Rice University

Wave Equation Based Stencil Optimizations on a Multi-core CPU

by

Muhong Zhou

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Arts

Approved, Thesis Committee:

Dr. William W. Symes, *Chair*
Noah G. Harding Professor of Computational and Applied Mathematics

---

Dr. Béatrice M. Rivière
Professor of Computational and Applied Mathematics

Dr. Timothy Warburton
Professor of Computational and Applied Mathematics

Houston, Texas
October 2014

Abstract

Wave Equation Based Stencil Optimizations on a Multi-core CPU

by

Muhong Zhou

Wave propagation stencil kernels are engines of seismic imaging algorithms. These kernels are both compute- and memory-intensive. This work targets improving the performance of wave equation based stencil code parallelized by OpenMP on a multi-core CPU. To achieve this goal, we explored two techniques: improving vectorization by using hardware SIMD technology, and reducing memory traffic to mitigate the bottleneck caused by limited memory bandwidth. We show that with loop interchange, memory alignment, and compiler hints, both icc and gcc compilers can provide fully-vectorized stencil code of any order with performance comparable to that of SIMD intrinsic code. To reduce cache misses, we present three methods in the context of OpenMP parallelization: rearranging loop structure, blocking thread accesses, and temporal loop blocking. Our results demonstrate that fully-vectorized high-order stencil code will be about 2X faster if implemented with either of the first two methods, and fully-vectorized low-order stencil code will be about 1.2X faster if implemented with the combination of the last two methods. Our final best-performing code achieves 20%$\sim$30% of peak GFLOPs/sec, depending on stencil order and compiler.

Acknowledgements

I am sincerely grateful to my advisor, Professor Symes, for taking me as his student and providing the guidance and valuable advices along working on this project. He has granted me ample opportunities to attend high performance computing and geophysics conferences, workshops, seminars. Without these exposures, I would have never thought of doing geophysics research and pursuing a career in the energy industry. Despite my slow progress and poor writing skills, he had showed great patience in helping me improve this thesis and presentations.

Also, I would like to thanks Professor Rivière and and Professor Warburton for being my committee members and teaching me the numerical analysis/methods and parallel computing courses when I was in my first year. They are the most interesting and challenging courses at CAAM.

Thanks my parents for their encouragement.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

2 Literature Review 5

3 Wave Equation Based FDTD Stencil Codes 10

3.1 Acoustic Constant Density Wave Equation 10

3.2 CFD Approximation 11

3.3 FDTD Stencil Kernel 12

3.4 Implementations of Boundary and Initial Conditions 15

4 Improving Vectorization 17

4.1 SIMD Technology 17

4.2 Explicit Vectorization 19

4.3 Auto-vectorization with GCC and ICC Compilers 20

5 Cache Optimization Methods 25

5.1 Memory Bottleneck 25

5.2 Source of Memory Traffic 29

5.3 Thread-blocking Method 31

5.4 Separate-and-interchange Method 36

5.5 Parallelized Time-skewing Method 38

6 Results and Discussion **41**
6.1 Experiment Setup 41
6.2 Vectorization 42
6.3 Cache Optimization 46
6.4 Summary 49

7 Conclusion **51**

A Xeon E5-2660 Latency Curve **53**

B Finite Difference Coefficients **55**

C Boundary Effects **56**

D Sandy Bridge Instruction Table **57**

E Effects of Memory Alignment on Compiler Auto-vectorization **58**
E.1 Effects of Memory Alignment on gcc Auto-vectorization 58
E.2 Effects of Memory Alignment on Intel Auto-vectorization 61
E.3 Memory Alignment and Event Counters 62

F Perf Setup **63**

References **64**

List of Figures

1.1 The Architecture of Our Test Device 3

3.1 Stencil Operation 14
3.2 Implementation of Dirichlet Boundary Condition 15

4.1 Difference between Scalar and Packed SIMD Instructions 18
4.2 SSE Vectorization of Stencil Codes 20
4.3 Pad 4th order SC on 32-byte Boundary 24

5.1 Roofline model based on Xeon E5-2660. 28
5.2 Snapshot of Conventional OpenMP Thread Access 30
5.3 OpenMP thread access pattern when adding `schedule(static,1)`. . 32
5.4 No Cache Reloads Along `i`, `j` Dimensions 34
5.5 Illustration of The Optimal Case With Thread-Blocking 35
5.6 Separate-and-exchange Method 36
5.7 Parallelized time-skewing method 39
5.8 The Range of `U_out` and `U_in` `k`-planes Associated With Two Consecutive Rounds 40

6.1 SIMD Instruction Composition Stacked Bar Chart of Each Kernel . . 43
6.2 L3 Cache Misses of Each Kernel 44
6.3 Run Time of Each Kernel 45
6.4 Cache Misses and Run Time of the Stencil Kernels Implemented with Parallelized Time-skewing Method 48
6.5 Contribution of Each Optimization Method 50

A.1 Xeon E5-2660 Latency Curve 54

List of Tables

1.1 Yearly Growth Rate of The Performance of Each Component 3

4.1 SIMD Registers and Instruction Sets 18

5.1 Loads and Stores Per Time Step If Using Conventional OpenMP parallelization 30
5.2 Loads and Stores Per Time Step With Thread-blocking 33
5.3 Optimal Loads and Stores Per Time Step of Thread-blocking Method 35
5.4 Optimal Loads and Stores Per Time Step With Separate-and-exchange Method 38
5.5 Optimal Loads and Stores Per Time Step With Parallelized Time-skewing Method 40

6.1 L3 Cache Misses of Each Kernel 47
6.2 Run Time of Each Kernel 49

C.1 AUTOVEComp Run Time With(out) Boundary 56

Listings

3.1 NAIVE Kernel 13
3.2 EXTEND Kernel 14
4.1 AUTOVEC Kernel 21
5.1 Stencil kernel implemented with separate-and-interchange method . . 36
E.1 i-loop-1 in AUTOVEC. 58
E.2 Memory unaligned gcc assembly codes. 58
E.3 Memory Aligned gcc Assembly Codes. 60
E.4 Memory Unaligned Intel Assembly Codes. 61
E.5 Memory Aligned Intel Assembly Codes. 61

Chapter 1

Introduction

Finite Difference Time Domain (FDTD) wave propagation kernels are engines of seismic imaging algorithms, which help to characterize the subsurface structure and locate the underlying interested targets. This work aims to optimize wave propagation kernel on an Intel Sandy Bridge processor (Xeon E5-2660) by implementing *single instruction, multiple data* (SIMD) vectorization and cache optimization methods.

The FDTD method discretizes the wave fields on a structured grid, and approximates both time and spatial derivatives by using the *Central Finite Difference* (CFD) schemes [Moczo et al., 2007]. The resulting kernels update wave field values on each grid point over one time step by using a set of formulas called the *FDTD stencil*. The stencil updates a field value by using the field value itself and the nearby field values in a regular pattern repeated at every grid point.

The repetitive feature of *FDTD stencil* kernel makes it very easy to parallelize. Modern CPUs provide several levels of parallelizations. Specially, in-core data-level parallelization can be provided by SIMD technology built into most contemporary x86-based CPUs, for instance, those manufactured by Intel and AMD. SIMD support includes vector registers and packed instruction sets that operate on them, executing several same operations simultaneously and providing data level parallelization.

The latest Intel (icc) and Free Software Foundation (gcc) compilers can automatically replace original C codes with SIMD packed instructions, a process knowns as *vectorization*. However, if the loop structure has certain features, such as non-unit stride accessing, or statement dependency [Intel, 2012], then SIMD technology may not be fully exploited, that is, the compiler will not be able to maximize the percentage of SIMD packed instructions in total instructions, nor maximize the number of SIMD registers in use concurrently.

One way to avoid such inefficiency is to code with SIMD intrinsics (C function interfaces to SIMD assembly instructions). Expression of algorithms with SIMD intrinsic functions typically requires almost complete rewrite of conventional C sources. The programmer will need to deal with data movements between SIMD vector registers and main memory (DRAM), and explicit loop unrolling that improves the number of SIMD registers in use concurrently.

The first part of this work focuses on having the compilers fully auto-vectorize *FDTD stencil* kernel of any order with minimal effort: by rearranging loop structure and adding compiler hints/options, with the expectation that the auto-vectorized codes could achieve comparable performance to codes using SIMD intrinsic functions, and compatible with the latest icc and gcc compilers.

Even though the stencil kernels are fully-vectorized, the performance gap between main memory (DRAM) and cache may still prevent stencil kernel from obtaining machine performance peak. As an example, Figure 1.1 shows the architecture of our test device and marks the position of memory bottleneck. The practical problem size usually exceeds the cache capacity, or even DRAM capacity [Etgen and O'Brien, 2007], so there will be extensive data movements between DRAM and cache every time step. As presented in Table 1.1, CPU speed grows much faster than DRAM to cache speed and bandwidth, so it is possible that sometimes the CPUs are in idle state waiting

for data references from DRAM.

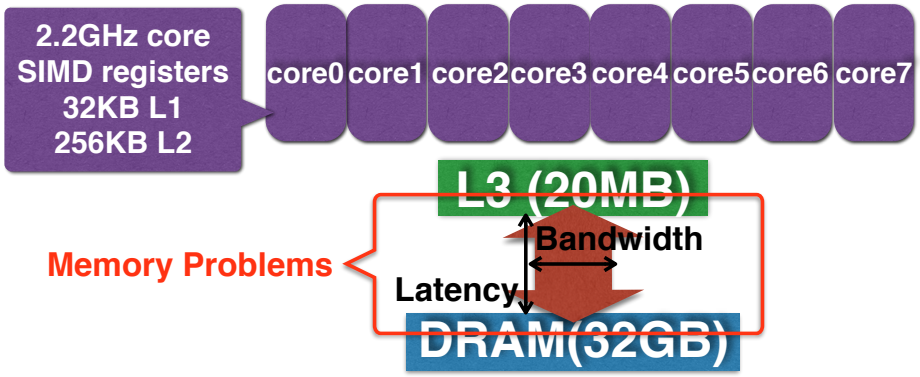


Figure 1.1: The architecture of our test device.

CPU speed	Memory speed	Memory Bandwidth
~60%	~7%	~23%

Table 1.1: Yearly growth rate of the performance of each component. [Demmel, 2014]

Memory latency can be hidden by implementing short-stride accessing that improves cache line hit rates and engages hardware prefetching to keep the memory pipeline busy (see Appendix A). The effects of limited memory bandwidth can be mitigated by implementing cache optimization methods, which improve cached data utilization so as to reduce memory traffic.

The second part in this thesis evaluates several cache optimization methods on FDTD stencil kernel: thread-blocking, separate-and-interchange, and time-skewing methods.

Examples in this thesis use FDTD algorithm with its time derivative approximated by 2nd order CFD scheme. The orders of the CFD schemes for spatial derivatives are 4 or larger, depending on the accuracy and run-time requirements [Etgen and O'Brien, 2007]. For a symmetric *stencil*, where the orders of the CFD schemes are the same in each spatial dimension, *order* refers to this unique order

of all the spatial CFD schemes. In this work, I will evaluate the effectiveness of each cache optimization methods on *stencil codes* (SCs) of both low-order(=4) and high-order(=16), i.e., SC4 and SC16.

In addition, to accommodate the multi-core trend, all codes are parallelized on Xeon E5-2660 using OpenMP, so the cache optimizations are all in parallelized as well.

To summarize, this work focuses on optimizing wave propagation FDTD stencil kernel, on a single Intel Sandy Bridge processor. The methods are:

- Two levels of parallelization: OpenMP for core-level parallelization and compiler auto-vectorization for data-level parallelization. Auto-vectorization aims to achieve performance similar to that of tuned intrinsic codes with minimal source code modification, works for stencil codes of any order, and is compatible with both the latest icc and gcc compilers.

- Three cache optimizations: thread-blocking, separate-and-exchange, time-skewing methods. All of them work for OpenMP parallelized codes, and are evaluated in both low-order(=4) and high-order(=16) cases.

All the codes are available at `https://svn.rice.edu/r/mz10/MA/DocCodes/`

The rest of the thesis is organized as follows: Chapter 2 reviews relevant literature; Chapter 3 introduces the background and formulation of the *FDTD stencil* codes; Chapter 4 describes my intrinsics codes and strategies to assist compiler to auto-vectorize the original C source codes; Chapter 5 presents three cache optimization methods; Chapter 6 shows results of performance studies; Chapter 7 concludes the thesis.

Chapter 2

Literature Review

Most of previous literature on stencil optimization work only on very low-order (=2) stencils [Datta et al., 2008, **?**, Datta, 2009, Nguyen et al., 2010, Henretty et al., 2011, Strzodka et al., 2011, Zumbusch, 2012, Wonnacott and Strout, 2012]. These strategies may not work effectively for high-order(=16 or more) stencils, or even the low-order(=4) stencils. Besides, most of the targeted stencils in these papers are of Jacobi type: they do not use the old value at the center of the stencil to update itself, hence one less FLOP executed per stencil compared with stencils considered here. Also, their test platforms are not Xeon E5-2660. Various machine characteristics (such as the memory bandwidth) may not be the same, so it's hard to predict our test results from theirs or compare our test results with theirs. However, these papers still propose a variety of optimization directions worth trying.

Most of the literatures on stencil vectorization use SIMD intrinsics [Datta et al., 2008, **?**, Datta, 2009, Dursun et al., 2009, Henretty et al., 2011, Strzodka et al., 2011, Dursun et al., 2012, Zumbusch, 2012, Zumbusch, 2013], and [Datta et al., 2008, Dursun et al., 2012] explicitly claimed that their compilers had failed to auto-vectorize the stencil codes. Borges [Borges, 2011] gives an example of auto-vectorizing an 8th order stencil kernel, however their procedures only work for stencils of a fixed order because all the

finite difference terms are explicitly written in their scheme, so whenever the stencil order changes, the codes need to be rewritten.

My tuned SIMD intrinsic codes are primarily based on Dr. Christof Stork's flexSIMD package (Stork, personal communication, 2013). Stork points out that explicit unrolling, aligned vector loads, cache bypassing stores would further improve the naive intrinsic codes' performance, and these ideas are integrated in my codes. In addition, [Fog, 2013, Intel, 2014] provide latency and throughput of every SIMD instruction, and they are good references when choose between SIMD intrinsics having similar functions.

Intel [Intel, 2012] lists some auto-vectorization criteria and compiler hints/options to assist compiler auto-vectorization. The auto-vectorization strategies for FDTD stencil kernel in this work are primarily extracted from this article.

Although for AVX instructions (SIMD instruction sets used on Xeon E5-2660), aligned vector loads requirement is relaxed, still unaligned vector loads may cause performance penalty [Lomont, 2012], so array memory should be padded to ensure maximal aligned vector loads. Henretty [Henretty et al., 2011] and Zumbusch [Zumbusch, 2012] present an interleaved memory allocation method to make every vector load aligned, especially for aligning every load for accumulating finite difference terms along unit-stride dimension, which cannot be obtained via normal 3D stencil allocation as demonstrated in by Borges [Borges, 2011]. However, if boundary condition is considered, as in my work, halo layers are built to keep stencils the same throughout the computation domain, then the thickness of a halo layer is the the product of vector length and ($order$/2-1). For high-order stencils and long vector length, the extra time to build the halo layers may offset the time reduction due to the aligned vector loads. Also, the vectorization experiments in this work show that, even though memory alignment generates fewer instructions per statement and higher unrolling factors, the run-time

barely reduces for the Xeon E5-2660. So this interleaved memory method is not implemented in this work.

Although memory bandwidth is growing slower than CPU speed, kernels may or may not be bandwidth-bound, depending on the problem size, kernel's *operation intensity* (FLOPs executed per bytes movement between cache and DRAM) [Williams et al., 2009], and the cache capacity of the platform. If the problem size exceeds the cache capacity, and its *operation intensity* is smaller than the *machine balance* (ratio between machine peak GFLOPs/sec and its sustained memory bandwidth) [McCalpin, 1995], then the problem is bandwidth bound, otherwise it is compute bound. The roofline model [Williams et al., 2009] based on this idea further predicts how much performance can degrade due to bandwidth for a given kernel and platform pair.

