP2 (main) SimLP3 (main)	50.0 17.1 6.6 5.5	Solves numerically the partial differential equations of elastic wave propagation, i.e., the Navier equations of elastodynamics, with these four kernels forming one single simulation step	SPEC CPU2000
435.gromacs (Chemistry/ Molecular Dynamics)	INL (inl1130)	58.8	Performs molecular dynamics, i.e., simulation of the Newtonian equations of motion for systems with hundreds to millions of particles	SPEC CPU2006
454.calculix (Structural Mechanics)	EC3D (e_c3d)	69.0	Performs finite element analysis on linear and nonlinear 3D structural applications	
Fluidanimate (Computer Graphics)	CFMT (Compute ForcesMT)	37.5	Simulates physics for animation purposes with the Smoothed Particle Hydrodynamics (SPH) method	PARSEC
miniMD (Molecular Dynamics)	CHN (compute _halfneigh)	77.0	Simulates the Newtonian equations of particle motion (a simplified version of the LAMMPS Molecular Dynamics Simulator)	Mantevo [Mantevo 2015]
C-Ray (Computer Graph-ics/Physics)	SHADE (shade)	46.5	Calculates direct illumination with the Phong reflectance model in a simple ray tracer	Raytracing [Tsiombikas 2015]
Brut4 (Physics)	G6ACC (G6ACC)	96.0	Approximates the motion of interactive particles using the brute force method	N-Body Simulation [Aarseth 2015]
Nbody5 (Physics)	NB5REG (regint) NB5LP1 (nbint) NB5LP2 (nbint) NB5LP3 (nbint)	33.7 15.0 13.3 7.0	Performs N-body simulations of star cluster	
Nbody6 (Physics)	NB6REG (regint) NB6NBI (nbint)	61.6 8.2	Performs N-body simulations of star clusters (the latest version)	

Table II. Kernel Characteristics

Kernel	IAA	DAA	ICF	NMA	Kernel	IAA	DAA	ICF	NMA
SMVP	✓	✓		✓	SHADE			✓	✓
SimLP1		✓		✓	G6ACC		✓	✓	✓
SimLP2		✓		✓	NB5REG			✓	✓
SimLP3		✓		✓	NB5LP1				✓
INL	✓			✓	NB5LP2	✓			✓
EC3D				✓	NB5LP3	✓			✓
CFMT			✓	✓	NB6REG			✓	✓
CHN	✓		✓	✓	NB6INT	✓			✓

2.1. Programs with Partial SIMD Parallelism

Table I lists a set of nine programs covering a variety of applications (as indicated), with their 16 most time-consuming loop kernels identified. Table II classifies these kernels in terms of whether they exhibit the following characteristics or not:

```
1: double/float x[n][3], fdot[n][3], f[n][3], x0dot[n][3], x0[n][3];
2: for (j=1; j<nnb; j++) {
3:   k = list[i][j];        // k and list[i][j] are of type int
4:   s = time − t0[k];   // s is of type double/float
5:   x[k][0] = (((fdot[k][0]*s) + f[k][0])*s + x0dot[k][0])*s + x0[k][0];
6:   x[k][1] = (((fdot[k][1]*s) + f[k][1])*s + x0dot[k][1])*s + x0[k][1];
7:   x[k][2] = (((fdot[k][2]*s) + f[k][2])*s + x0dot[k][2])*s + x0[k][2];
8: }
```

(a) Sequential code

Vectorizers		ICC (14.0.4)	GCC (4.9.1)	LLVM (3.6)	PAVER	
					Safe	Aggressive
Speedups (x)	**float**	0.66	–	–	1.71	1.95
	double	0.46	–	1.32	1.47	1.75

(b) Speedups over LLVM's scalar code on Intel Haswell with 256-bit AVX SIMD extension

Fig. 1. Performance benefits of exploiting partial SIMD parallelism on NB5LP3 from Nbody5 in Table I.

—IAA: Indirect array accesses on some arrays.
—DAA: Dynamically allocated arrays for some arrays.
—ICF: Input-dependent control flow inside innermost loops.
—NMA: Nonconsecutive memory accesses for some array references across consecutive iterations of innermost loops.

In these kernels, most of their computations happen inside their innermost loops. EC3D has the largest loop depth, which is 7. Each innermost loop contains typically one or several groups of N independent and isomorphic operations, where $N = 3$ or 9 for INL and $N = 3$ for the remaining kernels. Each operation group is often responsible for computing the displacement, velocity, acceleration, or force of an object in a 3D space.

2.2. Research Compilers

Existing loop-level vectorizers [Bik et al. 2002; Eichenberger et al. 2004; Kong et al. 2013; Nuzman et al. 2006; Sreraman and Govindarajan 2000; Sujon et al. 2013] fail to exploit loop-level parallelism effectively for these 16 kernels except NB5LP2 (Section 5). Due to the presence of IAA, DAA, ICF, or NMA in a kernel, a scalar operation often operates on nonconsecutive addresses with irregular strides across consecutive iterations of its innermost loop.

Existing SLP vectorizers [Barik et al. 2010; Larsen and Amarasinghe 2000; Liu et al. 2012] are also inadequate. Present-day compilers often cannot exploit cross-iteration SIMD parallelism profitably via loop unrolling (as discussed in Section 5). According to Table II, two 3-way operation groups found across consecutive iterations of the innermost loop in a kernel will (1) be guided by input-dependent conditionals (due to ICF), or (2) operate on non-consecutive memory addresses (due to NMA), or (3) exhibit some data dependences (due to IAA). Given a 3-way operation to be vectorized for a 4-way SIMD unit, existing SLP vectorizers generate scalar code to perform the three operations since the degree of SIMD parallelism does not match the SIMD datapath. In this case, all SIMD resources are left idle.

2.3. Production-Quality Compilers

We first examine how ICC, GCC, and LLVM have applied the state-of-the-art loop-level and SLP techniques to vectorize NB5LP3, a hot loop in Nbody5 (Figure 1), on a Haswell CPU (equipped with Intel's AVX). We then motivate the development of PAVER. This

loop, shown in Figure 1(a), performs coordinate prediction for each particle inside the simulation space during each integration step. In lines 5 through 7, all arrays accessed are of type `double` if double precision is used and `float` if single precision is used.

In Figure 1(a), the three statements in lines 5 through 7 are independent and isomorphic, with their matching loads and stores touching consecutive memory addresses. However, the main obstacle to exploiting cross-iteration SIMD parallelism is the presence of IAA to all of the arrays, especially `x`, via `k = list[i][j]`. Let `x[k1]` and `x[k2]` be two rows of `x` written into by two adjacent iterations of loop `j`, where `k1 = list[i][j]` and `k2 = list[i][j+1]`. During program execution, `x[k1]` and `x[k2]` may be consecutive, nonconsecutive, or even identical. A similar analysis applies to the other arrays that are read in lines 5 through 7. If `x[k1]` and `x[k2]` are identical, then output dependences between the two accesses exist. Thus, exploiting cross-iteration SIMD parallelism by executing more than three operations in parallel is usually not done due to the potentially high packing/unpacking overhead.

Figure 1(b) compares the speedups achieved by ICC, GCC, and LLVM and those achieved by PAVER in both single- and double precision on a Haswell CPU under the compiler flags given later in Table IV. The baseline is LLVM's scalar code.

ICC vectorizes this kernel by exploiting loop-level parallelism for every statement in lines 5 through 7 across consecutive loop iterations. As a result, expensive gather instructions, `vgatherdps` for `float` and `vgatherdpd` for `double`, are used for loads to all of the input arrays due to IAA. In addition, the results computed for consecutive instances of the same statement must be extracted from their containing vector register and committed in sequential order due to the existence of potential cross-iteration output dependences on `x`. Therefore, ICC's vectorization results in somewhat significant performance slowdowns. GCC does not vectorize this kernel. Neither does LLVM's loop-level vectorizer. When asked to utilize the full 256-bit data path, LLVM's SLP vectorizer cannot vectorize this kernel either. When asked to use only 128-bit SSE instructions, this SLP vectorizer has managed to parallelize two statements in lines 5 and 6 by exploiting SLP, only in double precision, achieving some reasonably good performance speedups. Note that Intel's AVX/SSE does not have 64-bit SIMD instructions to perform 2-way single-precision floating-point operations.

PAVER outperforms the state-of-the-art vectorization techniques quite significantly by improving SIMD utilization and reducing their packing/unpacking and memory access overheads incurred. This is achieved by executing lines 5 through 7 in parallel on a 4-way SIMD unit by orchestrating a correct and efficient execution of partial vector operations for loads, stores, and computations. PAVER can operate in two execution modes. In safe mode, we avoid introducing new numeric exceptions in the unused lane. In aggressive mode, this is unnecessary because all exceptions are masked.

3. PARTIAL VECTORIZATION: METHODOLOGY

SLP vectorization has two phases: statement grouping and code generation. PAVER generalizes it by exploiting partial SIMD parallelism more effectively. For full vector operations, we proceed the same as before [Barik et al. 2010; Larsen and Amarasinghe 2000; Liu et al. 2012]. For partial ones, we introduce a number of techniques to ensure their correct and efficient execution on SIMD units. Let VL be the size of vectors for a given data type. Without loss of generality, we discuss how to vectorize 3-way double-precision (64-bit) operations for Intel's AVX, a 256-bit SIMD unit, where VL = 4. Our example basic block is given in Figure 2(a), which is simplified and modified from Figure 1(a). We have selected it carefully to help the reader understand our new methodology, especially when facilitated by Figures 2 through 6.