Two categories of methods are effective in reducing memory traffic: purely spatial blocking methods and time-skewing methods, combining spatial and temporal blocking. Due to increasing cache capacity, purely spatial blocking can hardly improve stencil code performance on modern CPUs [**?**, Nguyen et al., 2010]. Note that it's better to prevent the unit-stride dimension from being blocked, as the prefetching engine needs warm-up before working at its peak, and intermittent recesses could degrade its performance [**?**].

Time-skewing method was first proposed in 1999 [Wonnacott, 1999], subsequently the serial version of this method has been observed to significantly improve low-order stencil performance by reducing memory traffic between CPU and DRAM [**?**] or even between DRAM and hard disk [Etgen and O'Brien, 2007]. Zumbusch [Zumbusch, 2012] applied this method on a 2nd order 1D stencil on various single/multi-core Intel and AMD CPUs, and achieved above 84% peak performance on all the CPUs.

There are various ways to parallelize time-skewing method to accommodate multi-core processors in terms of scalability, load balance and concurrent start [Wonnacott and Strout, 201
All of these methods assign each spatial block to a single thread. The difference between any two methods lies in the shape and traversing order inside each spatial block. Nguyen [Nguyen et al., 2010] proposed a parallelized time-skewing method with each spatial block parallelized by OpenMP (without thread-blocking), i.e., each spatial block is assigned to multiple threads, and only two most contiguous spatial dimensions are blocked. This variant gives 1.5x speed up for 2nd order 3D stencil of size 256^3 on a quad-core 3.2GHz Intel CPU.

Even though time-skewing has sped up low-order stencils, the results of time-skewing on high-order stencils are not optimistic. The CATS algorithm [Strzodka et al., 2011] (time-skewing with diamond spatial block scheme) demonstrated that on a quad-core 3.2GHz Xeon X5482, the GFLOPs/sec dropped by 2.8x when the order of a 3D stencil increased from 2 to 6. Zumbusch [Zumbusch, 2013] also observed the performance dropped by 3x on a single-core Sandy Bridge Intel i7-2600 when the order of a 3D stencil increased from 2 to 12.

To reduce memory traffic for high-order stencils, Stork (Stork, 2013, personal communication) employed a novel approach that is different from the aforementioned blocking methods. It used no spatial or temporal blocking. It partitioned the stencil into 3 parts, and each part updates the central point along a fixed dimension. Accordingly, one 3-fold loop structure is partitioned into three 3-fold loops, each responsible for update along each dimension. Stork interchanged the two outermost loops for the loop structure corresponding to the least contiguous dimension, so that more points could get reused along the least contiguous dimension before being ejected from the cache. I provide a more detailed explanation of this separate-and-interchange method in Section 5.4

Other relatively less effective methods include: cache bypassing [Datta et al., 2008], which uses SIMD bypassing stores when array memory is properly aligned, and the sliding window algorithm [Zumbusch, 2013], which explicitly unrolls the loop and re-uses the register data as many times as possible before loading new data.

Chapter 3

Wave Equation Based FDTD Stencil Codes

This chapter sets up the mathematical equation and algorithms used for the codes developed for this thesis. Section 3.1 introduces the acoustic constant density wave equation, which is the most simple representation of all the wave equations used for seismic modeling research. Section 3.2 details the finite difference scheme used to discretize the derivatives in the wave equation. The resulting finite difference codes are shown in Section 3.3. Section 3.4 demonstrates how the boundary condition is implemented in the codes.

3.1 Acoustic Constant Density Wave Equation

Small amplitude wave propagation in fluid media can be described by the following equation:

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} = \kappa(x,y,z)\nabla\cdot(\frac{1}{\rho(x,y,z)}\nabla u(x,y,z,t)), \quad (x,y,z)\in\Omega, \quad t\in\mathbb{R}. \quad (3.1)$$

Here Ω is the computation domain; u is the pressure field, a scalar function of time and space variables; κ and ρ are the bulk modulus and density fields respectively, and

both of them characterize properties of the propagation medium.

In this work, the research is further narrowed down to the case when density is constant o'ver Ω. Then the above equation becomes:

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} = c^2(x,y,z)\nabla^2 u(x,y,z,t), \quad (x,y,z) \in \Omega, \quad t \in \mathbb{R}. \tag{3.2}$$

Here $c^2(x,y,z)$ is the squared velocity field, which equals to $\kappa(x,y,z)/\rho(x,y,z)$. This equation is known as the *acoustic constant density wave equation* (ACDEQ).

If the bulk modulus field is also constant over Ω, then $c^2(x,y,z)$ will be reduced to a constant c.

3.2 CFD Approximation

Finite difference methods are often used in practice to numerically simulate wave propagation because they are easy to implement. This method works on a rectangular grid discretizing the computational domain. Every partial derivative in the PDE is approximated by a linear combination of values at its nearby points. Hence every point updates its own value by using values at the surrounding (spatially or temporally) grid points by the same pattern called *stencil*.

Depending on how derivatives are linearized, *finite difference methods* can be categorized as *forward*, *backward*, and *central finite difference methods* [Moczo et al., 2007]. In this thesis we use the *central finite difference method* (CFD) to approximate both spatial and temporal derivatives of the wave equation. See Appendix B for CFD coefficients of various orders. For example, the second order CFD approximation to u_{tt} is:

$$u_{tt}(\vec{x},t) = \frac{u(\vec{x},t-\Delta t) - 2\,u(\vec{x},t) + u(\vec{x},t+\Delta t)}{\Delta t^2} + \mathcal{O}(\Delta t^2)$$

In practice, the time derivative of the wave equation is often approximated by second order CFD scheme. For spatial derivatives, the order of the CFD scheme varies based on the requirement on accuracy and run-time. For example, some reflection problems require wave propagation with less dispersion along vertical directions, thus the order of CFD should be larger; for some refraction problems of wave propagation over large horizontal distance, accuracy should be sacrificed for less run-time, thus CFD scheme with smaller order vertical derivative in z (depth) will be used [Etgen and O'Brien, 2007].

Recognizing that both low and high order stencils are important, in this work I will evaluate the effectiveness of each optimization method on CFD codes with both low- (=4) and high- (=16) spatial truncation error order. For simplicity, the order of CFD schemes used along all the spatial directions are the same.

3.3 FDTD Stencil Kernel

Since the second order CFD scheme for u_{tt} involves data values from three time steps, it needs at least two arrays to store temporary pressure field values. To minimize storage, I use `U_in` to store pressure field values at current time step t, and `U_out` to store pressure field values at previous time step $(t - \Delta t)$ and the next time step $(t + \Delta t)$.

We allocate both arrays with 3D indexing, i.e., `U_in[k][j][i]` would represent the acoustic field value at grid point $(\mathtt{i}\Delta x, \mathtt{j}\Delta y, \mathtt{k}\Delta z)$, where `i` loops over the unit-stride (or the most contiguous) dimension, `j` loops over the second most contiguous dimension, and `k` loops over the least contiguous dimension. Δx, Δy, and Δz are the corresponding grid sizes. Together with time step size Δt, the space steps must satisfy CFL condition [Moczo et al., 2007] to ensure convergence. `NX`, `NY`, `NZ` are the number of points in each dimension of the computation domain.

Listing 3.1 shows the resulting wave propagation simulation kernel. This kernel is not optimized for performance on any CPU. Optimizations discussed later in this work will be implemented as modifications of this kernel, so this kernel is named the NAIVE stencil kernel. The code body inside the `i`-loop is the *stencil operation*, which is the basic component of stencil codes.

```
/* FDTD stencil kernel -- NAIVE */
for(t = 1; t <= NT; t++)
  for(k = 1; k <= NZ; k++)
    for(j = 1; j <= NY; j++)
      for(i = 1; i <= NX; i++){
          /* c0, cx[ixyz], cy[ixyz], cz[ixyz] are coefficients combining both finite
              difference coefficients and velocity field value c. */
          /* r is the stencil radius, also equals to half of the spatial truncation
              error order */
          U_out[k][j][i] = - U_out[k][j][i] + c0*U_in[k][j][i];
          for(ixyz = 1; ixyz <= r; ixyz++)
             U_out[k][j][i] += cx[ixyz]*(U_in[k][j][ixyz+i]+U_in[k][j][-ixyz+i]);
          for(ixyz = 1; ixyz <= r; ixyz++)
             U_out[k][j][i] += cy[ixyz]*(U_in[k][ixyz+j][i]+U_in[k][-ixyz+j][i]);
          for(ixyz = 1; ixyz <= r; ixyz++)
             U_out[k][j][i] += cz[ixyz]*(U_in[ixyz+k][j][i]+U_in[-ixyz+k][j][i]);
  /* + process certain boundary condition */}
```

Listing 3.1: NAIVE Kernel

Figure 3.1 visualizes this operation using 2-4 CFD scheme. Every point is associated with a `U_in` value, and only the central point is associated with a `U_out` value, and this `U_out` value is updated by using itself and the nearby `U_in` values on the stencil according to the rule given by the *stencil operation*. Then completing the 3-fold loop structure (`k`, `j`, `i` loops) can be visualized as sweeping the *stencil operation* over the entire computation domain, which updates the entire `U_out` array values by one time step.

In IWAVE [Symes, 2013, Terentyev, 2009], our research group's software framework for seismic modeling and inversion, the stencil kernel in the acoustic constant

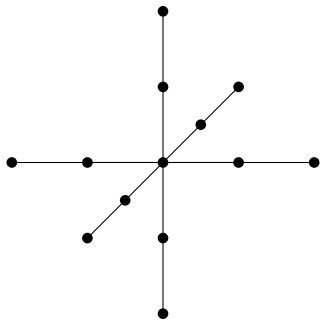


Figure 3.1: A 3D stencil operation using 2-4 CFD scheme

density package are written with the coefficient loops fully unrolled. We denote this kind of kernel as EXTEND. Listing 3.2 gives an example of EXTEND using 2-4 CFD scheme.

```
/* FDTD stencil kernel -- EXTEND*/
/* Uses CFD of 2nd order in time and 4th order in space */
for(t = 1; t <= NT; t++)
  for(k = 1; k <= NZ; k++)
    for(j = 1; j <= NY; j++)
      for(i = 1; i <= NX; i++){
      /* c0, cx[ixyz], cy[ixyz], cz[ixyz] are coefficients combining both finite
          difference coefficients and velocity field value c. */
            U_out[k][j][i] = - U_out[k][j][i] + c*U_in[k][j][i]
            + cx[1]*(U_in[k][j][1+i]+U_in[k][j][-1+i])
            + cx[2]*(U_in[k][j][2+i]+U_in[k][j][-2+i])
            + cy[1]*(U_in[k][1+j][i]+U_in[k][-1+j][i])
            + cy[2]*(U_in[k][2+j][i]+U_in[k][-2+j][i])
            + cz[1]*(U_in[1+k][j][i]+U_in[-1+k][j][i])
            + cz[2]*(U_in[2+k][j][i]+U_in[-2+k][j][i]);
          /* + process certain boundary condition */}
```

Listing 3.2: EXTEND Kernel

The coding style of EXTEND adds complexity for programmers as stencil kernel of different order requires a different coding implementation, and the length of its innermost loop structure increases with the order of stencil. In views of its flexibility in changing the stencil order, this work will focus on optimizations for NAIVE kernel,

and use EXTEND as a performance reference instead. Later the work will show that the final best-tuned NAIVE code outperforms EXTEND code at any stencil order.

3.4 Implementations of Boundary and Initial Conditions

The computation domain boundaries are processed with *Homogeneous Dirichlet* condition in my experiments. Instead of changing the stencil scheme near the boundaries, I allocate ghost points to store the temporary values so that the stencil scheme stays the same throughout the entire computation domain. Figure 3.2 demonstrates on how to use ghost cells to keep stencil pattern the same in 2D case. The computation domain includes the solid white points and solid colored points. The dashed white points represent the ghost cells. Once finish updating all the `U_out` values in the computation domain, `U_out[2][-1]=-U_out[2][1]`. In general, the value of a ghost cell is set to be the negative of the value of its mirror point with respect to the computational domain face nearby, on which array values are kept constant zero.

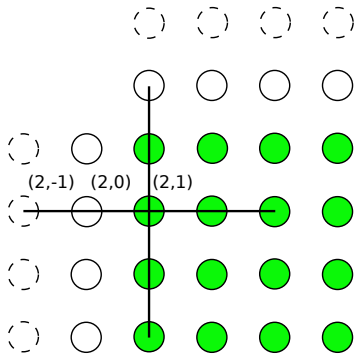


Figure 3.2: Illustration on using ghost points in 2D case with 2nd order SC.

The initial pressure field is set to be one at the center of the computational domain and zero elsewhere, which resembles a small initial disturbation in the middle. And we set the time derivative of the initial pressure field also be zero throughout the

domain.

Note that the second-order CFD approximation for u_t at $t = 0$ is:

$$u_t(x, y, z, 0) \approx \frac{u(x, y, z, \Delta t) - u(x, y, z, -\Delta t)}{2\Delta t} \tag{3.3}$$

This motivates us to set $u(x, y, z, t+\Delta t) = u(x, y, z, t-\Delta t)$ to approximate a zero derivative.

Computing the initial condition only requires one iteration, while updating boundary condition occurs at all iterations. Considering the large iteration number (5001) used in our experiments and small time duration per iteration (< 0.03 sec), how the initial condition is implemented barely affects the final performance of the stencil codes, and experiments in Appendix C demonstrate the time spent in computing the boundary condition is also small compared with the stencil computation. So the primary optimization task lies in optimizing the stencil computation.

Chapter 4

Improving Vectorization

The Intel Sandy Bridge processor provides vector registers and SIMD instructions that can process four or eight *single precision floating point* (SPFP) operations simultaneously. This chapter will first give an overview of SIMD technology, then present manual and automatic approaches to apply this technology on stencil codes, and lastly, it will describe the array padding technique used to produce aligned memory that could reduce the number of load/store instructions in stencil codes.

4.1 SIMD Technology

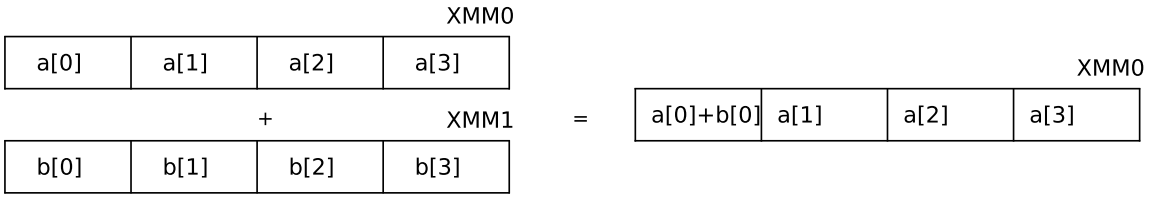
Single Instruction, Multiple Data (SIMD) technology is widely adopted in the modern x86 based CPUs, e.g., Intel, AMD CPUs. This technology exploits data-level parallelism by issuing only one instruction to process several operations of the same type simultaneously, thus improving the FLOPs executed per CPU cycle.

As listed in Table 4.1, SIMD technology consists of two parts: SIMD registers of various lengths and SIMD instruction sets that operate on the corresponding SIMD registers. Our test device is a Sandy Bridge processor, which contains XMM/SSE and YMM/AVX registers.

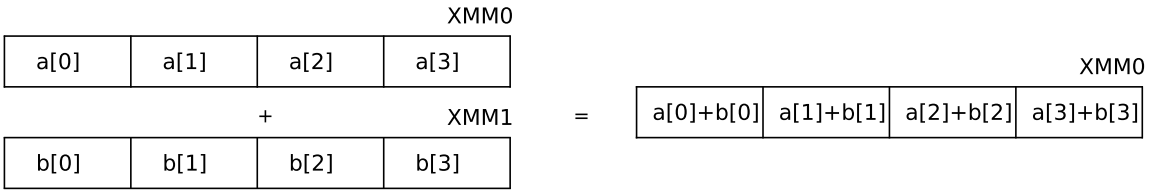
SIMD registers	XMM (128-bit)	YMM (256-bit)	ZMM (512-bit)
SIMD instruction sets	SSE	AVX	AVX-512

Table 4.1: SIMD registers and instruction sets.

SIMD instruction sets provide two kinds of instructions to process FLOPs: *scalar SIMD instructions* and *packed SIMD instructions*. The *scalar* one only performs the operation on the first data pair, while the *packed* one performs the same operations on all the data pairs simultaneously, forming the so-called *vector operation*. Figure 4.1 illustrates this difference using XMM/SSE.



(a) Scalar SSE performs 1 SPFP additions.



(b) Scalar SSE performs 4 SPFP additions simultaneously

Figure 4.1: Difference between Scalar and Packed SIMD Instructions

Almost all of the stencil codes' FLOPs are executed by using these two forms of SIMD instructions. Very few (order of magnitudes smaller) FLOPs are executed by using X87 instructions on the non-SIMD registers. Appendix D shows that it takes the same latency and throughput to complete a scalar instruction and its equivalent packed instruction. Therefore, the code performance will be improved by processing more floating point operations with packed SIMD instructions. This procedure is called *vectorization*.

Currently there are three approaches to get vectorization:

1. Writing assembly code.

2. Using SIMD intrinsic functions [Intel, 2014], which are C function interfaces providing direct control over the generation of SIMD instructions.

3. Adapting code structure and adding compiler hints to assist compiler to auto-generate SIMD instructions. [Intel, 2012, Borges, 2011]

In this work, I focus on improving vectorization using the last two methods. As opposed to traditional C codes, in this work, codes using SIMD intrinsics are referred as *intrinsic codes*, and codes generated from the third approach are referred as *auto-vectorized codes*.

4.2 Explicit Vectorization

SIMD intrinsics provide C function interfaces that directly control the generation of assembly vectorization instructions. So we can apply them directly on NAIVE kernel.

Figure 4.2 illustrates the vectorization strategy using XMM/SSE. The basic principle is that the consecutive stencils are grouped into pack of 4 (length of the XMM register) along unit-stride dimension, then each segment along x dimension can be loaded into a XMM register, then call SSE instructions to compute 4 stencils at a time.

As employed by Stork (personal communication, 2013), explicit loop unrolling at i-loop level can keep all the vector registers busy simultaneously and reuse coefficients that have already loaded into vector registers. So this work will also evaluate the effectiveness of this trick by benchmarking the intrinsic codes that have been explicitly unrolled for various times.

Since SIMD intrinsics directly control SIMD instruction generation, SIMD code performance is less affected by compiler heuristics. However, manually porting C

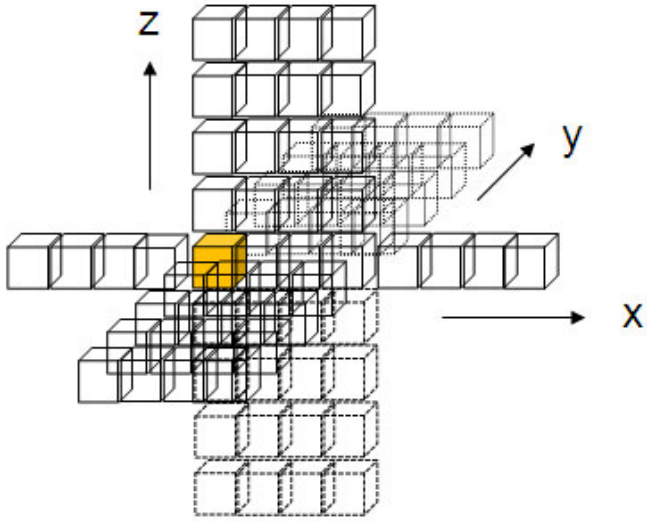


Figure 4.2: SSE vectorization of stencil code [Borges, 2011]. x is the most contiguous dimension, and z is the least contiguous dimension.

codes to SIMD codes demands considerable effort from the programmer.

4.3 Auto-vectorization with GCC and ICC Compilers

The principle of automatic vectorization is the same as that of manual vectorization. Both Intel and GCC compilers are able to generate vector instructions from C source code, that is, *auto-vectorize* loops, given that the loops have met certain criteria.

Current compilers are so cautious that they sometimes may refuse to auto-vectorize loops that are potentially vectorizable. One can never be sure about whether a loop is auto-vectorized or not, before checking the compiler vectorization report or the assembly code. Because of these elusive compiler heuristics, it is hard to provide complete criteria for reference, but some common features that prevent auto-vectorization include [Intel, 2012]:

1. The loop is not the innermost loop.

2. Loop count is low (<8).

3. Array accessing is not contiguous.

4. Existence of potential data dependencies.

NAIVE kernel is a typical loop structure that cannot be auto-vectorized. Because of item 1 above, only `ixyz`-loops in NAIVE kernel can be vectorized. However, when `order` is under 8, short length of `ixyz`-loops prevent auto-vectorization again (item 2). Even if `order` is greater than 8, the last two `ixyz`-loops, `U_in[k][ixyz+j][i]` and `U_in[ixyz+k][j][i]` do not access data contiguously with `ixyz` as the loop index, hitting another obstacle (item 3).

To circumvent these obstacles, I revised the loop structure in NAIVE kernel. The new kernel AUTOVEC is in Listing 4.1. AUTOVEC divides the original i-loop in NAIVE kernel into two parts, i-loop-1 and i-loop-2, then it interchanges ixyz-loop and i-loop-2. Now the innermost loops are i-loops, in which any data references are contiguous. Also, loop count is not a problem because NX is usually set long enough to ensure long unit-stride accessing, and there are no logical data dependencies inside i-loops.

```
/* FDTD stencil kernel -- AUTOVEC */
for(t = 1; t <= NT; t++)
  for(k = 1; k <= NZ; k++) //k-loop
    for(j = 1; j <= NY; j++){ //j-loop
      for(i = 1; i <= NX; i++) //i-loop-1
          U_out[k][j][i] = c0*U_in[k][j][i] - U_out[k][j][i];
      for(ixyz = 1; ixyz <= r; ixyz++){ //ixyz-loop
          #pragma ivdep
          for(i = 1; i <= NX; i++)  //i-loop-2
               U_out[k][j][i] += cx[ixyz]*(U_in[k][j][ixyz+i]+U_in[k][j][-ixyz+i])
                   + cy[ixyz]*(U_in[k][ixyz+j][i] +U_in[k][-ixyz+j][i])
                   + cz[ixyz]*(U_in[ixyz+k][j][i]+U_in[-ixyz+k][j][i]);}}
  /* + process certain boundary condition */ }
```

Listing 4.1: AUTOVEC Kernel

After checking the vectorization reports by both compilers (ICC: `-vec-report[n]`, GCC: `-ftree-vectorizer-verbose=[n]`, `n` controls the detail extent, `n` is from 0 to 6), I found that only GCC compiler could vectorize two i-loops. Intel compiler could only vectorize i-loop-1, but failed to vectorize i-loop-2 in which it assumed potential data dependencies. To remove this assumption, I added `ivdep` pragma to notify the Intel compiler that there is no data dependencies. After that, Intel vectorization report showed that i-loop-2 was vectorized.

In addition, if the stencil is asymmetric, which corresponds to different accuracy on different dimensions, this method still works, except that there should be three ixyz-loops instead of one ixyz-loop, and each ixyz-loop should be interchanged with i-loop to make i-loop the innermost loop.

Since the Sandy Bridge machine has 16 XMM/YMM registers, another key to improve code performance besides getting more packed vector instructions is to keep more vector registers busy at the same time, i.e., keep the pipeline busy, or in other words, improve the unrolling factor. Based on default compiler heuristics, the compiler option `-funroll-loops` can automatically unroll the innermost loop and maximize the unrolling factor till it reaches 16.

Compared with NAIVE or EXTEND kernel, in which the stencil traverses the entire computational domain only once per time step, stencil of AUTOVEC traverses the domain twice per time step because of the partition to i-loop. In consequence, entire `U_in` array will be loaded into registers at least twice per time step, which will slightly slow down the code speed.

4.3.1 Ensuring Memory Alignment by Array Padding

As unaligned references might cause performance penalty, I have also aligned vector loads/stores by implementing array padding techniques.

In this thesis, "memory alignment" means the memory addresses of `U_out[k][j][1]` and `U_in[k][j][1]` are aligned on 16- or 32-byte boundary for arbitrary `k`, `j` in range. Aligned memory can assist compiler to generate less data movement instructions, and increase unroll factors for innermost loops (see Appendix E for a detailed comparison between unaligned and aligned assembly codes).

In addition, once the arrays are aligned, intrinsic codes can use `VMOVNTPS`, which requires aligned stores, to move new `U_out` values back to memory by-passing the cache, which prevents other useful data being ejected from the cache, hence reduces cache misses.

I use an array padding method to obtain memory alignment. This method works by allocating extra data and then shifting the address of the effective data memory to an aligned boundary. Take aligning `U_in[k][j][1]` on 32-byte boundary as an example, this method is implemented in two phases:

1. Align the first `U_in[k][j][1]`, i.e., `U_in[-r+1][-r+1][1]`, which is the `r`-th element in the old unaligned array memory.

   (a) Allocate (32/sizeof(float)-1) more floating points of memory for `U_in`

   (b) Find the offset between the `r`-th element and the position of the next nearest element whose address is aligned on 32-byte boundary

   (c) Shift the array base pointer until `U_in[-r+1][-r+1][1]` is aligned to this element.

2. Pass the alignment property to the remaining `U_in[k][j][1]`s.

   For simplicity, in my experiments, `NX` is set to be a multiple of 16, so if `U_in[-r+1][-r+1][1]` is aligned, then `U_in[-r+1][-r+1][NX]` will be aligned automatically because of contiguous data storage. But to pass alignment property from `U_in[-r+1][-r+1][NX]` to `U_in[-r+1][-r+2][1]`, the number of

floating points padded in between should be a multiple of 8. So besides boundary and ghost elements, more padding may be needed to satisfy this requirement. Figure 4.3 shows an example of padding extra 4 elements between every `U_in[k][j][NX]` and `U_in[k][j+1][1]` to pass alignment property for this SC of order 2. In this case, we pad four extra cells encompassed by the dashed frame to pass alignment property from `U_in[k][j][NX]` to `U_in[k][j+1][1]`.

If `NX` is not a multiple of 16, then some array elements near the end of the i-loop have to be processed in non-vectorized way. But after properly adjust the pad size between two adjacent segment, it is still easy to get `U_in[k][j][1]`s aligned.

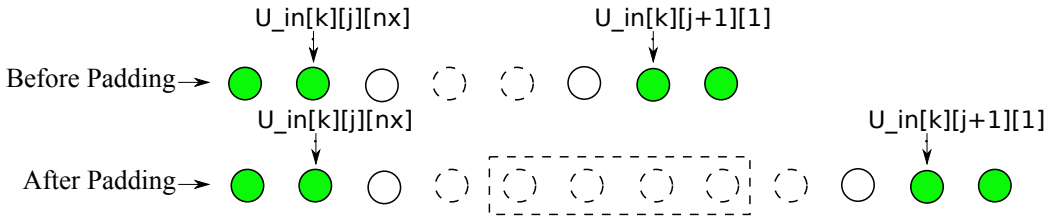


Figure 4.3: Use array padding to align 4th order SC on 32-byte boundary. Computation domain includes colored and solid white circles (boundary), dashed circles are either ghost or padding cells.

Chapter 5

Cache Optimization Methods

This chapter first uses the roofline model to verify that 4-th and 16-th order stencil codes (SCs) are memory bandwidth bound on Xeon E5-2660 and analyses the source of memory traffic, then it describes three cache optimization methods that could reduce the memory traffic and hence break the bottleneck: thread-blocking, separate-and-divide, and parallelized time-skewing. Parallelized time-skewing involves temporal blocking and combines the features of conventional time-skewing and the thread-blocking scheme.

5.1 Memory Bottleneck

The roofline model [Williams et al., 2009] helps to differentiate performance bottleneck of a kernel on a given platform and predict the performance upper bound of a kernel on that platform.

Before presenting the roofline model based on Xeon E5-2660, I will give definitions for some terms which are essential elements of a roofline model.

1. *Machine peak*: the product of the number of cores and the peak GFLOPs/sec generated by each core. It reflects the computing power of a platform. For Xeon

E5-2660, where AVX addition and multiplication pipelines are independent, its *machine peak GFLOPs/sec* is:

$$8\,(\#cores) \times [8\,(add) + 8\,(mul)]\,FLOPs/cycle \times 2.2\,GHz = 281.6\,GFLOPs/sec.$$

2. *Algorithmic peak*: the theoretical peak GFLOPs/sec obtained by a kernel on a given platform. Since for wave equation based stencil kernel of any order, its ratio between the number of additions and the number of multiplications is approximately equal to 2, its *algorithmic peak* is no greater than

$$281.6\,GFLOPs/sec \times 3/4 = 211.2\,GFLOPs/sec$$

3. *Machine balance* [McCalpin, 1995]: the ratio between *machine peak GFLOPs/sec* and sustained DRAM to cache memory bandwidth. It predicts the maximum FLOPs executed by the platform per byte movement. For Xeon E5-2660, I use *STREAM Triad* benchmark [McCalpin, 2013] (setup: array of SPFP type, having total memory of 152.6MB, which is greater than 4 times of 20MB L3 cache to ensure the observed memory bandwidth is between DRAM and cache) to measure its sustained DRAM to cache memory bandwidth and its observed value is 32447.4 MB/sec. So the *machine balance* of our test device is:

$$281.6\,GFLOPs/sec\,/\,32447.4\,MB/sec = 8.68\,FLOPs/byte.$$

4. *Operation intensity* [Williams et al., 2009]: the ratio between the number of FLOPs executed per byte transferred between DRAM and cache for a given kernel. For $2r$-th order stencil kernel based on wave equation, if the problem size exceeds the L3 cache size, which is usually the case, then at each time step

each array will be loaded into the cache at least once and one array will be stored back to DRAM once. As a result, there will be at least two loads and one store per grid point. On the other hand, computing each point requires a total of $(9r + 2)$ FLOPs. So the upper bound of the *operation intensity* of 4-th order stencil codes is:

$$(9 \times 2 + 2)\, FLOPs/(3 \times 4)\, bytes = 1.67\, FLOPs/byte.$$

Similarly, for 16-th order stencil, its *operation intensity* upper bound is:

$$(9 \times 8 + 2)\, FLOPs/(3 \times 4)\, bytes = 6.17\, FLOPs/byte.$$

Figure 5.1 shows the roofline model based on our test device, Xeon E5-2660. The X axis represents the *operation intensity* of an arbitrary kernel, the Y value of the corresponding point on the roofline indicates its performance upper bound in terms of GFLOPs/sec.

Note that the slope of the skewed line is equal to the platform's sustained memory bandwidth. When the *operation intensity* of a kernel is less than the *machine balance*, the performance of the kernel is bounded by the limited memory bandwidth, and it cannot attain the *machine peak*. When the *operation intensity* is greater than the *machine balance*, the performance of the kernel is then compute bound.