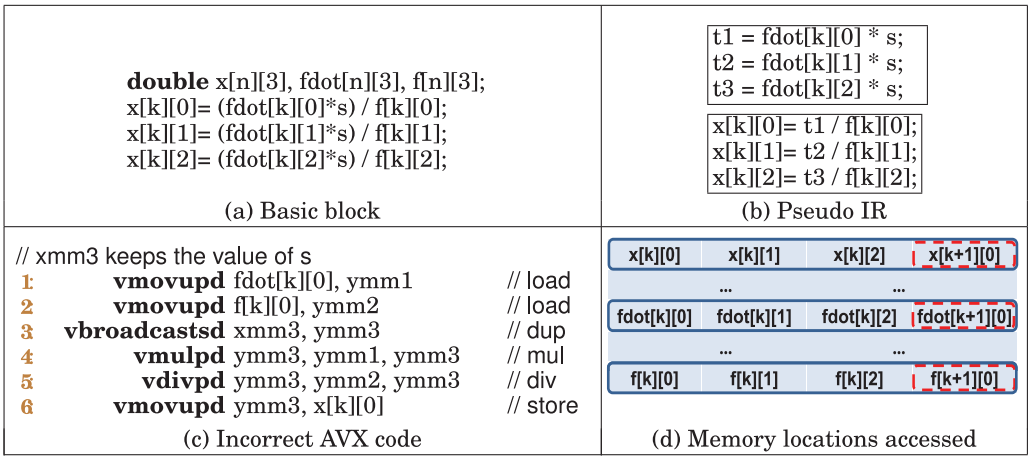


Fig. 2. An "ideal" but incorrect exploitation of 3-way partial SIMD parallelism on a 4-way SIMD unit. There are potential memory protection and corruption errors at the locations marked by dashed boxes in (d) and numeric exceptions in the unused SIMD lane.

3.1. Statement Grouping

Given the basic block in Figure 2(a), its pseudo intermediate representation (IR) is shown in Figure 2(b). Unlike existing vectorizers, PAVER also identifies and vectorizes partial vector operations if better performance can be achieved. In Figure 2(b), two groups (or boxes) of three independent and isomorphic operations are found. In either case, the superword size is 3, which is smaller than VL = 4.

3.2. Code Generation

Given the 3-way double-precision operations in Figure 2(b), Figure 2(c) gives an "ideal" translation in terms of full vector operations that utilize the entire datapath of Intel's AVX (as a 4-way SIMD unit). Instruction #1 is a load of 256-bit data starting from `fdot[k][0]` into the 256-bit register `ymm1`. Instruction #3 broadcasts the value in the low 64-bit location of the 128-bit register `xmm3` (i.e., the value of `s`) into the other three 64-bit locations of the 256-bit register `ymm3`, where `xmm3` is aliased with the lower half of `ymm3`. Instruction #4 (#5) is a 4-way vector multiplication (division). Instruction #6 is a store of 256-bit data in `ymm3` at `x[k][0]`. However, the vectorized code is incorrect since instructions #1, #2, and #6 access the locations, highlighted by the dashed boxes in Figure 2(d), that are not originally intended, causing potentially wrong output, memory errors, or even program crashes. In addition, instructions #4 and #5 may trigger new user-visible numeric exceptions on the unused lane.

Next, we describe how to perform partial vector loads, partial vector computations, and partial vector stores correctly and efficiently. By introducing PAVER, assisted by Figures 3 through 6, all AVX instructions used are expected to be self-explanatory.

3.2.1. Partial Vector Loads. How do we load efficiently the three consecutive 64-bit elements of array `f`, i.e., `f[k][0]`, `f[k][1]`, and `f[k][2]`, shown in Figure 2(b) into a 256-bit vector register from memory?

(a) Simple techniques. There are two straightforward approaches: data insertion and masking. As shown in Figure 3, data insertion is accomplished by scalar and vector loads followed by packing.

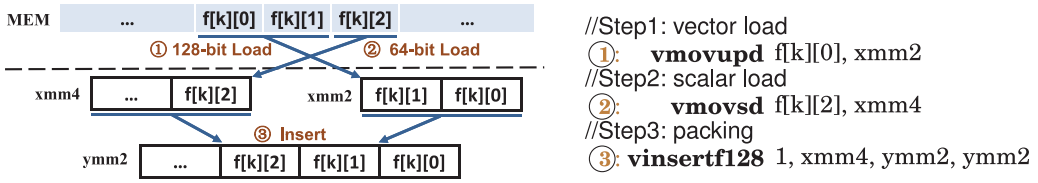


Fig. 3. Data insertion for loading `f` in Figure 2(b).

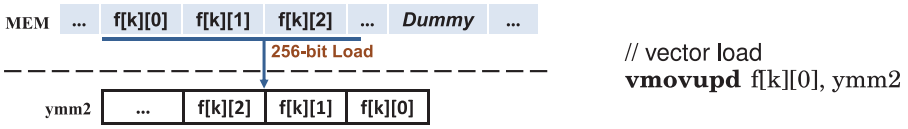


Fig. 4. Widened vector loads for `f` in Figure 2(b), facilitated by adding dummy elements at the end of `f` to avoid memory protection errors, where the number of dummy elements is less than VL.

Instead of insertion, masked vector loads are supported by some SIMD units, such as Intel's AVX2 (but not ARM's NEON). One masked load, "`vmaskmovpd f[k][0], mask, ymm2`", is sufficient, with `mask` set to select `f[k][0]`, `f[k][1]`, and `f[k][2]`.

(b) Widened vector loads. Data insertion can suffer from memory access overhead (due to many loads being issued) and packing overhead. Masked loads may suffer from mask bookkeeping overhead and some performance pitfalls [Intel 2014].

We introduce a simple yet effective approach to performing partial vector loads. The basic idea is to widen the vectors used to avoid the packing and masking overheads incurred traditionally. Of course, we must avoid introducing new memory protection errors when "walking off the end" of the array being operated on.

As shown in Figure 4, one single widened vector load suffices if the elements loaded are consecutive in memory, forming a superword. However, a wider vector load may touch locations that are not accessed originally at the end of an array, causing potentially memory protection errors. This can be avoided by adding N dummy elements at its end, where $N < \text{VL}$, via tail padding. For a static array, this can be done at compile time, safely for ANSI-compliant C programs. For a dynamic array, its original size can be increased by N in a call to, e.g., `malloc`. In practice, the extra space requested is smaller (or even 0), as a dynamic array is aligned to 8 (16)-byte boundary on 32-bit (64-bit) machines. Thus, the extra space consumed is negligible.

Such simple data transformation will not affect how locality-related compiler optimizations are applied, as they cannot effectively exploit the information regarding how different arrays are stored relatively. For example, loop tiling [Wolfe 1989; Xue 2000] reorders loop iterations without being sensitive to how each array is stored. Data tiling [Chatterjee et al. 1999; Huang et al. 2003] aims to optimize the layouts of individual arrays.

(c) Discussion. In addition to data insertion and masking, widened vector loads represent a new approach to performing partial vector loads. In PAVER, whichever approach is selected for a given SIMD unit is determined by a cost model. In the case of Intel's AVX and ARM's NEON, our new approach is generally the fastest (and thus selected) when the elements loaded are consecutive in memory.

3.2.2. Partial Vector Computations. We introduce a simple yet elegant solution to execute partial vector computations correctly and efficiently for SIMD units, such as SSE/AVX/AVX2 for Intel CPUs and NEON for ARM CPUs, where masked arithmetic instructions are not available. To execute partial vector computations, some SIMD

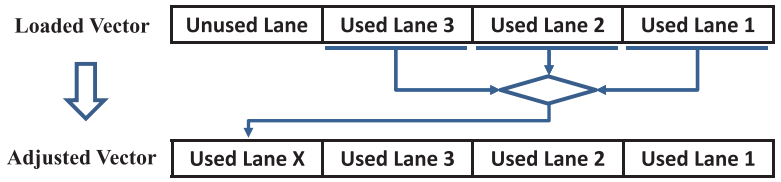


Fig. 5. Safe execution of partial vector computations.

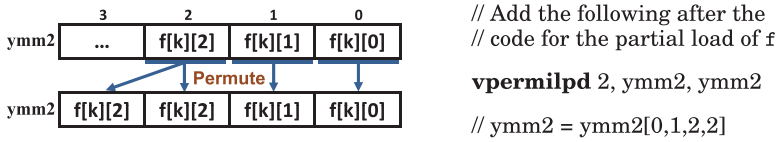


Fig. 6. Safe execution of partial vector computations for Figure 2(b) on Intel's AVX.

lanes do not perform any computation in the original program. To avoid introducing any new user-visible numeric exceptions, appropriate data values need to be assigned to these lanes. For example, the appropriate value for a divisor can be 1 (but not 0).

However, adjusting vector operands for each partial vector computation this way is costly, and different partial vector computations may require different value adjustments. For efficiency and simplicity, we propose to adopt two different execution modes—safe mode and aggressive mode—as explained next.

(a) Safe mode. As shown in Figure 5, we replicate the computational behavior of an arbitrarily selected used lane in every unused lane by simply replicating the values loaded initially into the used lane. Such initial values can be available in a vector initialized from memory or data packing. As a result, no new numeric exceptions will arise. This is a general approach that works on any data type. Its efficiency depends on the SIMD instructions provided by the underlying hardware.

Figure 6 illustrates how the safe mode is enforced for Intel's AVX. In Section 5.4, we discuss how to do this for ARM's NEON.

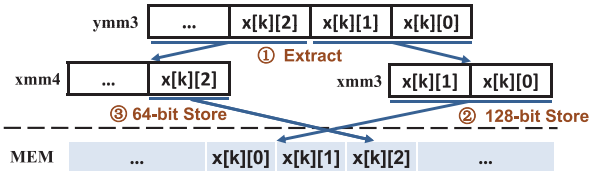
(b) Aggressive mode. The preceding replication is unnecessary if all numeric exceptions are masked. For example, on Intel CPUs, all floating-point exceptions are masked on a power-up, and SSE/AVX integer instructions do not generate numeric exceptions. When both flush-to-zero (FTZ) and denormals-are-zero (DAZ) are also enabled, application performance improves, in general.

3.2.3. Partial Vector Stores. How do we store efficiently the three consecutive 64-bit elements of array `x`, i.e., `x[k][0]`, `x[k][1]`, and `x[k][2]`, as shown in Figure 2(b), from a 256-bit vector register into memory?

(a) Simple techniques. There are also two straightforward approaches: data extraction and masking. As shown in Figure 7, data extraction is an analogue of data insertion in Figure 3 (but "reversed"). Instead of extraction, the results in `x` can be stored into memory with one masked vector store, "`vmaskmovpd ymm3, mask, x[k][0]`," with `mask` set to select `x[k][0]`, `x[k][1]`, and `x[k][2]`.

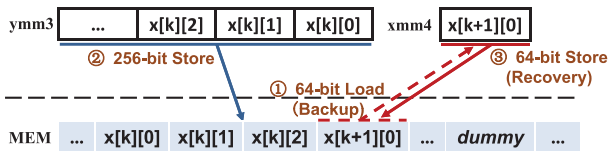
(b) Widened vector stores. Data extraction can suffer from memory access overhead (due to many stores being issued) and unpacking overhead. Masked stores may suffer from mask bookkeeping overhead and some performance pitfalls [Intel 2014].

We introduce a novel yet simple approach to performing partial vector stores. Just like the case of partial vector loads, we widen the vectors used to avoid the unpacking and masking overheads incurred traditionally. Meanwhile, we must also avoid




```
//Step1: unpacking
1: vextractf128  1, ymm3, xmm4
//Step2: vector store
2:     vmovupd   xmm3, x[k][0]
//Step3: scalar store
3:     vmovsd    xmm4, x[k][2]
```

Fig. 7. Data extraction for storing x in Figure 2(b).




```
//Step1: scalar backup load
1:     vmovsd    x[k+1][0], xmm4
//Step2: vector store
2:     vmovupd   ymm3, x[k][0]
//Step3: scalar recovery store
3:     vmovsd    xmm4, x[k+1][0]
```

Fig. 8. Widened vector stores for x in Figure 2(b), facilitated by appending dummy elements to x to avoid memory protection errors and by performing BR to avoid memory corruption errors.

introducing not only memory protection errors (at the end of an array) as before but also memory corruption errors (in the middle of an array).

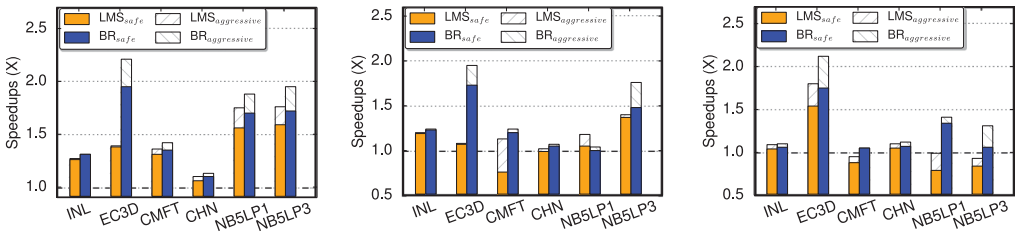
As shown in Figure 8, our approach for handling partial vector stores is an analogue of that for handling partial vector loads illustrated earlier in Figure 4. By appending again a few dummy elements to x via tail padding, `x[k+1][0]` is guaranteed to be "part" of array x. Thus, we have avoided memory protection errors as before (for loads).

Unlike the case for partial vector loads, however, we must also avoid memory corruption errors due to the writes that do not exist originally. In this example (shown in Figure 8), the full vector write in line 2 may corrupt the value in `x[k+1][0]`. We propose to apply a backup and recovery (BR) mechanism to correct such a corrupted value. As illustrated in Figure 8, the value in `x[k+1][0]` is first backed up in line 1 and then recovered in line 3, after it may have been corrupted previously in line 2.

BR may not be needed for 1D arrays of size $N <$ VL. For example, if N-way operations are performed, the corrupted locations are all dummy values due to tail padding. Thus, there is no need to back up and recover the corrupted dummy values.

For the scalar backup load in line 1, its latency can be usually hidden when executed earlier or in parallel with the full vector write in line 2. For the scalar recovery write in line 3, it may introduce a flow dependence with a subsequent load from the same location represented by `x[k+1][0]`. This occurs rarely for the kernels with partial SIMD parallelism due to IAA and NMA highlighted in Table II. By profiling, some flow dependences are found to be introduced by BR only in `INL` (11 out of 8,639 BR operations), `CHN` (3 out of 3,670,014 BR operations), and `CFMT` (1,260 out of 4,800 BR operations) among the 16 kernels given in Table II. Even though such flow dependences exist, there may be sufficiently many other operations that can be executed to hide the latency caused by the recovery write. By design, PAVER guarantees correctness regardless of whether such flow dependences exist or not (for single-threaded programs as discussed later). These statistics explain why BR is effective (in our evaluation), as its overhead is usually more than offset by its benefit.

An alternative solution to BR is load-modify-store (LMS), which proceeds also in three steps and illustrated later with a reference to Figure 8. First, `x[k+1][0]` is backed up in `xmm4` as before. Second, a packing instruction is executed so that `x[k+1][0]` contained in `xmm4` is inserted into the highest 64-bit of `ymm3`. Finally, a full vector store from `ymm3` into `x[k][0]` is performed. Unlike BR, LMS can aggravate contention on the hardware resource that performs its packing operation, as the same resource may also be used



(a) Single-Precision on Intel's AVX (b) Double-Precision on Intel's AVX (c) Single-Precision on ARM's NEON

Fig. 9. Kernel speedups of PAVER over LLVM's scalar, with BR and LMS used for partial vector stores in safe and aggressive modes.

by subsequent partial vector loads. By shifting some workload to a relatively less-busy resource for handling the store operations (owing to the widened store used), BR avoids this drawback. Figure 9 compares the performance speedups of PAVER when BR and LMS are used for the six kernels that require BR, as indicated later in Table V, on the two platforms described later in Table III with the compiler flags in Table IV. For ARM's NEON, only single-precision is considered, as partial SIMD parallelism is not exploitable on NEON when double-precision computations are performed in a 2-way fashion. BR is inferior to LMS only for `NB5LP1` in double precision on Intel's AVX due to cache-line split accesses, which is analyzed in Section 5.3.3(a), but is always superior over LMS on ARM's NEON. For some kernels, such as CMFT, LMS's performance is not even as competitive as the scalar case. Thus, LMS is not discussed any further in the rest of this article.

In this work, we focus on single-threaded programs. For multithreaded programs, BR (and the alternative solution LMS discussed earlier) may cause data races on the memory location represented by `x[k+1][0]`. Therefore, some improvement is needed to guarantee thread-safe execution if BR is enabled by a compiler flag in this case.

(c) Discussion. In addition to data extraction and masking, widened vector stores represent a new way to perform partial vector stores. In PAVER, the best technique is selected by a cost-benefit analysis for a given SIMD unit. For the set of kernels listed in Table II running on Intel's AVX and ARM's NEON, widened vectors are preferred.

4. PARTIAL VECTORIZATION: AUTOMATION

We present PAVER, introduced earlier, as a generalization of the traditional SLP algorithm [Larsen and Amarasinghe 2000] by considering full vectorization as a special case of partial vectorization. However, our approach can be used to empower a loop-level vectorization algorithm to also exploit partial SIMD parallelism. In Section 4.1, we present the core of PAVER, which works by searching for independent and isomorphic statements from stores. The code for starting from loads is similar and thus omitted. In Section 4.2, we discuss its implementation in LLVM.

4.1. PAVER: A Generalized SLP Algorithm

Figure 10 gives `Paver()` on some pseudo IR. There are two phases: statement grouping (line 1) and code generation (line 2). For full vector operations, PAVER emits code in the same way as before [Larsen and Amarasinghe 2000]. For partial arithmetic operations, their corresponding full vector operations are used. To handle a partial vector load (store), PAVER considers several approaches—data insertion (extraction), masking, and widened vectors—as described in Section 3.2.1 (Section 3.2.3). The instruction sequences generated by different approaches are costed (in terms of estimated

```
Procedure PAVER(B : BasicBlock)
begin
[1]  P = STMTGROUPING(B)
[2]  CODEGEN(P)
Procedure STMTGROUPING(B : BasicBlock)
begin
[3]  S ← set of stores in B
[4]  C ← {P | P ⊆ P(S), ∀ G ∈ P : (2 ⩽ |G| ⩽ VL ∧ all stores in G are data independent),
           every two groups in P are pairwise disjoint and have no dependence cycles}
[5]  foreach P ∈ C do
[6]      P.gain ← 0
[7]      foreach G ∈ P do
[8]          gain ← ESTIMATEGAIN(G)
[9]          if gain > gain_threshold then
[10]             P.gain ← P.gain + gain
[11]         else P ← P \ {G}
[12] return P such that P.gain is the largest
```

```
Procedure ESTIMATEGAIN(G : StmtGroup)
begin
[13] gain ← 0
[14] if ∃ s ∈ G : s is unvectorizable or nonisomorphic with the others in G then
[15]     return gain − PackingCost(G)
[16] fn ← function performed in, i.e., operator applied to G
[17] for i = 1 → fn's arity do
         // a load's arity is 0
[18]     G′ ← set of statements defining the i-th operands of all statements in G
[19]     gain ← gain + ESTIMATEGAIN(G′)
[20] return gain + ScalarCost(G) − VectorCost(G)
```

```
Procedure CODEGEN(P : SetOfStmtGroups)
begin
[21] for G ∈ P do
         GROUPGEN(G)
Procedure GROUPGEN(G : StmtGroup)
begin
[22] if ∃ s ∈ G : s is unvectorizable or nonisomorphic with the others in G then
[23]     Scalarize G and emit packing code
[24] fn ← function performed in, i.e., operator applied to G
[25] for i = 1 → fn's arity do
         // a load's arity is 0
[26]     G′ ← set of statements defining the i-th operands of all statements in G
[27]     op_i ← result of GROUPGEN(G′)
[28] Vectorize fn(op_1, ..., op_{fn's arity}), and if needed, emit unpacking code
```

Fig. 10. A generic SLP algorithm.

execution cycles) individually in line 20 of Phase 1, with the least-cost instruction sequence emitted in line 28 of Phase 2. Let us look at these two phases:

Phase 1: Statement grouping (line 1). Given a basic block B, STMTGROUPING() returns a set P of groups of independent stores. Each group G in P is identified as a

set of stores signifying the starting points of chains of partial vector operations (with full vector operations treated as a special case). In lines 3 and 4, all possible choices for P are found. VL once again denotes the vector length expressed as the number of elements of a data type. In lines 5 through 12, the most profitable one is selected by a cost-benefit analysis. In line 9, *gain_threshold* is typically set as 0. In line 20, $ScalarCost(G)$ and $VectorCost(G)$ estimate the costs, i.e., number of execution cycles of scalarizing and vectorizing G, respectively. Together with $PackingCost(G)$, these two parameters are architecture specific and implemented as calls to the API of a cost model provided in LLVM and extended as described in Section 4.2.