For wave equation based stencils, since their unbalanced mix of addition and multiplication prevents them from achieving *machine peak*, the dividing point between memory bandwidth bound and compute bound further shifts to the left, 6.51 GFLOPs/sec. However, *operation intensities* of both 4-th order and 16-th order stencil are still less than this value, so both kernels are memory bandwidth bound. Based on the sustained memory bandwidth, their performance upper bounds are 54.18 GFLOP-

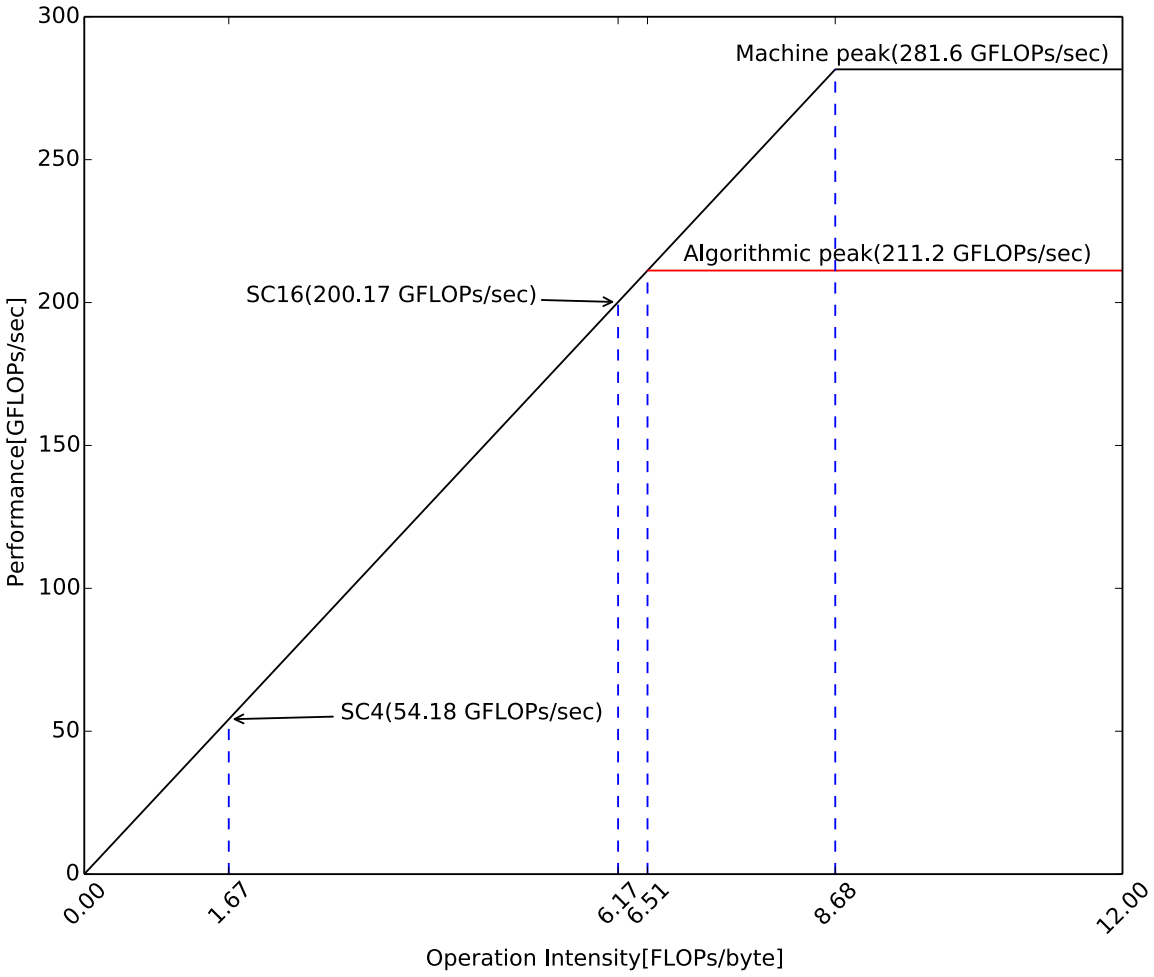


Figure 5.1: Roofline model based on Xeon E5-2660.

s/sec and 200.17 GFLOPs/sec respectively, which are also shown in Figure 5.1 with the first two dashed blue lines to the left.

5.2 Source of Memory Traffic

In the roofline model, the performance upper bound is found by referring to the upper bound of the *operation intensity* of a kernel, which assumes both arrays will be loaded into cache only once, and `U_out` will be stored back to DRAM only once. When this assumption is true, stencil codes of both orders are already memory bandwidth bound. If cache capacity misses and cache associative misses exist, then `U_in` array will be loaded into cache more than once per time step, which leads to more bytes moved per FLOP executed, then the *operation intensity* would further shift to the left, which results in a lower performance upper bound.

Conventional OpenMP parallelization on SC with a multi-fold loop structure usually parallelizes the outermost loop (`k`-loop) unless the OpenMP pragma is specified with `schedule(static,1)`. As demonstrated in Figure 5.2, each thread automatically gets a bunch of contiguous `NZ/NT` `k`-planes (e.g., `U_in[k][1:NY][1:NX]` is a single `k` plane) to process along the increasing `k` direction, where `NZ` is the range of the `k`-loop and `NT` is the number of OpenMP threads (note: not the number of time steps!)

When thread `#p` updates `U_out` at (i_p, j_p, k_p), it will need `U_in` at $(i_p, j_p, k_p + 1)$. Then thread `#p` will execute (`NY-r`)×`NX-1` stencils before updating `U_out` at $(i_p, j_p - r, k_p + 1)$ when it reuses `U_in` at $(i_p, j_p, k_p + 1)$ again. Note that none of these stencil operations in between involves `U_in` at $(i_p, j_p, k_p + 1)$. Denote the data memory size involved in these operations as $mem(SC)$.

Suppose at a certain time thread `#p` is updating `U_out` at (i_p, j_p, k_p), `p` = 0, ..., `NT`-1. If `NZ/NT` is large enough that any two threads will need non-overlapped

regions of `U_in` planes, and if the cache capacity is less than `NT`×$mem(SC)$, then `U_in` at $(i_p, j_p, k_p + 1)$ must have been out of the cache before the thread `#p` arrives at $(i_p, j_p - r, k_p + 1)$, `p` = 0, ..., `NT`-1, so `U_in` at $(i_p, j_p, k_p + 1)$ has to be reloaded from DRAM again.

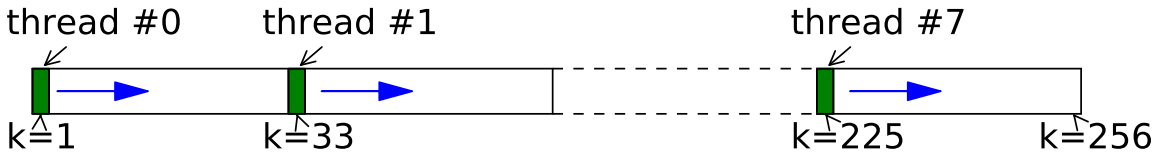


Figure 5.2: Snapshot of conventional OpenMP thread access. `k` ranges from 1 to 256. OpenMP forks 8 threads in total. The picture only shows `k` dimension, but hides `i,j` dimensions.

A stencil has `2r+1` points along `k` dimension, which means almost every `U_in` data will be used as a finite difference term along `k` dimension `2r+1` times, so it has to be loaded `2r+1` times from DRAM every time step. Table 5.1 summarizes the number of array loads and stores associated with the stencil kernel every time step in this case.

#loads	#stores
(2r+1)×`U_in` + 1×`U_out`	1×`U_out`

Table 5.1: Loads and stores per time step if using conventional OpenMP parallelization.

For our test problem, the computation domain is of 256^3 excluding boundary layer of thickness 1. Since stencils of different orders have different thickness of ghost layers, the cache can hold at most 67 or 73 `k`-planes of SPFP type for SC16 or SC4 respectively, which is equivalent to 8, 9 `k`-planes per CPU core. Note that updating `U_out[k][1:NY][1:NX]` would require `2r+1` `U_in` `k`-planes, and one `U_out` `k`-plane. So *mem(SC)* size is amount to at least ((`NY-r`)×`NX-1`)/(`NY`×`NX`)×(`2r+2`) planes. When `r` is 4, *mem(SC)* is at least 6 planes, less than the quota of 9 planes assigned to each core; However when `r` is 8, *mem(SC)* is at least 17 planes, greater than the quota of

8 planes assigned to each core, so the entire `U_in` array will be loaded into cache at least 17 times every time step. Then the resulting *operation intensity* becomes

$$(9 \times 8 + 2)\, FLOPs/((17 + 2) \times 4)\, bytes = 0.97\, FLOPs/byte.$$

Multiply it by the sustained memory bandwidth to get the performance upper bound: 31.59 GFLOPs/sec. As the `U_in` data at points with `k` index near 0 or near `NZ` will be accessed less than 17 times every time step, the actual GFLOPs/sec in this case should be slightly higher than the above value.

Therefore to lift the performance upper bound of the stencil codes, one has to first avoid multiple acesses of `U_in` per time step to prevent the *operation intensity* from shifting to the left. Then work on maximizing the time steps leaped over per array access, i.e., reference the array only once over several time steps, so to reduce bytes moved per FLOP executed, and hence shifting the *operation intensity* to the right. The roofline model reveals that this second step is indispensable if one wants to transform the problem to be compute bound.

So in the following sections, I will first present two methods that aim to avoid multiple accesses to DRAM for `U_in`, and then present a third method that blocks the temporal dimension so that in average the number of array accesses per time step will be less than once.

5.3 Thread-blocking Method

Note that in the conventional OpenMP parallelization scheme, because of the discrete thread accessing pattern, the total memory footprint associated with the intermediate stencil operations of `NT` threads is `NT` times $mem(SC)$. If the access locations of these threads are closer to each other, then some `U_in` planes can be re-used by multiple

threads before they get ejected. Consequently, the total memory footprint of those stencils should be smaller, so the cache may keep `U_in` data at (i_p, j_p, k_p) in cache when thread `#p` arrives at $(i_p, j_p - r, k_p + 1)$.

Based on the above idea, I propose the method of blocking the OpenMP threads while they are performing stencil operations. This method can be easily implemented by adding the `schedule(static,1)` directive in OpenMP pragmas outside k-loops. With this directive, threads will update `NT` contiguous k-planes along the increasing `k` direction every round and in each round, each thread will deal with only one `k`-plane. It ends up that all the threads are updating `NT` contiguous `k`-planes simultaneously. Figure 5.3 visualizes this process.

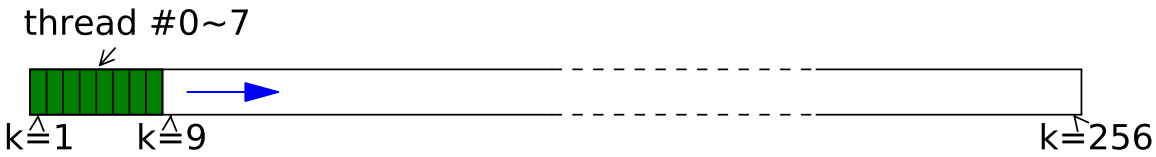


Figure 5.3: OpenMP thread access pattern when adding `schedule(static,1)`. `k` ranges from 1 to 256. OpenMP forks 8 threads in total. The eight green planes are the k planes currently being updated by these 8 threads. The blue arrow indicates the moving direction.

Because each thread has the same work load and all the cores are the same, during a certain period, the threads might be updating `U_out` at (i, j, k), ..., $(i, j, k + \texttt{NT} - 1)$ simultaneously, or at least points very close to these locations. So `U_in[k+1][j][i]` loaded for updating `U_out[k][j][i]` by thread `#0` might be used concurrently or very soon to update `U_out[k+1][j][i]` by thread `#1`, or to update `U_out[k+2][j][i]` by thread `#2`, etc. If there is no data reload along `i` or `j` direction in the current time step, then every time step each `U_in` data will be reused $min(\texttt{NT}, \texttt{r+1})$ times.

To guarantee there is no data reload along `i` or `j` direction, the cache has to keep at least the amount of data associated with updating two consecutive segments. The left picture in Fig 5.4 shows the memory footprint involved in updating two

consecutive segments (`U_out[k][j:j+1][1:NX]`). The left figure shows the memory footprint involved in updating `U_out[k][j:j+1][1:NX]`. In the right picture, the thread is updating `U_out[k][j+2][1:NX]` (the white chunk inside the red dashed line), so it loads the `U_out` and `U_in` data inside the red dashed region by releasing the space occupied by the memory inside the black dashed line.

When the thread moves downward to update the next segment (`U_out[k][j+2][1:NX]`), the data part inside the dashed red region (`U_out[k][j+2][1:NX]` and `U_in[k][j+2+r][1:NX]`, `U_in[k][j+2][1-r:0]`, `U_in[k][j+2][NX+1:NX+r]`) will be loaded into cache. Meanwhile, the cache will throw out least recently used data to make room for this data part. Among the data involved in updating the previous two segments, only the data part inside the dashed black region (`U_out[k][j][1:NX]` and `U_in[k][j-r][1:NX]`, `U_in[k][j][1-r:0]`, `U_in[k][j][NX+1:NX+r]`) might be thrown out of the cache since they have not been used in updating the second segment (`U_out[k][j+1][1:NY]`). Luckily, the amount of data inside the red region is the same as the one inside the black region, so without throwing out any possible data involved in updating the next segment, the cache can release the room previously occupied by the data in the black region and assign it to the data inside the red region. Once the data inside the black region are thrown out of the cache, they will never be reused in updating the current `k`-plane, which indicates that there will be no reloads along `i` or `j` direction. Table 5.2 summarizes the number of array loads and stores, as well as the cache size condition when there is no data reload along `i` or `j` direction.

#loads	#stores	lb(cache)
(2r-min(NT, $r+1$))×U_in + 1×U_out	1×U_out	((4+2r)NX+4r)NT

Table 5.2: Loads and stores per time step with thread-blocking. lb(cache) denotes the minimum cache capacity in order to attain these number of loads and stores, its unit is "SPFP(4 bytes)".

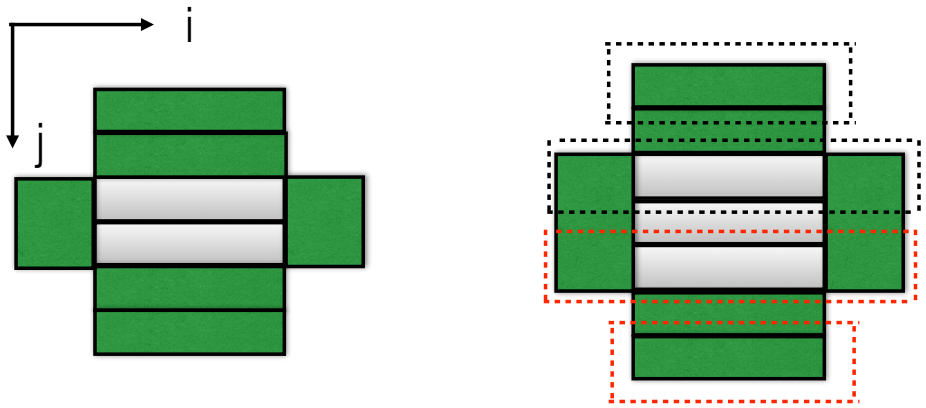


Figure 5.4: These two figures are based on SC4. `U_out` data are represented by white chunks and `U_in` data are represented by green chunks.

Furthermore, if the cache can hold the memory footprint associated with two consecutive rounds of stencil updates, i.e., 2NT `U_out` k-planes and (2NT + 2r) `U_in` k-planes, then there will be no data reload along `k` direction in the current time step. When all the threads proceed to update the next round, only part of the data related to the first round might be get ejected, making room for loading new data for the next round. Figure 5.5 visualizes this process. Among all the data related to the first two rounds, the data encompassed in the black dashed region (`U_out[k:k+NT-1][1:NY][1:NX]` and `U_in[k-r:k-r+NT-1][1:NY][1:NX]`) are ones most likely being ejected from cache since they have not been used for updating the second round, and hence they are less recently touched data. To update the third round, the threads only need to load `U_out[k+2NT:k+3NT-1][1:NY][1:NX]` and `U_in[k+2NT+r:k+3NT-1+r][1:NY][1:NX]` into cache, and the size of which is equal to the size of the data memory in the blacked dashed region. It means that without sacrificing any data memory related to the second round, the cached data can be replaced by the data related to the later two rounds of update. The reason to avoid any possible ejection of data related to the second round is that since they are updating the second round simultaneously because of OpenMP parallelization, if some of them were to be ejected, it might be the ones, e.g., `U_in[k+2NT-1][1][1]`,

that will be used for updating the third round. Ejecting these planes will result in extra data reloads.

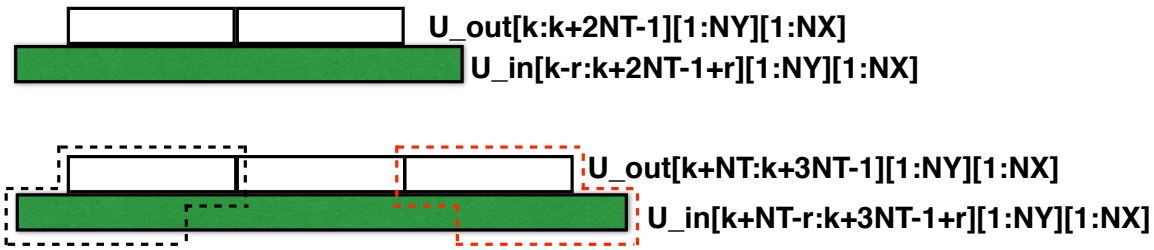


Figure 5.5: Illustration of thread-blocking when cache capacity meets certain requirement. The upper picture shows the memory footprint associated with two consecutive rounds of update, and they are kept in cache concurrently. A white chunk stores `NT` `U_out` `k`-planes, and the green chunk stores `U_in` `k`-planes. The lower picture shows that the threads are loading the data for next round (encompassed in the red dashed region) into cache and gradually releasing the space previously occupied by the data belonging to the first round (encompassed in the black dashed region).

Note that the memory encompassed in the black dashed region will be no longer used for later updates in the current time step. Apply the above analysis when the threads proceed to update `U_out` planes along increasing `k` direction, the planes being ejected along the way will be no longer be used for the corresponding later updates in the current time step, which means that all the data have been fully reused before they are out of the cache. Table 5.3 shows the number of array loads and stores per time step in this optimal case.

#loads	#stores	*lb*(cache)
1×`U_in` + 1×`U_out`	1×`U_out`	(4`NT` + 2`r`) `k`-planes

Table 5.3: Optimal loads and stores per time step of thread-blocking method. *lb*(cache) denotes the minimum cache capacity in order to attain these number of loads and stores, its unit is "SPFP(4 bytes)".

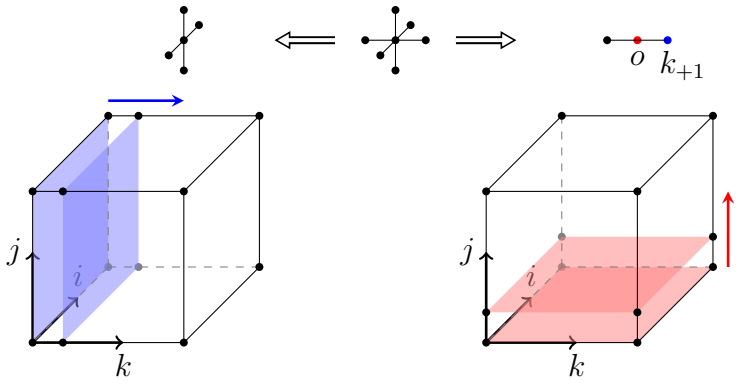


Figure 5.6: Separate-and-exchange method. The stencil is partitioned into two parts. The first part accumulates the finite difference terms along `i`, `j` dimensions and loops over the computation domain according to the original pattern, while the second part accumulates the finite difference terms along `k` dimension and loops over the computation domain according to the pattern illustrated in the right cubic.

5.4 Separate-and-interchange Method

Dr. Stork (Stork C. personal communication, 2013) demonstrates a novel approach to reduce cache misses by changing the loop traversal pattern. It partitions the accumulations of finite difference terms along three dimensions into three different loop structures. For the accumulation along `k`-dimension, it exchanges the `k`-loop and `j`-loop, making the stencil operator traverse along `j` direction last. Extracting this idea from his code and applying it to my symmetric stencil, I separate the accumulation of finite difference terms along `k` dimension, and then do loop interchanging for the two outermost loops. Figure 5.6 visualizes the procedure and Listing 5.1 demonstrates the necessary code transformation based on AUTOVEC kernel.

```
1 //Time loop{
#pragma omp parallel for private(j,i,ixyz)
3 for(k=1; k<=nz; k++)//k-loop
  for(j=1; j<=ny; j++){//j-loop
5   for(i=1; i<=nx; i++)//i-loop-1
      U_out[k][j][i]=-U_out[k][j][i]+c0*U_in[k][j][i];
7   for(ixyz=1; ixyz<=r/2; ixyz++){//ixyz-loop
      #pragma ivdep
```

```
9       for(i=1; i <= nx; i++)//i-loop-2
            U_out[k][j][i]+=cx[ixyz]*(U_in[k][j][ixyz+i]+U_in[k][j][-ixyz+i])
11                      +cy[ixyz]*(U_in[k][ixyz+j][i]+U_in[k][-ixyz+j][i])}}

13 #pragma omp parallel for private(k,i,ixyz)
   for(j=1; j<=ny; j++)//j-loop -> interchanges with k-loop.
15   for(k=1; k<=nz; k++)//k-loop
       for(ixyz=1; ixyz<=r/2; ixyz++)//ixyz-loop
17       for(i=1; i<=nx; i++)//i-loop
           U_out[k][j][i]+=cz[ixyz]*(U_in[ixyz+k][j][i]+U_in[-ixyz+k][j][i]);
19         // + process boundary data}
```

Listing 5.1: Stencil kernel implemented with separate-and-interchange method

For the first partial update, since there is no computation along `k` axis, as long as the cache capacity can ensure no data reloads along `i`, `j` dimensions, then the data will be fully used before being ejected. Using similar analysis as in Section 5.3, the lower bound on the cache capacity is approximately the size of the memory footprint associated with updating `2NX` contiguous stencils per thread, i.e., $\texttt{2NX} \times \texttt{NT}$ `U_out` array data and $((2 + 2\texttt{r}) \times \texttt{NX} + 4\texttt{r}) \times \texttt{NT}$ `U_in` array data.

For the second partial update, if the cache can ensure no data reloads along `k` dimension, then the data will be fully reused in cache. Similarly, the lower bound on the cache capacity for this condition is approximately the memory footprint associated with updating 2 contiguous stencils per thread, i.e., `2NT` `U_out` array data and $(2 + 2\texttt{r})\texttt{NT}$ `U_in` array data.

Therefore, if the cache can hold $((4+2\texttt{r}) \times \texttt{NX} + 4\texttt{r}) \times \texttt{NT}$ SPFP data simultaneously, then the data in two partial updates can be fully reused before being ejected. Table 5.4 summarizes the number of array loads and stores per time step given that the cache size is above a certain value.

The drawback of this method is that it needs another traversal over the computation domain due to two partial updates. For low-order stencil kernels, e.g., NAIVE

#loads	#stores	lb(cache)
2×U_in + 2×U_out	2×U_out	$((4 + 2\mathtt{r}) \times \mathtt{NX} + 4\mathtt{r}) \times \mathtt{NT}$

Table 5.4: Optimal Loads and stores per time step with separate-and-exchange method. lb(cache) denotes the minimum cache capacity in order to attain these number of loads and stores, its unit is "SPFP(4 bytes)".

kernel of second order, updating 2 contiguous U_out k-planes only requires 8 k-planes (2×U_out +(2 + 4)×U_in) per thread, and previous analysis shows that the quota for each core is 9 k-planes, so every time step the data can be fully reused, i.e., the number of loads plus stores is only 3 times per time step. Implementing this loop interchanging method can hardly benefit from cache reuse but double the number of loads and stores, which produces more memory traffic and slower code.

5.5 Parallelized Time-skewing Method

The importance of time-skewing method lies in temporal blocking, which reuses data of several time steps before ejecting them, thus reducing bytes moved between DRAM and cache per FLOP executed, i.e., increasing *operation intensity* of the kernel, and leading to higher GFLOPs/sec.

Inspired by the thread-blocking method that blocked thread accesses can generate less cache misses than discrete thread accesses, I set the *tiling width* (spatial blocking factor) in the time-skewing scheme to be NT, so that at each time step, threads will update a contiguous chunk of k-planes and every thread gets exactly one k-plane. Figure 5.7 demonstrates this idea. Each round, NT threads will first update NT U_out k-planes in time level $(t + 1)$, then update NT U_in k-planes in time level $(t + 2)$, and then update NT U_out k-planes in time level $(t + 2)$, etc. Besides, to avoid extra memory allocation for the temporal values at intermediate time steps, the offset between the starting k indices of chunks belonging to consecutive time steps in the

same round is set to be $max(\texttt{NT}, \texttt{r})$. For example, in Figure 5.7, the offset between any adjacent green chunks is 8 since both SC4 and SC16 have $\texttt{r} \leq 8$.

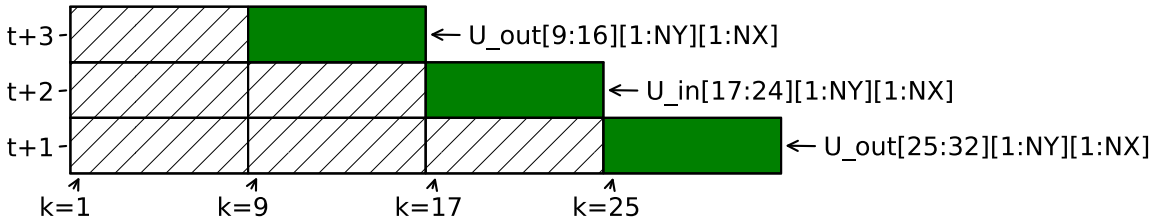


Figure 5.7: Parallelized time-skewing method. Here assume OpenMP forks 8 threads, with 1 thread per core. One data chunk is composed of 8 `k`-planes and processed by 8 OpenMP threads. The hatched chunks are ones already processed in previous rounds. The green chunks, with labels on the right denoting data sources, are ones being updated in the current round. Here one round covers three time steps: t+1, t+2, t+3.

The optimal *tiling height* (temporal blocking factor, denoted as `NTS`) is determined by minimizing data reloads. The ideal scenario is to load and store both arrays only once every `NTS` time steps. To achieve this goal, using similar arguments as those in the above section, the cache has to hold at least all the planes needed for two consecutive rounds.

When `NTS` is odd, as shown in the upper picture in Figure 5.8, this memory involves $[(\texttt{NTS}-1)\times max(\texttt{NT},\texttt{r})+2\texttt{NT}]$ `U_out` `k`-planes, and $[(\texttt{NTS}-1)\times max(\texttt{NT},\texttt{r})+2\texttt{NT}+2\texttt{r}]$ `U_in` `k`-planes; when `NTS` is even, as shown in the lower picture in Figure 5.8, this memory involves $[(\texttt{NTS}-1)\times max(\texttt{NT},\texttt{r})+2\texttt{NT}+\texttt{r}]$ `U_out` `k`-planes, and $[(\texttt{NTS}-1)\times max(\texttt{NT},\texttt{r})+2\texttt{NT}+\texttt{r}]$ `U_in` `k`-planes. Therefore, in both cases, the total memory is of size $[2(\texttt{NTS}-1)\times max(\texttt{NT},\texttt{r})+4\texttt{NT}+2\texttt{r}]$ `k`-planes. Table 5.5 summarizes the loads and stores per time step in the optimal case and the minimum cache capacity to hold this memory.

Although higher temporal blocking factor leads to greater cache miss reduction, the cache capacity imposes an upper limit on `NTS`. Denote $max(\texttt{NTS})$ as the maximum `NTS` value that makes the memory related to update two consecutive rounds fit into

L3 cache. If NTS is greater than max(NTS), then there will be multiple accesses to both arrays over NTS time steps.

On Xeon E5-2660, with L3 cache of 20 MB, parallelized by 8 OpenMP threads, in order to keep these planes in cache simultaneously, NTS of SC4 should be less than 2.98, and NTS of SC16 should be less than 2.22. Since there will be some extra room on chip due to L1/L2 cache, the theoretical optimal NTS for SC4, SC16 on the test machine is 3, 2 respectively.

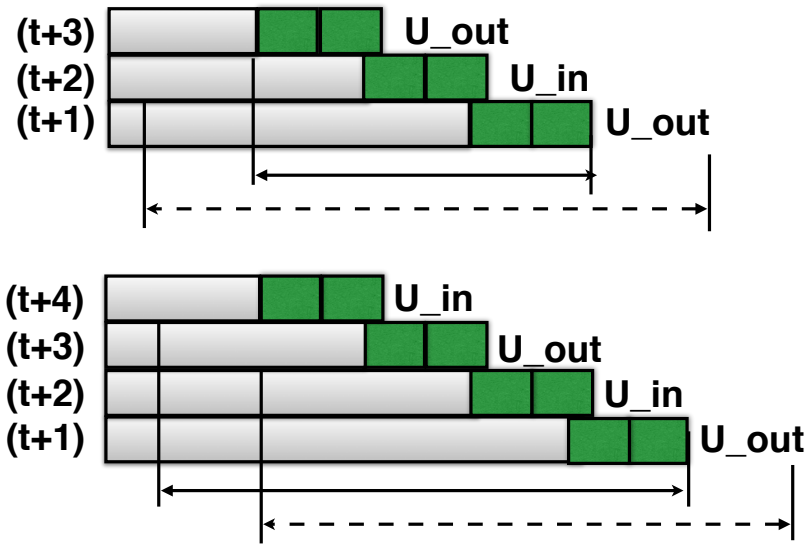


Figure 5.8: This figure is based on a more general case when NT is less than r. It shows the range of U_out and U_in k-planes associated with two consecutive rounds. The solid double-arrow lines mark the ranges of U_out k-planes, while the dashed double-arrow lines mark the ranges of U_in k-planes. When NT is greater than r, similar analysis would apply.

#loads	#stores
1/NTS×U_in + 1/NTS×U_out	1/NTS×U_in + 1/NTS×U_out
lb(cache)	
[2(NTS-1) × max(NT, r) + 4NT+2r] k-planes	

Table 5.5: Optimal Loads and stores per time step with parallelized time-skewing method. lb(cache) denotes the minimum cache capacity in order to attain these number of loads and stores, its unit is "SPFP(4 bytes)".

Chapter 6

Results and Discussion

In this section we will conduct a series of tests with various kernels to investigate the effectiveness of different optimization techniques.

6.1 Experiment Setup

Our test device is an Intel Xeon E5-2660 Sandy Bridge processor, containing 8 cores of frequency 2.2GHz. Each core has 32KB L1 cache, 256KB L2 cache, and a 20MB L3 cache is shared among 8 cores. DRAM associated with this processor is 32GB.

The computation domain size is 258^3 (including boundary layer of size one). The array data is of SPFP type. The total array memory will be 140MB$\sim$161MB, which is significantly larger than the shared L3 cache size but smaller than the socket DRAM, making the DRAM to cache memory latency and bandwidth be the major memory bottleneck.

The total number of time steps is 5001, which is chosen to restrict the total run time within 10 mins, meanwhile keeping the computation overhead negligible.

All the OpenMP threads are pinned to a specific core at the beginning of the program by setting the environment variables `KMP_AFFINITY=verbose,compact` (for

icc), and `GOMP_CPU_AFFINITY=0-7` (for gcc).

Each kernel is tested with both icc 14.0.0 and gcc 4.8.2. Compiler options passed into two compilers are:

```
icc -openmp -xAVX[-xSSE4.2] -O3 -funroll-loops
gcc -fopenmp -march=native -O3 -funroll-loops
```

All the event counts are measured by *Perf* (see *Perf* Setup at Appendix F). Every event count presented in the following charts is the mean value collected from three test runs.

6.2 Vectorization

Experiments in this section will present the vectorization results and performance improvement of stencil codes vectorized by both manual and automatic vectorization approaches. Besides, the effectiveness of basic vectorization, memory alignment, and explicit loop unrolling in accelerating code speed will be evaluated and compared.

For comparison purpose, I tested the SIMD instruction composition (Figure 6.1), L3 cache misses (Figure 6.2) and kernel run time (Figure 6.3) of the following kernels:

- **EXTEND [v, u/a] kernel** (Section 3.3): based on NAIVE kernel (Listing 3.1), with its coefficient loops fully unrolled, making i-loop be the innermost loop. It is also used in the old IWAVE acd package. [u] indicates unaligned memory and [a] indicates aligned memory. [v] means adding Intel pragma `ivdep` to remove compiler-assumed data dependencies (icc only).