Phase 2: Code generation (line 2). How to vectorize a partial vector load or store has been determined in line 20 of Phase 1. Given a set P of groups of stores found by StmtGrouping() for a basic block, CodeGen() is called to emit vectorized code.

Let us apply Paver() to the basic block in Figure 2(a). Let s_1, s_2, and s_3 denote its three stores. First, StmtGrouping() is called. Then $\{s_1, s_2, s_3\}$ is assigned to S in line 3 and $\{\{\{s_1, s_2\}\}, \{\{s_1, s_3\}\}, \{\{s_2, s_3\}\}, \{\{s_1, s_2, s_3\}\}\}$ is assigned to C in line 4. Thus, $P = \{G\} = \{\{s_1, s_2, s_3\}\}$ is the best set of statement groups returned. By calling CodeGen(), Paver generates vectorized code for this basic block, as discussed in Section 3, according to the cost model used.

4.2. Paver: Implementation

We have implemented Paver on top of LLVM's SLP vectorizer by covering statement grouping, cost modeling, and code generation. Unlike LLVM's SLP vectorizer, Paver does not restrict the size of a statement group to be VL to harness partial SIMD parallelism. The time complexity of Paver is dominated by that of StmtGrouping(), which is determined largely by $|C|$ (line 4 in Figure 10). To reduce the size of C (and consequently the overall search space), we group stores that are close together and access consecutive memory locations. In practice, since an operation group that contains consecutive stores is small (less than 10 for the 16 kernels in Table II), Paver is efficient for real code as shown in Section 5.

We have modified LLVM's cost model (built in a similar spirit of Barik et al. [2010]) and code generation module to cost and vectorize partial vector loads and stores. In the case of BR illustrated in Figure 8, the backup cost is assumed to be hidden and its recovery cost is accounted for (as discussed in Section 3.2.3(b)). As for masked vector loads and stores, we have applied a patch from Intel [Demikhovsky 2015] to estimate their costs. Statically unmodelable quantities such as cache behavior (e.g., cache misses and cache-line splits) are ignored.

To avoid memory protection errors for an array in Paver, we apply tail padding at its end, as described in Section 3. Such arrays, which may be accessed via pointers, are found by performing Andersen's pointer analysis [Ye et al. 2014].

Finally, Paver, like every other vectorizer, uses an aligned load/store instruction `movapd` instead of an unaligned one `movupd` shown in Figures 4 and 8, wherever it is possible to do so. This does not happen often for the class of kernels considered due to IAA, DAA, ICF, and NMA highlighted in Table II unless the underlying array is a 1D array (of size $\leqslant$ VL) aligned on VL boundaries.

5. EVALUATION

As this article is the first to exploit partial SIMD parallelism, we proceed to show that Paver is effective, where several production-quality compilers equipped with the state-of-the-art vectorization techniques fail, in exploiting partial SIMD parallelism inherent in the real-world programs exemplified in Table I.

Table III. Computing Platforms

Platform	Intel	ARM
CPU	i7-4770 Haswell (3.4GHz, 4 core)	Cortex-A15 (1.2GHz, 1 core)
SIMD width	256-bit	128-bit
L1D cache	32KB (8 way, 64B/line)	32KB (2 way, 64B/line)
L2 cache	256KB (8 way, 64B/line)	1MB (16 way, 64B/line)
L3 cache	8MB (Shared)	—
OS	Ubuntu 13.10 64-bit	Linaro 14.03 32-bit

Table IV. Compiler Configurations

Compiler	Flags	
ICC (14.0.4)	Vectorization:	-vec
	Others:	-O2 -march=core-avx2 -ftz -ansi-alias -restrict -no-fma -prec-div
GCC (4.9.1)	Vectorization:	-ftree-vectorize
	Others:	-O2 -march=native -ffast-math -unroll-loops -mno-fma -mno-recip
LLVM (3.6)	Vectorization:	-vectorize-loops -vectorize-slp (SLP)
	Others:	-O2 -loop-unroll -march=native -ffast-math
PAVER (LLVM 3.6)	Vectorization:	-vectorize-loops -vectorize-slp (SLP)
	Others:	-O2 -loop-unroll -march=native -ffast-math

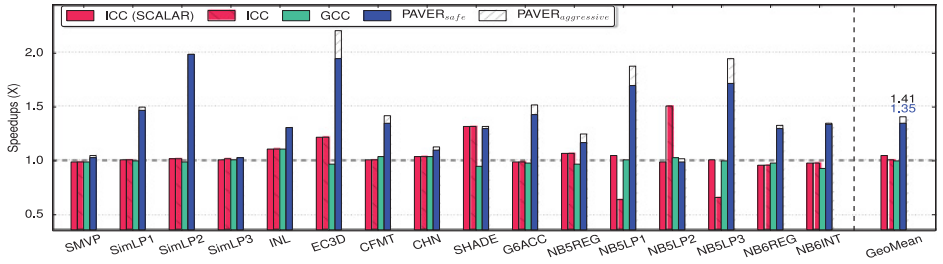
5.1. Hardware and Software Platforms

Our computing platforms are listed in Table III. We have selected an Intel Haswell CPU because it has a sophisticated SIMD unit, supporting data insertion and extraction as well as masked loads and stores (but not masked arithmetic instructions), so that different compiler techniques can be compared and contrasted. Given the 256-bit SIMD datapath available, its vector length VL is 4 for double precision and 8 for single precision. To demonstrate the generality of our approach, an ARM Cortex-A15 CPU is also selected. However, its NEON SIMD unit does not support any masked operation. Given the 128-bit SIMD datapath width available, its VL is 2 for double precision and 4 for single precision.

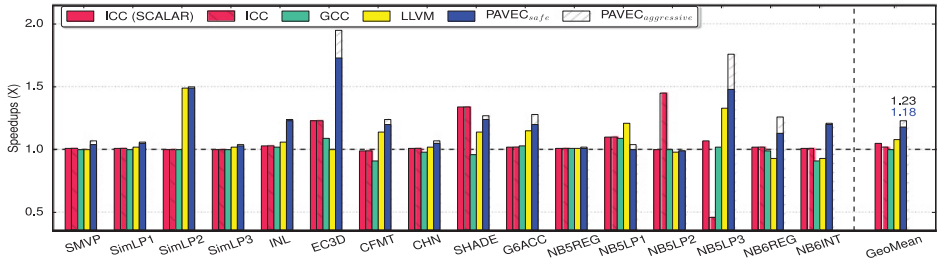
We compare PAVER with some latest releases of ICC, GCC, and LLVM under the compiler flags used as in Table IV. These compilers support both loop-level and basic block vectorization. LLVM has one loop-level vectorizer and two basic block vectorizers, i.e., SLP and BBV. As the SLP and BBV vectorizers exhibit comparable performance across the benchmarks evaluated in our experiments with the former outperforming the latter slightly on average, only the results of SLP are discussed. Later, LLVM denotes a configuration in which both loop-level and SLP vectorizers are enabled.

We have selected the flags given in Table IV to force different compilers to generate code with a similar precision (by disabling FMA and reciprocal estimates for division) and a similar optimization strength (by turning on "-O2" and loop unrolling). In ICC (for the Haswell CPU), unrolling is implied by "-O2." In addition, FTZ and DAZ are enabled, with `-ftz` for ICC and `-ffast-math` for GCC and LLVM. In all configurations, the `restrict` type qualifier for C programs is recognized to facilitate alias analysis.

We have also compared the performance results of every vectorizer under "-O2" and "-O3" on both computing platforms. No vectorizer has exhibited noticeable performance differences for the set of loop kernels considered under the two optimization settings.



(a) Single precision



(b) Double precision

Fig. 11. Kernel speedups of PAVER, ICC, GCC, and LLVM over LLVM's scalar code on Intel's Haswell.

5.2. Benchmarks

For the nine benchmarks listed in Table I, both kernel and whole-program speedups are presented. Our kernels are the 16 hot loops given in Table I. Their loop characteristics are summarized in Tables I and II and discussed in Section 2.1. To execute these loops, we have used the reference inputs for the six SPEC kernels, the "large" input for `CFMT` from PARSEC and the larger of the two inputs for `SHADE`. For the others, the inputs from their associated programs are used. For `SMVP`, `SimLP1`, `SimLP2`, `SimLP3`, and `G6ACC`, the current compiler technology cannot disambiguate their nonaliased arrays to enable their automatic vectorization. This problem is overcome by adding the `restrict` qualifier as required. For each kernel, the same source code (for both single precision and double precision) is used by all compilers.

The whole programs are executed using the same inputs as discussed earlier.

The execution time of a kernel or program is measured as the average of 20 runs.

5.3. Results and Analysis on an Intel Haswell CPU

Figure 11 compares PAVER with ICC, GCC, and LLVM on the 16 hot kernels in single- and double precision over LLVM's scalar code. PAVER always opts to use widened vectors to perform partial vector loads and stores according to its cost model.