- **SIMD kernels [u/a], including AVX# or SSE#** (Section 4.2): based on NAIVE kernel, coded with intrinsics. # is the unrolling factor. There is no GCC version of SSE# kernel because "-march=native" option makes GCC be aware of the machine can generate AVX, so GCC generates AVX only.

- **AUTOVEC kernel [u/a]** (Section 4.3): based on NAIVE kernel, with rearranged loop structure and `ivdep` pragma to remove data dependencies assumed by icc.

- **NAIVE kernel [u]** (Section 3.3): with unaligned memory.

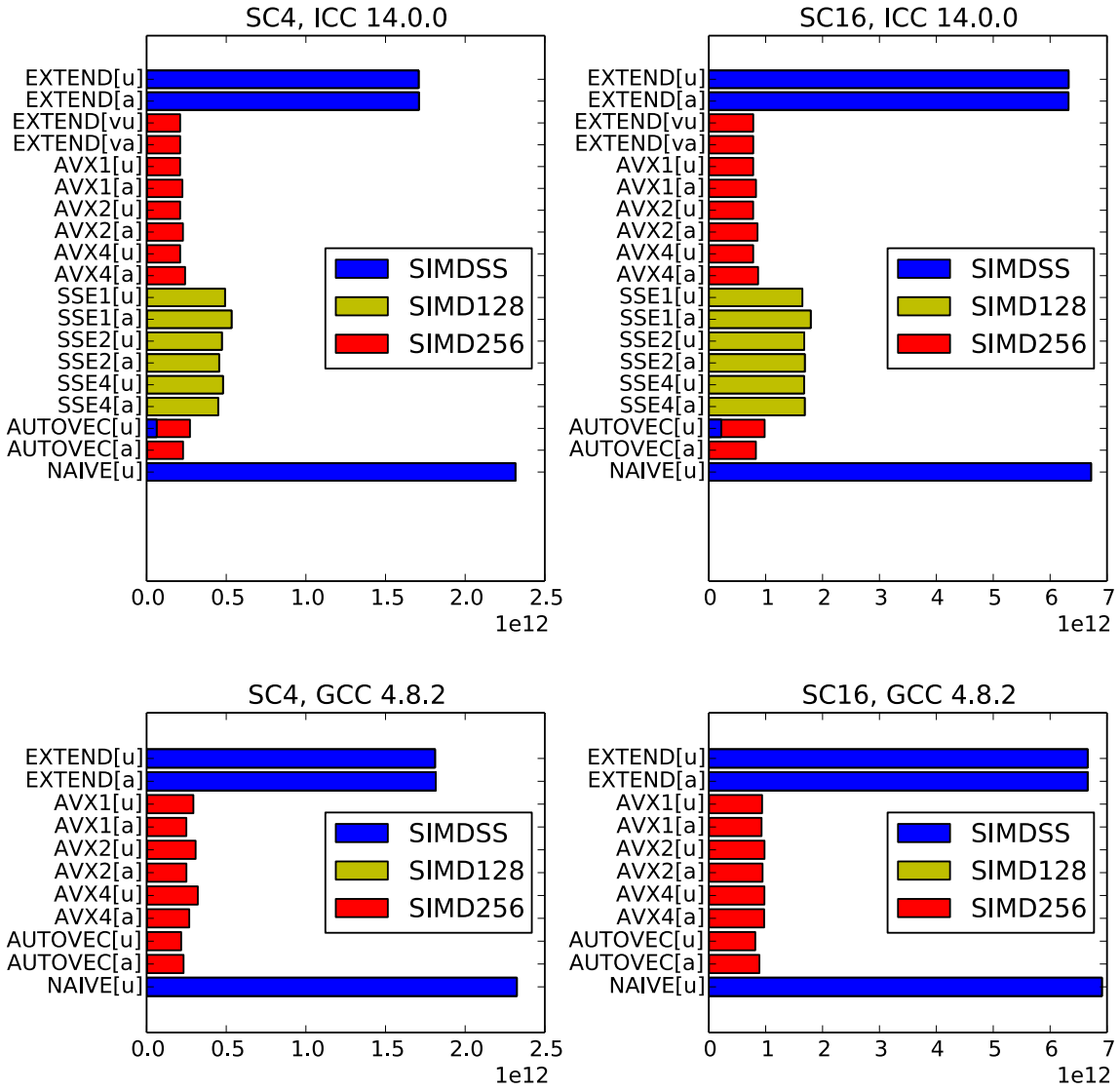


Figure 6.1: Stacked bar chart of SIMD instruction composition of each kernel. *max*(std-dev/mean) is less than 0.06%. SIMDSS: number of *scalar* SIMD instructions; SIMD128: number of *packed* SSE instructions; SIMD256: number of *packed* AVX instructions.

SIMD instruction composition results (Figure 6.1) reveal that:

1. No matter whether the array memory is aligned or not, EXTEND without ivdep cannot be vectorized by either compiler.

2. SIMD kernels of various unrolling factors, EXTEND with ivdep, and aligned AUTOVEC kernel are all fully vectorized by both compilers.

3. Unaligned AUTOVEC kernel will generate a portion of scalar instructions when compiled by icc, but not by gcc.

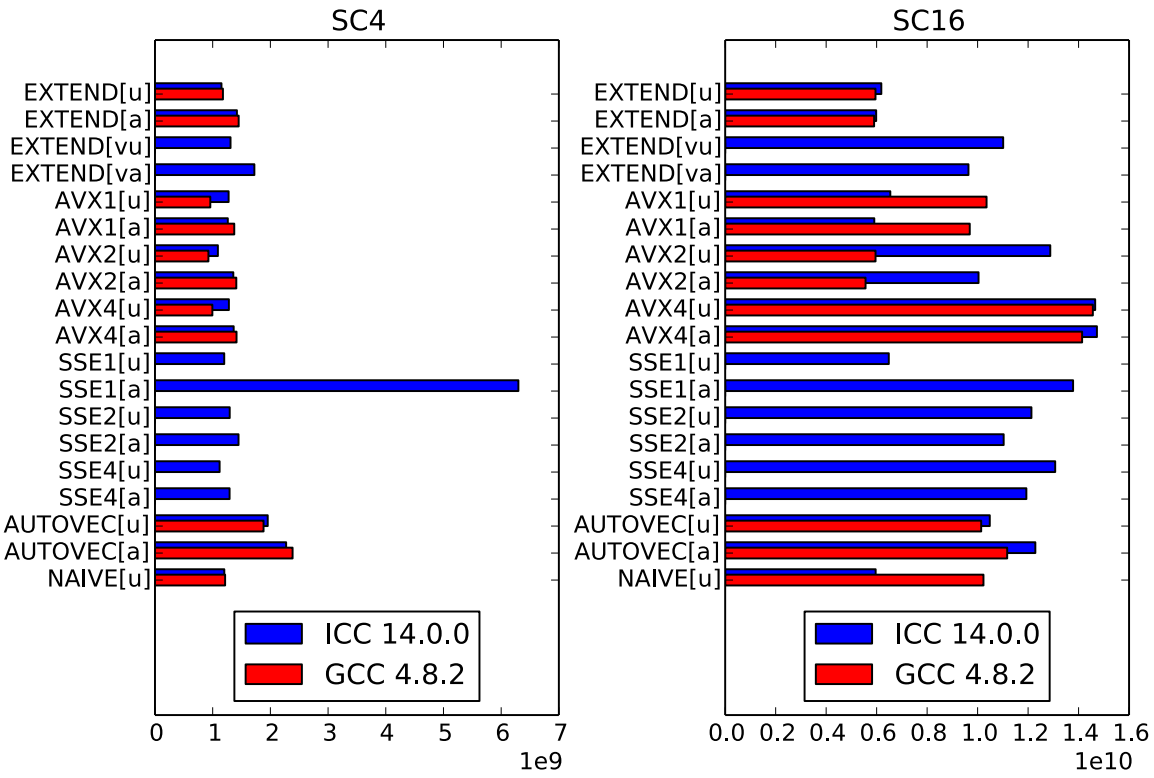


Figure 6.2: L3 cache misses of each kernel. *max*(std-dev/mean) is less than 1.39%.

In L3 cache miss results (Figure 6.2), I have discovered a pattern:

For low-order SC, the number of cache misses for the aligned version is greater than that of its unaligned version; for high-order SC, except SSE1 and AUTOVEC kernels, the number of cache misses for the aligned version is smaller than that of its unaligned version.

The problem may result from the array padding implementation. For low-order stencils, to pass alignment property from `U_in[k][j][NX]` to `U_in[k][j+1][1]`, useless memory is padded in between. When the hardware prefetching engine arrives at the virtual layer, as all the data elements before it are used by CPU, the engine will assume this piece of useless memory is also useful and load it. As a result, memory that will be used later may get ejected from the cache, hence increasing cache misses. For high-order stencils, the virtual layers are thick enough that there is no memory padded in between but only front padding. It means that memory alignment is more efficient for high-order SC.

There are two observations I can't explain theoretically: one is that SSE1[a] produces far more cache misses compared with SSE1[u], the other is that for high-order SC, AUTOVEC[a] produces more cache misses than AUTOVEC[u].

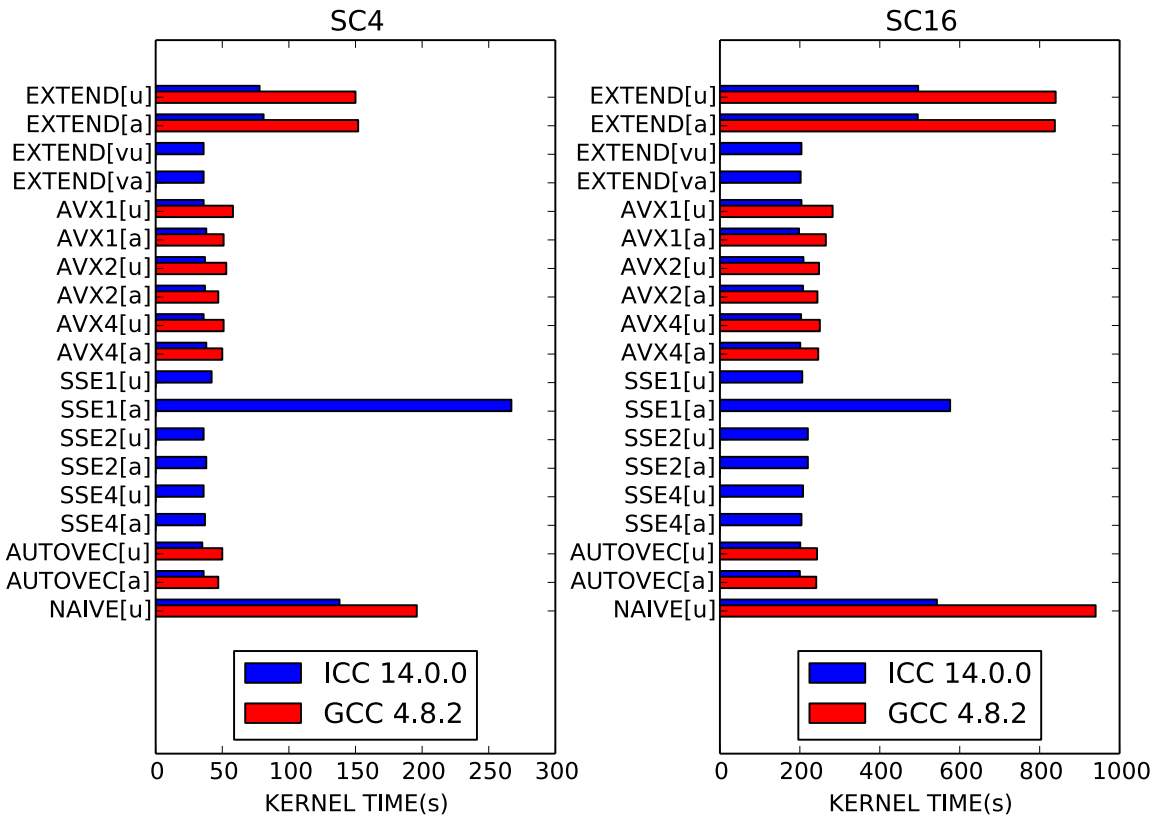


Figure 6.3: Run time of each kernel. Maximum deviation is less than 1 sec.

Kernel run time results (Figure 6.3) show that:

1. EXTEND with assumed dependency removed, SIMD kernels (except aligned SSE1) of various unrolling factors, and AUTOVEC[u/a] attain similar run time results. It means that loop unrolling, and memory alignment barely affect the final results, even if they do affect vectorization and cache misses. More importantly, now it seems unnecessary to hand-vectorize codes using intrinsics or align the memory, since AUTOVEC[u] can produce codes with comparable performance as tuned intrinsic codes.

2. When using SSE with aligned memory, the kernel needs to be explicitly unrolled at least twice in order to attain similar run time as other fully vectorized kernels.

3. When compiling the same kernel, gcc always produces slower code than icc. But for intrinsic codes and AUTOVEC kernel, the run time difference due to two different compilers is not much.

6.3 Cache Optimization

Since one L3 cache miss will result in a memory access, the number of L3 cache misses can be an indicator of the memory traffic. The following benchmark results reveal the effectiveness of each optimization method in reducing memory traffic, and how the reduction in memory traffic would affect run time results. Besides aligned AUTOVEC as the reference, other kernels are:

- **AUTOVECz**: aligned AUTOVEC implemented with separate-and-divide method.
- **AUTOVEComp**: aligned AUTOVEC implemented with thread-blocking method.
- **AUTOVECts**: aligned AUTOVEC implemented with parallelized time-skewing method. `NTS` is chosen as 3 for SC4, and 2 for SC16.

Table 6.1 is the L3 cache miss result of each kernel. Results from both compilers follow the same pattern: as expected, AUTOVECz only reduces cache misses for high-order stencil, but incurs more cache misses for low-order stencil; AUTOVEComp slightly reduces cache misses for low-order stencil, and for high-order stencil, it reduces the cache misses by 55%∼62%; For both low- and high-order cases, the cache misses of AUTOVECts is about one third of it of AUTOVEComp kernel.

	SC4		SC16	
Kernel	[I]Miss	[G]Miss	[I]Miss	[G]Miss
AUTOVEC	2.27e9	2.38e9	1.23e10	1.12e10
AUTOVECz	4.95e9	5.32e9	8.09e9	5.95e9
AUTOVEComp	2.05e9	2.13e9	2.73e9	2.91e9
AUTOVECts	7.17e8	7.36e8	1.07e9	1.15e9

Table 6.1: L3 cache misses of different kernels. Miss is the number of L3 cache line misses measured by Perf. [I] represents the kernel is compiled by icc 14.0.0, and [G] represents the kernel is compiled by gcc 4.8.2.

Figure 6.4 verifies my analysis for optimal temporal blocking factor in Section 5.5. For SC4, the cache misses and run time will approach their own minimums when `NTS` goes to 3, while for SC16, the optimal blocking factor is 2. Note that AUTOVEComp has higher cache misses than AUTOVECts when `NTS` is 1 because for AUTOVECts, it updates ghost boundary layers once it finishes updating the values near the boundary, so the mirror values may still reside in cache, and hence it has lower cache misses. Again, even if this factor contributes to cache miss reduction, the run time doesn't improve as much.

Figure 6.2 shows the run time of each kernel. Run time results of AUTOVECz and AUTOVEComp almost match their cache miss results: more reduction in cache misses produces shorter run time. However the run time improvement by implementing parallelized time-skewing method is not obvious, even though it significantly reduces the cache misses compared to AUTOVEComp.

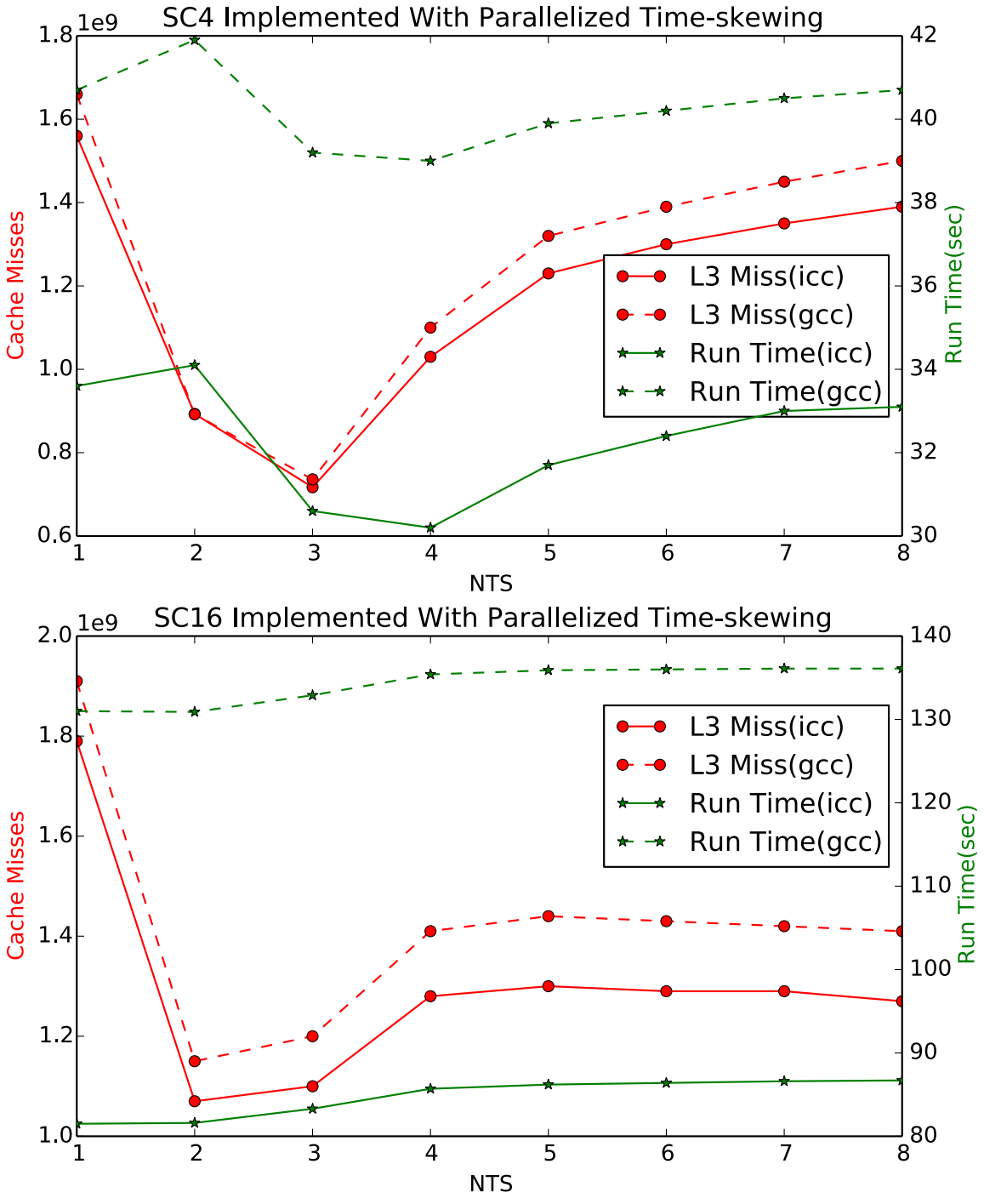


Figure 6.4: Cache misses and run time of the stencil kernels when implemented with parallelized time-skewing methods with various `NTS`.

	SC4		SC16	
Kernel	[I]Time(s)	[G]Time(s)	[I]Time(s)	[G]Time(s)
AUTOVEC	36	47	200	241
AUTOVECz	65	72	97	156
AUTOVEComp	35	41	84	133
AUTOVECts	31	39	82	131

Table 6.2: Run time results of different kernels. Notations are the same as the ones used in Table 6.1.

6.4 Summary

Figure 6.5 evaluates the effectiveness of vectorization, thread-blocking, parallelized time-skewing methods in terms of improving the SC's GFLOPs/sec and Mstencil/sec (Million stencils per second).

Note that without any cache optimization methods, the GFLOPs/sec of SC4 is about 50.72 (icc), which matches the roofline model prediction in Section 5.1, and it indicates that for SC4, the number of array loads and stores are indeed 3 times per time step, so only cache optimization methods involving temporal blocking can further increase its performance. The GFLOPs/sec of SC16 without any cache optimization methods is about 33.14 (icc), and this value matches my theoretical prediction in Section 5.2 based on the condition that `U_in` array is loaded 17 times per time step.

For low-order SC, vectorization alone can give near-peak performance, the combined cache optimization methods only provide 1.2x speed up for fully-vectorized codes; for high-order SC, both vectorization and thread-blocking method are indispensable, and they can provide about equal speed up for SC.

Time-skewing is implemented with the greatest effort, yet except for providing moderate performance gain for low-order SC, the contribution of it to high-order SC is negligible. This fact suggests that it might not worth implementing time-skewing for high-order SC.

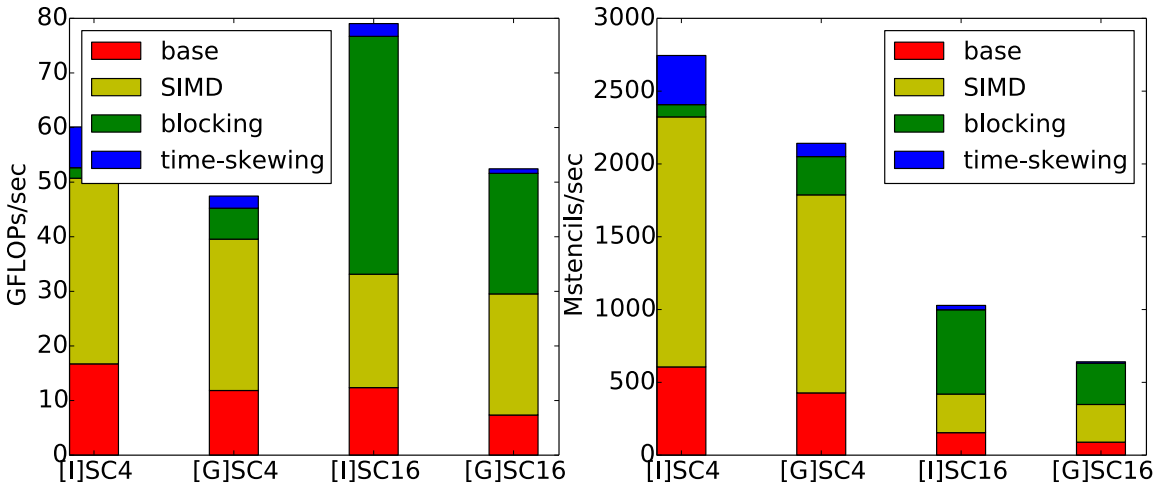


Figure 6.5: Contribution of each optimization method. Unaligned NAIVE kernel performance as the base, SIMD represents the performance gain due by using AUTOVEC kernel, blocking represents the performance gain due to thread-blocking method, time-skewing represents the performance gain by using time-skewing method.

Since the *machine peak* is 281.6 GLOPs/sec, depending on compiler in use, the final best tuned SC4 attains 16%∼21% of the machine peak, 22%∼28% of the algorithmic peak; SC16 achieves 18%∼28% of the machine peak, 24%∼37% algorithmic peak.

Chapter 7

Conclusion

In this work, I have optimized the acoustic constant density wave equation based *FDTD stencil* of size 256^3 on a Sandy Bridge Xeon E5-2660 processor. I have implemented two categories of optimizations: vectorization and cache optimization, and compared the effectiveness of each method to SC in terms of GFLOPs/sec and Mstencil/sec increment.

To obtain fully-vectorized stencil codes of any order, I have manually vectorized the code using intrinsic functions with explicit loop unrolling, and I have changed the loop structure and added compiler pragmas to facilitate compiler auto-vectorization. By padding the array memory, I make boundary alignment an available option. The final experiments show that both intrinsic and auto-vectorized codes are fully-vectorized, and the auto-vectorized ones achieve comparable performance as the intrinsic ones, which suggests that the effort required to code with intrinsics might not necessary for SC. In addition, my experiments show that except for aligned SSE codes, explicit loop unrolling achieves little performance gain for intrinsic codes. Also, even though memory alignment enables the compiler to produce greater unrolling factor, less load/store instructions, and higher vectorization ratio, this trick also barely affects the final run time. Lastly, I have benchmarked the tuned EXTEND kernel,

which is widely adopted in scientific applications including IWAVE but lacks the flexibility of changing stencil order. This kernel cannot be vectorized by gcc 4.8.2, but it can be vectorized by icc 14.0.0, and its run time is similar to that of my intrinsic and auto-vectorized kernel.

The second part of optimizations focus on hiding the long memory latency and limited memory bandwidth. I have used the roofline model to verify that SCs of both orders are bandwidth bound on Xeon E5-2660 processor. To hide memory latency, I updates the stencil first along the unit-stride dimension, and left this dimension uncut when performing blocking so as to have the hardware prefetching constantly work at its peak. To break the limited bandwidth bottleneck, I proposed two methods: one is thread-blocking method, the other is parallelized time-skewing that combines the features of both conventional serial time-skewing with thread-blocking scheme. I also presented the separate-and-interchange method extracted from flexSIMD.cpp (Stork), which targets on high-order SC optimizations. My experiments show that parallelized time-skewing method can further speed up fully-vectorized low- and high-order SC by 1.2x and 2x respectively. However, in high-order SC case, it is the thread-blocking that makes the major contribution to this additional speed up, and the contribution of time-skewing is negligible. Separate-and-interchange method provides similar improvement for high-order SC as thread-blocking method, but it slows down low-order SC.

The final best tuned kernels works with both icc 14.0.0 and gcc 4.8.2 compiler and achieve 20%$\sim$30% peak performance of the machine.

Appendix A

Xeon E5-2660 Latency Curve

FigIn Figure A.1, each line represents the averaged observed latency in fetching each data element from an array of varying size. The line property "stride" is the address distance between any of two adjacent array elements. The experiment is tested on our test device with hardware prefetching only, benchmarked by *lmbench* [McVoy and Staelin, 1996]. The cache line size is 64 KB.

Initially the array resides in DRAM. If the array is small enough to fit into a certain cache, then after it is traversed once, it will leave a copy in that cache. The height of each "stair" reflects the observed latency related to the cache which has the least capacity to hold the entire array with size in a fixed range. Since it takes longer time to access data from cache far away from CPU, "jumps" appear in the latency curve no matter which stride it uses. The height of the highest stair in each curve relates to the DRAM.

Another observation is that the curve with shorter stride has relatively lower observed DRAM latency. This could be explained by the hardware prefetching engine is engaged by short-stride accessing. When stride size exceeds 256 bytes, no hardware prefetching is present, the observed latency reflects the real DRAM to CPU latency, about 90 ns; when stride size is less or equal than 128 bytes, hardware prefetching is automatically enabled and it helps to reduce observed memory latency; when stride size decreases to 32 bytes, doubling of cache hits per cache line further helps the reduction, and now the observed DRAM latency is very close to the observed L2 cache latency at lower "stair", which is less than 10 ns.

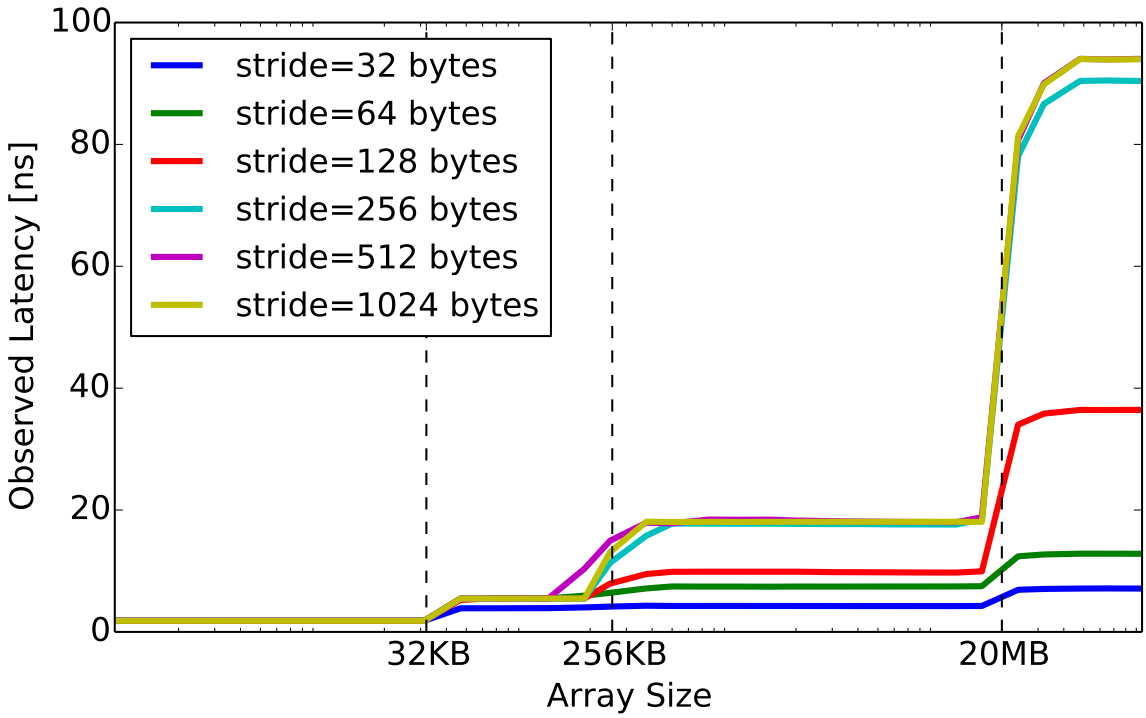


Figure A.1: Xeon E5-2660 latency curve. This experiment demonstrates that (1) accessing data from an array of size exceeding L3 cache will result in longer observed latency; (2) short-stride memory layout can engage hardware prefetching to reduce the latency.

Appendix B

Finite Difference Coefficients

Based on central finite difference scheme approximating the second order derivative, the following table presents coefficients of finite difference terms at various offsets and for various truncation error orders.

Error Order	Cofficients at Offset 0, 1, 2, ...
2	-2 1
4	$\frac{-5}{2}$ $\frac{4}{3}$ $\frac{-1}{12}$
8	$\frac{-205}{72}$ $\frac{8}{5}$ $\frac{-1}{5}$ $\frac{8}{315}$ $\frac{-1}{560}$
16	$\frac{-1671}{547}$ $\frac{16}{9}$ $\frac{-14}{45}$ $\frac{112}{1485}$ $\frac{-7}{396}$ $\frac{112}{32175}$ $\frac{-2}{3861}$ $\frac{16}{315315}$ $\frac{-1}{411840}$

For example, the fourth-order finite difference computation along X axis is:

$$\begin{aligned}\frac{\partial^2 u(x,y,z,t)}{\partial x^2} = \{&-\frac{5}{2}\,u(x,y,z,t) + \frac{4}{3}\,[u(x+\Delta x,y,z,t) + u(x-\Delta x,y,z,t)] \\ &-\frac{1}{12}\,[u(x+2\Delta x,y,z,t) + u(x-2\Delta x,y,z,t)]\}/\Delta x^2 + \mathcal{O}(\Delta x^4)\end{aligned}$$

Appendix C

Boundary Effects

The following table shows the time spent in computing the boundary condition. When codes get optimized, the part computing the boundary is untouched. This table records the run time of AUTOVEComp kernel. "With boundary" indicates the kernel is implemented with boundary condition, while "Without boundary" is not. The difference between two records in each column reflects the boundary computation time under different order or compiler.

	SC4		SC16	
Kernel	[I]Time(s)	[G]Time(s)	[I]Time(s)	[G]Time(s)
With boundary	35	41	84	133
Without boundary	32	39	77	128

Table C.1: AUTOVEComp run time with(out) boundary.