For PAVER, the results in safe and aggressive modes are given. In safe mode, the speedups achieved by PAVER over ICC, GCC, and LLVM are 1.12x, 1.18x, and 1.09x in double precision and 1.29x, 1.35x, and 1.35x in single precision. In aggressive mode, these speedups have gone up to 1.17x, 1.23x, and 1.14x in double precision and 1.34x, 1.41x, and 1.41x in single precision due to removal of the associated permute instructions for data replication (Figure 6). In practice, this can be very beneficial for running fully debugged programs, with numeric exceptions masked. It is important to appreciate these speedups by noting the limited SIMD parallelism inherent in the kernels considered (Table II).

Table V. Vectorization Diagnosis (with the Diagnostic Messages from ICC and GCC)

	SMVP	SimLP1	SimLP2	SimLP3	INL	EC3D	CFMT	CHN	SHADE	G6ACC	NB5REG	NB5LP1	NB5LP2	NB5LP3	NB6REG	NB6INT
ICC	$\times^{*}$	$\times^{*}$	$\times^{*}$	$\times^{*}$	$\times^{*}$	$\times^{\dagger}$	$\times^{*}$	$\times^{*}$	$\times^{+}$	$\times^{+}$	$\times^{*}$	✓	✓	✓	$\times^{*}$	$\times^{*}$
GCC	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{+}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$	$\times^{\ddagger}$
LLVM	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	×	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$	$\checkmark^{8}$
PAVER	✓	✓	✓	✓	$\checkmark^{br}$	$\checkmark^{br}$	$\checkmark^{br}$	$\checkmark^{br}$	✓	✓	✓	$\checkmark^{br}$	✓	$\checkmark^{br}$	✓	✓

*: Unvectorized due to "the existence of vectordependence" and/or "insufficient SIMD parallelism."
†: Vectorization possible but seems inefficient.
+: Nonstandard loops not considered as vectorization candidates.
‡: Failed to find loop-level parallelism due to "bad data dependences" and SLP due to "not enough data-refs" (i.e., insufficient SIMD parallelism in basic blocks).
8: Vectorized only for double precision (Section 5.3.2).
br: Backup and recovery needed, as illustrated in Figure 8. Note that BR could be optimized away for 1D arrays with size no larger than VL (Section 3.2.3(b)).

Table V describes whether GCC, ICC, LLVM, and PAVER has succeeded in vectorizing a kernel and the diagnostic message for failing to do so. The presence of "insufficient SIMD parallelism" for many kernels highlights the performance benefits of its effective exploitation in this article. ICC, GCC, and LLVM have applied loop unrolling to `EC3D` and `G6ACC` only for exposing ILP only. In the case of PAVER, BR is not needed for a 1D array whose size is no larger than VL, as the backed-up locations are all dummy values due to tail padding (Section 3.2.3(b)). GCC fails to vectorize any kernels evaluated and achieves performance similar to that of LLVM on scalar code on average. Next, we examine ICC, LLVM, and PAVER individually in that order.

5.3.1. ICC. The ICC compiler has vectorized only three kernels—`NB5LP1`, `NB5LP2`, and `NB5LP3`—at the loop level, in the same way as explained for `NB5LP3` in Figure 1(a), with drastically different speedups. For the reason given there, significant slowdowns for `NB5LP3` are observed: 0.46x in double precision and 0.66x in single precision. `NB5LP1`, which has a similar computational pattern as `NB5LP3`, does not suffer quite the same way in double precision, as its vector loads are realized by data insertion (with individual scalar loads and packing) instead of using gather operations. Thus, different speedups are observed in single- and double precision. `NB5LP2` can be regarded as `NB5LP3` in Figure 1(a) but with loop-level parallelism inherent in a long-latency operation `a = body[k]/(rij2*sqrt(rij2))` inserted just before its line 4, where `a` is used in lines 5 through 7. By exploiting this extra amount of loop-level parallelism in this kernel, the only one that can be thus done profitably among the 16 kernels evaluated, ICC has outperformed all other vectorizers by somewhat big margins.

In the case of `SHADE`, the ICC compiler with -no-vec turned on, denoted "ICC (SCALAR)" in Figure 11, outperforms the others in terms of its scalar code by taking advantage of dot-product instructions. For this kernel, the ICC vectorizer emits the same code as ICC (SCALAR). PAVER, which underperforms ICC's scalar code slightly, is expected to outperform ICC if dot products can also be handled (orthogonally).

The ICC vectorizer is slightly faster than LLVM's scalar baseline on average. This is mainly due to better scalar code generated by ICC's scalar compiler, which is 1.05x faster in both double- and single precision than LLVM's scalar baseline on average.

5.3.2. LLVM. The LLVM vectorizer cannot vectorize any kernel given the full SIMD datapath, as the degree of SIMD parallelism exploited is required to match VL. By dropping it to VL/2 (so that only 128-bit SSE instructions are used), LLVM has vectorized all 16 kernels except EC3D in double precision but still none in single precision. Recall that there are no instructions to perform two single-precision floating-point operations on Intel's Haswell. Therefore, Figure 11(a) has no bars for LLVM in single precision. For its double-precision results given in Figure 11(b), the degree of SIMD parallelism exploited is always 2. The mean speedup achieved by LLVM (over its own scalar baseline) is 1.08x.

By exploiting partial SIMD parallelism more effectively, PAVER outperforms LLVM for all kernels except NB5LP1 due to cache-line splits analyzed later. For example, INL contains a set of nine independent and isomorphic statements, which read from several arrays with piece-wise consecutive addresses. LLVM vectorizes INL by performing three 2-way and three 1-way operations in double precision. In contrast, PAVER has opted to three 3-way operations, achieving higher speedups. CHN is vectorized similarly except the achieved speedup is less due to ICF (see Table II).

SimLP2 is an interesting case. PAVER exploits 3-way operations and achieves a similar speedup as LLVM that exploits 2-way operations only. This is because a 256-bit division costs twice as much as a 128-bit division on Intel's AVX.

5.3.3. PAVER. In general, PAVER performs better in single- than double precision on a Haswell CPU, as shown in Figure 11. There are several reasons for the poorer performance in double precision, including (1) poorer cache locality and more cache-line splits (due to twofold increases in working sets and data element sizes), (2) longer latencies for 256-bit vector loads (one extra cycle due to the bypass delay between the Integer and X87/AVX-FP_HIGH execution stacks), and (3) longer latencies for 256-bit vector computations than 128-bit vector operations (e.g., division). PAVER is expected to perform better in double precision if better hardware support is available.

Let us consider SimLP1, SimLP2, EC3D, NB5LP1, and NB5LP3. In single precision, PAVER is the best performer among the four vectorizers compared. In double precision, however, PAVER continues to excel for EC3D and NB5LP3 but is not as impressive for SimLP1 (due to poorer cache locality), SimLP2 (due to costly 256-bit divisions performed as explained in Section 5.3.2), and NB5LP1 (due to more cache-line splits incurred).

Next, we analyze the effects of cache-line splits (using NB5LP1) and cache behavior (using SIMLP1) on the performance of PAVER. We also demonstrate the performance advantages of PAVER over data insertion/extraction and masking. In addition, we examine the relevance of PAVER in the context of some recent and future SIMD extensions. Finally, we discuss the whole-program speedups obtained.

(a) Cache-line splits. For NB5LP1 in Nbody5, PAVER significantly outperforms the other three compilers compared in Figure 11 in single precision but is the worst performer in double precision. To understand this, we compared the numbers of loads/stores and loads/stores that degenerate into cache-line split accesses, obtained from VTune for ICC, LLVM, and PAVER, which have successfully vectorized this kernel.

By using wider vector loads and stores, PAVER only requires 52% of the memory accesses made by LLVM's scalar compiler and ICC in both single- and double precision and 71% of the memory accesses made by LLVM in double precision. Note that LLVM has no single-precision bars in Figure 11. However, PAVER suffers more cache-line splits in double precision than single precision (by 2x), causing it to perform significantly better than the other compilers only in single precision.

(b) Cache behavior. Like NB5LP3 in Figure 1, the four hot kernels—SVMP, SimLP1, SimLP2, and SimLP3—from 183.equake each contain three independent and

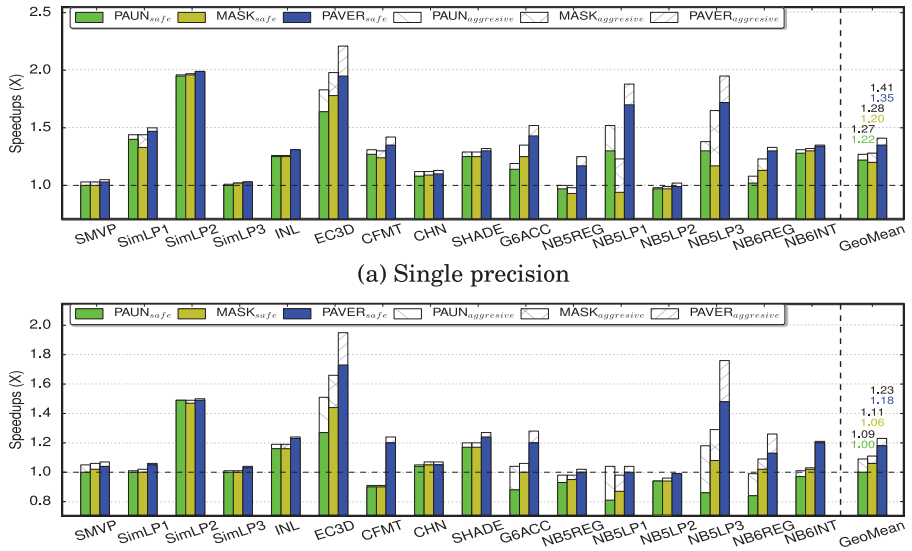


Fig. 12. Kernel speedups of PAVER, PAUN, and MASK (over LLVM's scalar code) on Intel's Haswell.

isomorphic chains of computations with varying arithmetic operators. Their working sets exceed the L2 cache size. `SimLP2` shows good speedups in single- and double precision (Figure 11), as its division operations can hide memory latency better.

For `SimLP1`, PAVER significantly outperforms the three compilers compared in Figure 11 in single precision but suffers from a big performance drop in double precision. To understand this, we analyzed the effects of the cache behavior of `SimLP1` on the effectiveness of the two vectorizers—LLVM (again in double precision only) and PAVER—that have vectorized this kernel. Under the reference input, the working set of `SimLP1` exceeds the L2 cache size by 22x and 36x in single- and double precision, respectively. Due to frequent cache misses and thus long-latency memory accesses, PAVER achieves consistently good speedups only in single precision. As its working set decreases (done by decreasing its problem size parameter `ARCHnodes`), `SimLP1` shifts from being memory- to compute bound. Accordingly, LLVM and PAVER have achieved improved speedups. However, PAVER outperforms LLVM by exploiting partial SIMD parallelism more effectively. When all accessed data fit into L2 in both single- and double precision, the speedups from PAVER start moving up, but those for LLVM have plateaued at 1.06x, as the pipeline resources have become a bottleneck for LLVM (due to more operations performed). In the end, PAVER$_{safe}$ reaches 2.42x and 1.60x in single- and double precision, respectively. As for PAVER$_{aggressive}$, these speedups are 3.00x and 1.63x, respectively.

(c) Comparing with data insertion/extraction and masking. We compare PAVER with PAUN and MASK. Here, PAUN (which stands for PAacking and UNpacking) applies data insertion to perform partial vector loads (Figure 3) and data extraction to perform partial vector stores (Figure 7). Instead of packing and unpacking, MASK uses masked vector loads and stores, as discussed in Sections 3.2.1(a) and 3.2.3(a), respectively. As mentioned earlier, PAVER has opted to use widened vectors to perform partial vector loads and stores for all kernels evaluated (by its cost model).

Figure 12 reveals the superiority of PAVER over PAUN and MASK in single- and double precision under safe and aggressive modes. PAVER achieves speedups that are better

than or similar to PAUN and MASK for the 16 kernels in all four configurations evaluated. Thus, the average speedups achieved by PAVER over the other two are better. In single precision, PAVER is faster than PAUN (MASK) by 10.7% (12.5%) in safe mode and 11.0% (10.2%) in aggressive mode. In double precision, PAVER is faster than PAUN (MASK) by 18.0% (11.3%) in safe mode and 12.8% (10.8%) in aggressive mode. PAVER achieves such consistent performance improvements, particularly in the case of `EC3D` and `NB5LP3`, because it has avoided the high packing/unpacking and memory access overheads incurred by PAUN and the masking overheads incurred by MASK. For `INL`, PAVER is slightly better than PAUN and MASK because many long-latency operations, e.g., `div` and `sqrt`, lie on its critical execution path, alleviating the impact of the packing/unpacking or masking overheads incurred.

On average, MASK is slightly better than PAUN, and PAUN wins only in single precision under safe mode. Looking at the 16 kernels individually, we find that their results are a mixed bag, with each beating the other visibly at some kernels. Overall, PAUN performs more poorly in double precision than in single precision.

PAUN suffers from packing/unpacking overhead and more memory access overhead (due to more loads and stores issued) than MASK and PAVER, exerting more pressure on the limited hardware resources, such as the reserve stations (RS) of the out-of-order pipeline. By examining the hardware event RESOURCE_STALLS.RS with VTune, we find that a kernel vectorized by PAUN tends to incur more stalls in double- than single precision due to the lack of eligible RS entries caused by poorer cache behavior. Therefore, PAUN achieves worse speedups in double- than single precision.

In contrast, MASK avoids PAUN's packing/unpacking overhead but ends up with some other performance pitfalls, such as increased uops, reduced load/store throughput, and ineffective store-load forwarding [Intel 2014; Fog 2014]. A masked vector load has a similar latency as a normal vector load. However, their reciprocal throughputs are different: 0.5 for a normal load but 2 for a masked load. In addition, a masked load requires 3x uops as a normal load. Masked vector stores are even worse. Finally, there exists no store-load forwarding for loads that depend on the masked stores with a non–all-one/zero mask. According to ILD_STALL.IQ_FULL and RESOURCE_STALLS.ROB from VTune, the instruction queue and reorder buffer are frequently the stall-inducing bottlenecks. However, bad cache behavior has less impact on MASK than PAUN, as the memory access delays can be partially hidden by the pipeline bubbles (caused when the instruction queue or reorder buffer is full). Therefore, MASK has managed to achieve better or similar speedups as PAUN in double precision for all kernels except for `NB5LP1` in aggressive mode. There are two reasons for the exception here. First, MASK suffers three times as many cache-line splits as PAUN (as MASK uses longer vectors than PAUN). Second, the port that executes the packing and unpacking instructions under PAUN is its performance bottleneck only in safe mode due to vector permutes (Figure 6). In aggressive mode, such permute instructions are no longer needed.

(d) Other SIMD extensions. The latest Intel SIMD extension AVX-512 provides not only masked loads and stores as 256-bit AVX does but also masked arithmetic instructions that are not supported by 256-bit AVX. On AVX-512, it is therefore possible to achieve partial vectorization by performing only the masked operations (for loads, stores, and computations) on the active SIMD lanes.

However, PAVER will still represent a promising solution for three reasons. First, some kernels do not benefit from masked operations due to masking overheads. If masked arithmetic instructions were available on Intel's AVX, then $\text{MASK}_{aggresive}$ analyzed earlier in Figure 12 would represent approximately the performance achievable (since $\text{MASK}_{aggresive}$ does not use permutation shown in Figure 6). However, PAVER is still more competitive than $\text{MASK}_{aggresive}$, on average. In single precision, PAVER_{safe} and

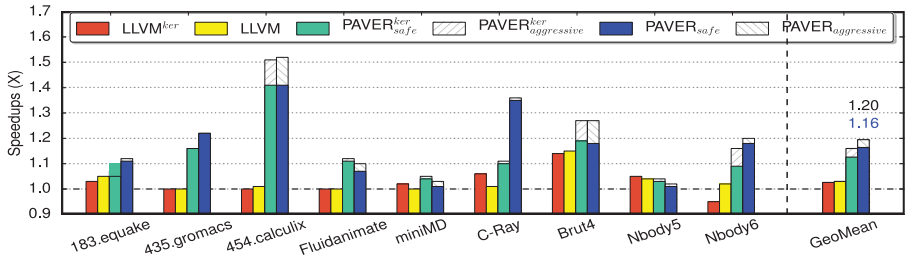


Fig. 13. Whole-program speedups of LLVM and PAVER (over LLVM's scalar code) on Intel's Haswell.

Table VI. Compile Time (secs) of Autovectorization (with PAVER$_{safe}$ and PAVER$_{aggressive}$ Indistinguishable)

Program	183.equake	435.gromacs	454.calculix	Fluidanimate	miniMD	C-Ray	Brut4	Nbody5	Nbody6
LLVM	0.19	3.38	11.46	0.08	0.62	0.05	3.42	2.24	4.78
PAVER	0.25	5.16	15.85	0.11	0.71	0.06	6.50	4.34	9.02

PAVER$_{aggressive}$ are faster than MASK$_{aggressive}$ by 5.5% and 10.2%, respectively. In double precision, PAVER$_{safe}$ and PAVER$_{aggressive}$ are faster than MASK$_{aggressive}$ by 6.3% and 10.8%, respectively. In safe mode, PAVER$_{safe}$ is inferior to MASK$_{aggresive}$ only in `EC3D` and `CHN` in single precision and `SMVP` and `CHN` in double precision. In aggressive mode, PAVER$_{aggressive}$ outperforms MASK$_{aggresive}$ across all 16 kernels. Second, for some kernels that benefit from masked operations on AVX-512, this masking scheme can be selected based on a cost-benefit analysis, as our cost model allows different vectorization techniques to be selected. Finally, some SIMD extensions, such as ARM's NEON, do not support masked operations. In Section 5.4, we demonstrate the effectiveness of PAVER in this case.

(e) Whole-program speedups. We compare PAVER, under its two configurations PAVER$_{safe}$ and PAVER$_{aggresive}$, with LLVM in terms of whole-program speedups achieved. As the 16 selected kernels are execution hot spots, representing an average of 64.7% of the total execution time in each program (Table I), it will also be instructive to see how the speedups achieved by these kernels shown in Figure 11 can translate into whole-program speedups. Suppose that a program P contains n selected hot kernels, with their percentage execution times, $t_1, \ldots, t_n$, as indicated in Table I and their speedups (over LLVM's scalar code) from Figure 11 given in absolute terms, $s_1, \ldots, s_n$. Let $V \in \{\text{LLVM}, \text{PAVER}_{safe}, \text{PAVER}_{aggresive}\}$. Let V^{ker} be its configuration under which all and only the selected hot kernels in a program are vectorized with their kernel speedups ideally translated into whole-program speedups. Thus, the absolute whole-program speedup for P under V^{ker} is given by $1/(1 - \sum_i^n t_i + \sum_i^n (t_i/s_i))$x.

Figure 13 shows the results, obtained under the compiler flags in Table IV, for the nine programs given in Table I. On average (with respect to LLVM's scalar code), PAVER achieves a whole-program speedup of 16% in safe mode (PAVER$_{safe}$) and of 20% in aggressive mode (PAVER$_{aggresive}$). In contrast, LLVM achieves only an average speedup of 3%. In most cases, the whole-program speedups achieved by a vectorizer are consistent with the cumulative speedups of their vectorized hot kernels achieved. Table VI gives their compile times. PAVER is more time consuming than LLVM, as PAVER spends more time in STMTGROUPING() and padding-related analysis.

This represents an apples-to-apples comparison, as PAVER differs from LLVM only in basic blocks where partial vectorization is performed. In particular, LLVM and PAVER vectorize the 16 kernels given in Table II in the same way as discussed earlier, where `INL` in `435.gromacs` and `CFMT` in `Fluidanimate` are single-precision kernels and the remaining 14 are double-precision kernels. For every other code (kernel), PAVER handles it either identically as LLVM (by either exploiting full SIMD parallelism

with LLVM's loop-level vectorizer or LLVM's SLP vectorizer or resorting to serial execution) or differently by exploiting partial SIMD parallelism in some of the basic blocks.

Let us now go through the results given in Figure 13. For `454.calculix` and `Brut4`, their whole-program speedups are achieved mainly by vectorizing their hot kernels selected in Table I (since V is comparable as V^{ker} in each case). For `183.equake`, `435.gromacs`, and `Nbody6`, vectorizing some other kernels (that are not selected in Table I) is also generally beneficial (since V outperforms V^{ker} in each case). Next, we analyze the remaining four—`Fluidanimate`, `minMD`, `C-Ray`, and `Nbody5`—in more detail.

Let us start with `Fluidanimate`, which is a single-precision program. LLVM fails to perform any vectorization on any kernels, including `CFMT` selected in Table I. Thus, LLVM behaves identically to LLVM^{ker}. As for PAVER, PAVER$_{safe}$ (PAVER$_{aggresive}$) is inferior to PAVER$^{ker}_{safe}$ (PAVER$^{ker}_{aggresive}$). In addition to `CFMT`, PAVER has also partially vectorized one of the two basic blocks residing in the innermost loop of a 4-level nested loop contained in the `AdvanceParticlesMT` function by exploiting its 3-way SIMD parallelism. However, this partial vectorization is ineffective, as the vectorized code generated is slower than LLVM's scalar code due to the packing overhead incurred in packing constants into the vector loads generated. The speedups of LLVM, PAVER$_{safe}$, and PAVER$_{aggresive}$ are 1x, 1.07x, and 1.1x, respectively.

In the case of `miniMD`, every vectorizer V is slightly worse than V^{Ker}, where $V \in$ {LLVM, PAVER$_{safe}$, PAVER$_{aggresive}$}. Its function, `compute_halfneigh_`, contains a 2-level nested loop, with its inner loop iterating over a set of three basic blocks (all guided by some if conditionals). Both LLVM and PAVER have vectorized two of these three blocks, but ineffectively, except LLVM exploits 2-way SIMD parallelism and PAVER exploits 3-way SIMD parallelism in both blocks. Overall, the speedups of LLVM, PAVER$_{safe}$, and PAVER$_{aggresive}$ are 1x, 1.01x, and 1.03x, respectively.

Let us now consider `C-Ray`, where LLVM is slightly inferior to LLVM^{ker} but PAVER$_{safe}$ and PAVER$_{aggresive}$ are significantly better than PAVER$^{ker}_{safe}$ and PAVER$^{ker}_{aggresive}$, respectively. As a result, PAVER outperforms LLVM significantly in this program. There is one hot kernel, `SHADE`, selected from the `shade` function (Table I), which operates in double precision in the source code given. For this kernel, PAVER generates faster code than LLVM, as shown in Figure 11(b). In addition, PAVER obtains its additional performance improvements by exploiting 3-way SIMD parallelism in six basic blocks residing in three functions—(1) two in `trace`, (2) three in `render`, and (3) one in `ray_sphere`—resulting in `trace` to be inlined in its two callers, `render` and `shade`, and `ray_sphere` to be inlined in its caller, `shade`. The four functions—`trace`, `render`, `ray_sphere`, and `shade`—contribute as a whole 60% to the total execution time of this program. Thus, PAVER has succeeded in enabling their performance to be boosted by 1.7x. Note that by vectorizing the hot kernel `SHADE` alone (Figure 11(b)), PAVER is not as impressive: 1.24x under PAVER$_{safe}$ and 1.27x under PAVER$_{aggresive}$. In contrast, LLVM exploits only 2-way SIMD parallelism in (1), (2), and (3). Due to the relatively larger sizes of vectorized `trace` and `ray_sphere`, these two functions are not inlined. For the same four functions combined, LLVM has boosted their performance by only 1.04x (despite its speedup of 1.14x achieved for `SHADE` in Figure 11(b)). Overall, the speedups of LLVM, PAVER$_{safe}$, and PAVER$_{aggresive}$ are 1.01x, 1.35x, and 1.36x, respectively.

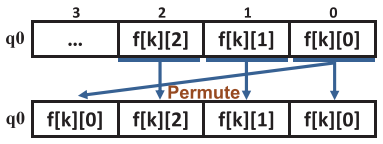
Finally, we consider `Nbody5`, where every vectorizer V is slightly worse than V^{Ker}. Both LLVM and PAVER have ineffectively vectorized a total of 13 basic blocks in three functions—(1) four in `fpcorr`, (2) four in `fpoly1`, and (2) five in `fpoly2`—resulting in vectorized code that is slightly slower than LLVM's scalar code, except LLVM exploits 2-way SIMD parallelism and PAVER exploits 3-way SIMD parallelism. The speedups of LLVM, PAVER$_{safe,}$ and PAVER$_{aggresive}$ for `Nbody5` are 1.04x, 1.01x, and 1.02x, respectively.

Data Insertion		Widened Vector Load	
vldr	f[k][0], d0	**vld1.32**	f[k][0], {d0, d1}
vld1.32	f[k][2], d1[0]		

Fig. 14. Partial vector loads for `f` in Figure 2(b) on ARM's NEON.

Data Extraction		Widened Vector Store	
vstr	d0, x[k][0]	**vldr**	x[k+1][0], s4
vst1.32	d1[0], x[k][2]	**vst1.32**	{d0, d1}, x[k][0]
		vstr	s4, x[k+1][0]

Fig. 15. Partial vector stores for `x` in Figure 2(b) on ARM's NEON.




```
// Add the following after the
// code for the partial load of f
vext.32 q1, q0, q0, #3
vext.32 q0, q1, q0, #1
// q0 = q0[0,1,2,0]
```

Fig. 16. Safe execution of partial vector computations in Figure 2(b) on ARM's NEON.

LLVM is the best performer, as PAVER is ineffective in vectorizing kernel `NB5LP1` due to the cache-line splits suffered, as discussed in Section 5.3.3(a).

5.4. Results and Analysis on an ARM Cortex-A15 CPU

We demonstrate the generality of PAVER by considering an ARM Cortex-A15 CPU with a 128-bit NEON SIMD unit, which does not support masked operations. As VL = 2 in double precision, partial SIMD parallelism is not exploitable. Therefore, we will focus on vectorizing the kernels listed in Table II in single precision, for which VL = 4.

ICC does not support ARM platforms. As before, GCC does not vectorize any kernel in Table II. Therefore, we will evaluate PAVER against LLVM in single precision.

For NEON, partial vector operations are performed similarly to the case of AVX, as described in Section 3. Therefore, we highlight only some differences in Figures 14 and 15, as well as Figure 16, by using again the example given in Figure 2(b). Figure 14 is an analogue of Figures 3 and 4 for ARM's NEON, except explicit packing may be avoided in data insertion. In NEON, a 128-bit quad-word register (q0 – q7) can be viewed as either two double-word registers or four single-word registers. By executing `vldr f[k][0], d0`, the two data elements `f[k][1]` and `f[k][0]` are loaded into the lower double-word of `q0`. By executing `vldr f[k][2], d1[0]` next, `f[k][2]` is loaded into the lower single-word of the upper double-word of `q0`. Similarly, Figure 15 is an analogue of Figures 7 and 8 for ARM's NEON, except explicit unpacking is unnecessary. Figure 16 is an analogue of Figure 6 on executing partial vector computations safely on ARM's NEON. In LLVM (version 3.6), the high-level IR instruction abstracted by `q0 =q0[0,1,2,0]` is always translated into the two **vext.32** instructions as shown, even though a single instruction **vmov.f32** suffices for duplicating the lowest single-word to the highest single-word of `q0`. Despite this, PAVER is shown to outperform LLVM for 15 out of the 16 kernels evaluated.

5.4.1. Speedups. Figure 17 compares PAVER and LLVM in terms of their kernel speedups achieved for the 16 kernels listed in Table II on the Cortex-A15 with VL = 4 over LLVM's scalar code. PAVER always opts to use widened vectors to perform partial vector loads/stores according to its cost model for the Cortex-A15. LLVM cannot vectorize any kernel when VL = 4. By setting VL = 2 (as 2-way single-precision

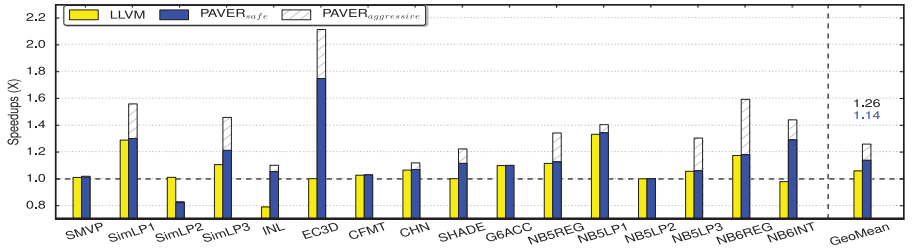


Fig. 17. Kernel speedups of PAVER and LLVM over LLVM's scalar code on ARM's Cortex-A15.

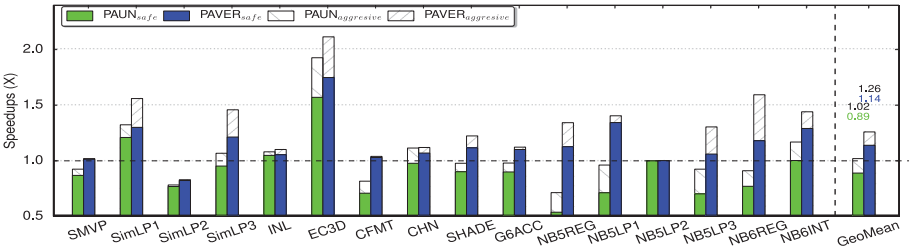


Fig. 18. Kernel speedups of PAVER and PAUN (over LLVM's scalar code) on ARM's Cortex-A15.

floating-point operations are supported by NEON but not AVX/SSE), LLVM has successfully vectorized all 16 kernels except EC3D. In safe mode, PAVER outperforms LLVM by 1.14x over LLVM's scalar code and 1.08x over LLVM's vector code, on average. In aggressive modes, these speedups are 1.26x and 1.19x, respectively.

PAVER is faster than LLVM in both safe and aggressive modes for all 16 kernels except SimLP2. As discussed in Section 5.3.2, effective handling of SimLP2 requires its division operations to be vectorized properly. However, the Cortex-A15 does not support vector division. For this kernel, PAVER has vectorized 3-way operations, whereas LLVM has exploited 2-way operations only. When processing each unpack-divide-pack sequence, PAVER suffers from a higher overhead than LLVM.

For the 16 hot kernels in Table II, only INL in 435.gromacs and CFMT in Fluidanimate are single-precision kernels. The whole-program speedups achieved by PAVER for 435.gromacs (Fluidanimate) over LLVM's scalar code are 1.05x and 1.06x (1.02x and 1.02x) in safe and aggressive modes, respectively. The speedups achieved by LLVM for 435.gromacs and Fluidanimate are 0.88x and 1.02x, respectively.

5.4.2. Comparing with Data Insertion/Extraction. Figure 18 compares PAVER and PAUN on ARM's NEON. Note that MASK is not available, as NEON does not support masked operations. PAVER is always better than PAUN for the kernels evaluated in both execution modes (safe and aggressive). On average, PAVER outperforms PAUN by 1.28x in safe mode and by 1.24x in aggressive mode. In safe mode, PAUN suffers more than PAVER, as the hardware resource for performing data reorganization (e.g., insertion, extraction, and permutation) can easily become the performance bottleneck.

5.5. Limitations

Let us summarize several limitations of PAVER. First, widening vector load/store requires tail padding, and the BR mechanism is not thread safe, as discussed in Section 3.2. Second, its cost model ignores some runtime features, such as cache-line split accesses and cache behavior. Third, LLVM's cost-benefit analysis, which is shared by PAVER, is sometimes inaccurate. Finally, PAVER does not directly address data

alignment issues. How to do so to boost the performance of PAVER further is interesting future work.

6. RELATED WORK

Loop-level vectorization. Earlier work [Eichenberger et al. 2004; Nuzman et al. 2006; Ren et al. 2006; Sreraman and Govindarajan 2000] applied strip mining to exploit parallelism across consecutive loop iterations by adopting the technology developed for vector machines [Zima and Chapman 1991]. Over the years, several problems have been addressed, including non–unit array accesses [Nuzman et al. 2006], nonunified data alignment [Eichenberger et al. 2004; Larsen et al. 2002], virtual vectors [Wu et al. 2005], polyhedral transformations [Kong et al. 2013; Trifunovic et al. 2009], branch divergence [Shin 2007; Sujon et al. 2013], function calls [Karrenberg and Hack 2011; Karrenberg 2015], and split vectorization [Nuzman et al. 2011]. Loop-level vectorization works well for loops with regular computations that exhibit little dependences but often poorly for loops with loop-carried dependences (caused by IAA and ICF in Table II), as evaluated here.

SLP vectorization. Larsen and Amarasinghe [2000] introduced the first SLP approach, which was later extended to operate on predicated basic blocks in the presence of control flow [Shin et al. 2005], to capture more superword reuse across the statement groups [Liu et al. 2012; Shin et al. 2002] and to expand opportunities for vectorization by dynamic programming [Barik et al. 2010]. Porpodas et al. [2015] focused on transforming nonisomorphic statement sequences into isomorphic ones. In this article, we have generalized the traditional SLP algorithm by exploiting partial SIMD parallelism on existing computing platforms.

Exploitation of mixed SIMD parallelism. Some existing vectorization techniques exploit both intraiteration parallelism (i.e., SLP) and interiteration (i.e., loop-level) parallelism in a loop [Rosen et al. 2007; Park et al. 2012; Zhou and Xue 2016]. Rosen et al. [2007] presented early work along this direction. Focusing on loops with small loop count, Park et al. [2012] considered how to effectively map isomorphic subgraphs in basic blocks to fully utilize SIMD resources based on the code generated by traditional loop-level vectorization. Zhou and Xue [2016] introduced an approach to exploit both intra- and interiteration SIMD parallelism simultaneously by reducing the data reorganization overhead incurred in vectorizing loops. All of these techniques do not handle partial vectorization.

Vectorization of irregular code. There are research efforts on vectorizing trees [Jo et al. 2013] and irregular data structures [Ren et al. 2013] with interleaved accesses such as FFTs [Kral et al. 2003; McFarlin et al. 2011] and irregular strides [Kim and Han 2012]. However, they cannot be applied to harness partial SIMD parallelism for the kernels considered in this article. Govindaraju et al. [2013] argued to expose as much of a specialized microarchitecture as possible to improve compiler effectiveness, particularly for irregular code.

User-assisted partial SIMD parallelization. We are not aware of any prior work on developing automatic vectorization techniques for exploiting partial SIMD parallelism with guaranteed correctness. ICC provides a compiler flag `-qopt-assume-safe-padding` [Green 2012] for the Intel MIC architecture to enable full vector operations for peeled and remainder loops only (due to strip mining).

Vectorization with data layout transformations. For a statically allocated multidimensional array, adding dummy elements at some appropriate locations inside (e.g., by changing `a[m][3]` to `a[m][4]`) can improve data alignment [Larsen et al. 2002] or

even completely avoid the overhead incurred by BR in performing partial vector stores. However, automating such data layout transformations by the compiler for whole programs can be difficult. Even if this is possible, existing compiler frameworks do not perform partial vectorization as PAVER does.

7. CONCLUSION

In this article, we have demonstrated the existence of partial SIMD parallelism in real-world programs. We have introduced a new compiler technique for vectorizing partially vectorizable loops by evaluating the effects of cache behavior, cache-line splits, hardware support, packing/unpacking, and masking on its effectiveness. To the best of our knowledge, this article is also the first to provide such compiler technology with guaranteed correctness by avoiding memory-related errors and numeric exceptions. Our evaluation shows significant performance benefits that can be achieved but missed by the state-of-the-art vectorization techniques implemented in several production-quality compilers.

REFERENCES

Sverre Aarseth. 2015. N-Body Simulation. Retrieved February 9, 2016, from http://www.ast.cam.ac.uk/research/nbody.

Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2010. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, Los Alamitos, CA, 201–212.

Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. 2002. Automatic intra-register vectorization for the Intel architecture. *International Journal of Parallel Programming* 30, 2, 65–98.

Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. 1999. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. ACM, New York, NY, 444–453.

Elena Demikhovsky. 2015. Implemented cost model for masked load/store operations. Retrieved February 9, 2016, from http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20150119/254753.html

Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 82–93.

Agner Fog. 2014. Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. Retrieved February 9, 2016, from http://www.agner.org/optimize/instruction_tables.pdf.

Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, Los Alamitos, CA, 341–352.

Ronald W. Green. 2012. Utilizing Full Vectors and Use of Option -Qopt-Assume-Safe-Padding. Retrieved February 9, 2016, from https://software.intel.com/en-us/articles/utilizing-full-vectors.

Q. Huang, J. Xue, and X. Vera. 2003. Code tiling for improving the cache performance of PDE solvers. In *Proceedings of the 2003 International Conference on Parallel Processing*. 615–624.

Intel. 2014. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-030. Retrieved February 9, 2016, from http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, Los Alamitos, CA, 363–374.

Ralf Karrenberg. 2015. *Automatic SIMD Vectorization of SSA-Based Control Flow Graphs*. Springer Vieweg.

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE, Los Alamitos, CA, 141–150.

Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 55–64.

M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 127–138.

Stefan Kral, Franz Franchetti, Juergen Lorenz, and Christoph W. Ueberhuber. 2003. SIMD vectorization of straight line FFT code. In *Euro-Par 2003 Parallel Processing*. Lecture Notes in Computer Science, Vol. 2790. Springer, 251–260.

Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, NY, 145–156.

Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. 2002. Increasing and detecting memory address congruence. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE, Los Alamitos, CA, 18–29.

Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 347–358.

Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, Los Alamitos, CA, 372–382.

Mantevo. 2015. The Mantevo Benchmark Suite. Available at http://mantevo.org.

Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 265–274.

Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE, Los Alamitos, CA, 151–160.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 132–143.

Yongjun Park, Sangwon Seo, Hyunchul Park, Hyoun Kyu Cho, and Scott Mahlke. 2012. 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 XVII)*. ACM, New York, NY, 363–374.

Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. IEEE, Los Alamitos, CA, 190–201.

Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE, Los Alamitos, CA, 1–10.

Gang Ren, Peng Wu, and David Padua. 2006. Optimizing data permutations for SIMD devices. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 118–131.

Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *Proceedings of GCC Developers' Summit (GCC Developers' Summit'07)*. 131–142.

Jaewook Shin. 2007. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE, Los Alamitos, CA, 280–291.

Jaewook Shin, Jacqueline Chame, and Mary W. Hall. 2002. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE, Los Alamitos, CA, 45–55.

Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE, Los Alamitos, CA, 165–175.

Narasimhan Sreraman and Ramaswamy Govindarajan. 2000. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* 28, 4, 363–400.

Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. 2013. Vectorization past dependent branches through speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, Los Alamitos, CA, 353–362.

Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE, Los Alamitos, CA, 327–337.

John Tsiombikas. 2015. C-Ray Raytracing Benchmark Results. Retrieved February 9, 2016, from http://www.futuretech.blinkenlights.nl/c-ray.html.

Michael Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing'89)*. 655–664.

Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. 2005. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, 169–178.

Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic, Norwell, MA.

Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the 21st International Symposium on Static Analysis (SAS'14)*. 319–336.

Hao Zhou and Jingling Xue. 2016. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'16)*.

Hans Zima and Barbara Chapman. 1991. *Supercompilers for Parallel and Vector Computers*. ACM, New York, NY.

Received August 2015; revised November 2015, December 2015; accepted January 2016