The observation is that the time spent in computing the boundary is less than 10% in the total run time of the second fastest kernel – AUTOVEComp.

Appendix D

Sandy Bridge Instruction Table

The two tables below are cropped from Fog's instruction table for Sandy Bridge architecture (pp147-160, [Fog, 2013]).

Table1: Floating Point X87 Instructions

Instruction	Latency	Throughput
FADD/FSUB	3	1
FMUL	5	1

Table2: Floating Point XMM and YMM Instructions

Instruction	Latency	Throughput
ADDSS/SUBSS/ADDPS/ SUBPS/VADDPS/VSUBPS	3	1
MULSS/MULPS/VMULPS	5	1

Latency is the number of clock cycles taken in executing an instruction.

Throughput is the number of same instructions output from the pipeline per cycle.

SIMD scalar instructions are denoted with suffix "SS", and vector instructions are denoted with "PS". 256-bit vector instructions are denoted with prefix "V", the rest of the vector instructions are 128-bit. All the values listed here are measured for inter-register operations.

Since the latency and throughput are the same in executing 256-bit vector instruction and its 128-bit or scalar equivalent, choosing 256-bit instructions can boost GFLOPS.

Appendix E

Effects of Memory Alignment on Compiler Auto-vectorization

This appendix uses a code snippet (Listing E.1) from AUTOVEC kernel to demonstrate that if both compilers have been notified of memory alignment, they can issue fewer instructions. In particular, Intel compiler will unroll the loop statement more aggressively so that more vector registers will be in use. In addition, it has been observed that the Intel compiler uses register data more efficiently and it can make SIMD instruction directly work on memory data, which to some extent explains why code compiled by the Intel compiler runs faster than the gcc one.

Listing E.1 is the i-loop-1 in AUTOVEC.

```
1 out_tmp=&U_out[k][j][1];
in_tmp=&U_in[k][j][1];
3 for(i = 0; i < nx; i++)
  out_tmp[i] = c0*in_tmp[i] - out_tmp[i];
```

Listing E.1: i-loop-1 in AUTOVEC.

E.1 Effects of Memory Alignment on gcc Auto-vectorization

When the memory addresses of `out_tmp[0]` and `in_tmp[0]` are unaligned, or the compiler is unaware of aligned memory, the compiler would generate assembly codes as shown in Listing E.2. An eight SPFP data chunk is loaded (stored) into (from) a YMM register in two batches, four SPFP per batch. A single 256-bit vectorization of the statement in L4 in Listing E.1 is executed by 9 instructions Listing E.2 (L3, 4, 5, 6, 8, 9, 13, 15). The assembly code also reveals that the compiler has unrolled L4 in Listing E.1 8 times to keep all the YMM registers busy, which can be deducted from the number of `vbroadcastss` instruction.

```
%The comments only show the first vectorization process.
2 .L207
  vmovups 4(%r11,%r13), %xmm1   #load in_tmp[i:i+3] to xmm1.
4  vbroadcastss  40(%r14), %ymm0   #store c0 to ymm0.
```

```
vmovups 4(%rax,%r13), %xmm4   #load out_tmp[i:i+3] to xmm4.
vinsertf128 $0x1, 20(%r11,%r13), %ymm1, %ymm2   #load in_tmp[i+4;i+7] and xmm1 to ymm2, now ymm2 = in_tmp[i:i+7].
vmulps  %ymm0, %ymm2, %ymm3   #compute c0*in_tmp[i:i+7].
vmovups 36(%rax,%r13), %xmm11     #load out_tmp[i+8:i+11] to xmm11.
vinsertf128 $0x1, 20(%rax,%r13), %ymm4, %ymm5   #load out_tmp[i+4:i+7] and xmm4 to ymm5, now ymm5 = out_tmp[i:i+7].
vsubps  %ymm5, %ymm3, %ymm6   #compute c0*in_tmp[i:i+7]-out_tmp[i:i+7] and store the new out_tmp[i:i+7] in ymm6.
vmovups 68(%rax,%r13), %xmm0    #load out_tmp[i+16:i+19] to xmm0.
vinsertf128 $0x1, 52(%rax,%r13), %ymm11, %ymm12   #load out_tmp[i+12:i+15] and xmm11 to ymm12, now ymm12 = out_tmp[i+8:i+15].
vinsertf128 $0x1, 84(%rax,%r13), %ymm0, %ymm3   #load out_tmp[i+20:i+23] and xmm0 to ymm3, now ymm3 = out_tmp[i+16:i+23].
vmovups %xmm6, 4(%rax,%r13)   #store out_tmp[i:i+3] from ymm6 back to memory.
vextractf128  $0x1, %ymm6, 20(%rax,%r13)    #store out_tmp[i+4:i+7] from ymm6 back to memory.
vmovups 36(%r11,%r13), %xmm7
vbroadcastss  40(%r14), %ymm9
vinsertf128 $0x1, 52(%r11,%r13), %ymm7, %ymm8
vmulps  %ymm9, %ymm8, %ymm10
vsubps  %ymm12, %ymm10, %ymm13
vmovups 100(%rax,%r13), %xmm9
vinsertf128 $0x1, 116(%rax,%r13), %ymm9, %ymm10
vmovups %xmm13, 36(%rax,%r13)
vextractf128  $0x1, %ymm13, 52(%rax,%r13)
vmovups 68(%r11,%r13), %xmm14
vbroadcastss  40(%r14), %ymm1
vinsertf128 $0x1, 84(%r11,%r13), %ymm14, %ymm15
vmulps  %ymm1, %ymm15, %ymm2
vsubps  %ymm3, %ymm2, %ymm4
vmovups 132(%rax,%r13), %xmm1
vinsertf128 $0x1, 148(%rax,%r13), %ymm1, %ymm2
vmovups %xmm4, 68(%rax,%r13)
vextractf128  $0x1, %ymm4, 84(%rax,%r13)
vmovups 100(%r11,%r13), %xmm5
vbroadcastss  40(%r14), %ymm7
vinsertf128 $0x1, 116(%r11,%r13), %ymm5, %ymm6
vmulps  %ymm7, %ymm6, %ymm8
vsubps  %ymm10, %ymm8, %ymm11
vmovups %xmm11, 100(%rax,%r13)
vextractf128  $0x1, %ymm11, 116(%rax,%r13)
vmovups 132(%r11,%r13), %xmm12
vbroadcastss  40(%r14), %ymm14
vinsertf128 $0x1, 148(%r11,%r13), %ymm12, %ymm13
vmulps  %ymm14, %ymm13, %ymm15
vsubps  %ymm2, %ymm15, %ymm0
vmovups %xmm0, 132(%rax,%r13)
vextractf128  $0x1, %ymm0, 148(%rax,%r13)
vbroadcastss  40(%r14), %ymm5
vmovups 164(%r11,%r13), %xmm3
vmovups 164(%rax,%r13), %xmm7
vinsertf128 $0x1, 180(%r11,%r13), %ymm3, %ymm4
vmulps  %ymm5, %ymm4, %ymm6
vmovups 196(%rax,%r13), %xmm14
vinsertf128 $0x1, 180(%rax,%r13), %ymm7, %ymm8
vsubps  %ymm8, %ymm6, %ymm9
vmovups 228(%rax,%r13), %xmm5
vinsertf128 $0x1, 212(%rax,%r13), %ymm14, %ymm15
vinsertf128 $0x1, 244(%rax,%r13), %ymm5, %ymm6
vmovups %xmm9, 164(%rax,%r13)
vextractf128  $0x1, %ymm9, 180(%rax,%r13)
vmovups 196(%r11,%r13), %xmm10
vbroadcastss  40(%r14), %ymm12
vinsertf128 $0x1, 212(%r11,%r13), %ymm10, %ymm11
vmulps  %ymm12, %ymm11, %ymm13
```

```
    vsubps  %ymm15, %ymm13, %ymm1
66  vmovups %xmm1, 196(%rax,%r13)
    vextractf128  $0x1, %ymm1, 212(%rax,%r13)
68  vmovups 228(%r11,%r13), %xmm2
    vbroadcastss  40(%r14), %ymm0
70  vinsertf128 $0x1, 244(%r11,%r13), %ymm2, %ymm3
    vmulps  %ymm0, %ymm3, %ymm4
72  vsubps  %ymm6, %ymm4, %ymm7
    vmovups %xmm7, 228(%rax,%r13)
74  vextractf128  $0x1, %ymm7, 244(%rax,%r13)
    addq  $256, %r13
76  cmpq  $1024, %r13
    jne .L207
```

Listing E.2: Memory unaligned gcc assembly codes.

When the two array bases are aligned and the compiler is aware of it, gcc assembly codes are shown in Listing E.3. Now a single 256-bit vectorization of L4 in Listing E.1 can be completed by only 4 instructions in the assembly codes (L3, 4, 5, 6). Also, gcc has unrolled the statement 8 times to keep all the YMM registers busy.

```
  %The comments only show the first vectorization process.
2 .L207:
    vbroadcastss  40(%r14), %ymm0   #load c0 to ymm0
4   vmulps  (%rcx,%r13), %ymm0, %ymm1   #compute c0*in_tmp[i:i+7]
    vsubps  (%rax,%r13), %ymm1, %ymm2   #compute c0*in_tmp[i:i+7]-out_tmp[i:i+7]
6   vmovaps %ymm2, (%rax,%r13)    #store out_tmp[i:i+7] back to memory
    vbroadcastss  40(%r14), %ymm3
8   vmulps  32(%rcx,%r13), %ymm3, %ymm4
    vsubps  32(%rax,%r13), %ymm4, %ymm5
10  vmovaps %ymm5, 32(%rax,%r13)
    vbroadcastss  40(%r14), %ymm6
12  vmulps  64(%rcx,%r13), %ymm6, %ymm7
    vsubps  64(%rax,%r13), %ymm7, %ymm8
14  vmovaps %ymm8, 64(%rax,%r13)
    vbroadcastss  40(%r14), %ymm9
16  vmulps  96(%rcx,%r13), %ymm9, %ymm10
    vsubps  96(%rax,%r13), %ymm10, %ymm11
18  vmovaps %ymm11, 96(%rax,%r13)
    vbroadcastss  40(%r14), %ymm12
20  vmulps  128(%rcx,%r13), %ymm12, %ymm13
    vsubps  128(%rax,%r13), %ymm13, %ymm14
22  vmovaps %ymm14, 128(%rax,%r13)
    vbroadcastss  40(%r14), %ymm15
24  vmulps  160(%rcx,%r13), %ymm15, %ymm0
    vsubps  160(%rax,%r13), %ymm0, %ymm1
26  vmovaps %ymm1, 160(%rax,%r13)
    vbroadcastss  40(%r14), %ymm2
28  vmulps  192(%rcx,%r13), %ymm2, %ymm3
    vsubps  192(%rax,%r13), %ymm3, %ymm4
30  vmovaps %ymm4, 192(%rax,%r13)
    vbroadcastss  40(%r14), %ymm5
32  vmulps  224(%rcx,%r13), %ymm5, %ymm6
    vsubps  224(%rax,%r13), %ymm6, %ymm7
34  vmovaps %ymm7, 224(%rax,%r13)
    addq  $256, %r13
36  cmpq  $1024, %r13
    jne .L207
```

Listing E.3: Memory Aligned gcc Assembly Codes.

E.2 Effects of Memory Alignment on Intel Auto-vectorization

When array bases are unaligned or the compiler is unaware of the memory alignment, Intel assembly codes shown in Listing E.4 reveal that the compiler will complete a vectorized statement with 5 instructions (L3, 4, 5, 6, 7). Like icc compiler, Intel compiler also moves 8 SPFP to a vector register in two batches of equal size. But Intel compiler only unrolls the statement 2 times. In consequence, YMM11 to YMM15 registers are not in use.

Unlike gcc, Intel compiler can perform AVX arithmetic instructions on memory address directly (see L6, `vsubps` operates on `out_tmp[i:i+7]` from memory), eliminating the process of loading data to register. In addition, icc loads the coefficient variable in YMM4 in advance, while in gcc codes, the coefficient variable is loaded into a register multiple times (see the number of `broadcastss`). These two features of Intel codes could probably explain why Intel AUTOVEC codes runs faster than gcc AUTOVEC codes despite its failure in unrolling the statement.

```
1  %The comments only show the first vectorization process.
   ..B1.225:
3          vmovups    4(%rcx,%rbx,4), %xmm0   #load in_tmp[i:i+3] to xmm0
           vinsertf128 $1, 20(%rcx,%rbx,4), %ymm0, %ymm1   #load in_tmp[i+4:i+7] and
               xmm0 to ymm1, now ymm1 = in_tmp[i:i+7]
5          vmulps     %ymm1, %ymm4, %ymm2   #compute c0*in_tmp[i:i+7]
           vsubps     (%rdi,%rbx,4), %ymm2, %ymm6   #compute c0*in_tmp[i:i+7] -out_tmp[i:
               i+7]
7          vmovups    %ymm6, (%rdi,%rbx,4)    #store the new out_tmp[i:i+7] back to
               memory
           vmovups    36(%rcx,%rbx,4), %xmm7
9          vinsertf128 $1, 52(%rcx,%rbx,4), %ymm7, %ymm8
           vmulps     %ymm8, %ymm4, %ymm9
11         vsubps     32(%rdi,%rbx,4), %ymm9, %ymm10
           vmovups    %ymm10, 32(%rdi,%rbx,4)
13         addq       $16, %rbx
           cmpq       %rsi, %rbx
15         jb         ..B1.225
```

Listing E.4: Memory Unaligned Intel Assembly Codes.

When array bases are aligned and the Intel compiler is aware of it, as shown in Listing E.5, the compiler now vectorizes the statement by only 3 instructions (L3, 4, 5), and unrolls it 8 times, so that all the YMM registers are employed.

```
1  %The comments only show the first vectorization process.
   ..B1.160:
3          vmulps     (%r11,%rcx,4), %ymm10, %ymm0    #compute c0*in_tmp[i:i+7]
           vsubps     (%rbx,%rcx,4), %ymm0, %ymm1   #compute c0*in_tmp[i:i+7]-out_tmp[i:i
               +7]
5          vmovups    %ymm1, (%rbx,%rcx,4)    #store new out_tmp[i:i+7] back to memory
           vmulps     32(%r11,%rcx,4), %ymm10, %ymm2
7          vsubps     32(%rbx,%rcx,4), %ymm2, %ymm3
           vmovups    %ymm3, 32(%rbx,%rcx,4)
9          vmulps     64(%r11,%rcx,4), %ymm10, %ymm4
           vsubps     64(%rbx,%rcx,4), %ymm4, %ymm5
11         vmovups    %ymm5, 64(%rbx,%rcx,4)
           vmulps     96(%r11,%rcx,4), %ymm10, %ymm6
13         vsubps     96(%rbx,%rcx,4), %ymm6, %ymm7
           vmovups    %ymm7, 96(%rbx,%rcx,4)
```

```
15          vmulps     128(%r11,%rcx,4), %ymm10, %ymm8
            vsubps     128(%rbx,%rcx,4), %ymm8, %ymm11
17          vmovups    %ymm11, 128(%rbx,%rcx,4)
            vmulps     160(%r11,%rcx,4), %ymm10, %ymm12
19          vsubps     160(%rbx,%rcx,4), %ymm12, %ymm13
            vmovups    %ymm13, 160(%rbx,%rcx,4)
21          vmulps     192(%r11,%rcx,4), %ymm10, %ymm14
            vsubps     192(%rbx,%rcx,4), %ymm14, %ymm15
23          vmovups    %ymm15, 192(%rbx,%rcx,4)
            vmulps     224(%r11,%rcx,4), %ymm10, %ymm0
25          vsubps     224(%rbx,%rcx,4), %ymm0, %ymm1
            vmovups    %ymm1, 224(%rbx,%rcx,4)
27          addq       $64, %rcx
            cmpq       $256, %rcx
29          jb         ..B1.160
```

Listing E.5: Memory Aligned Intel Assembly Codes.

E.3 Memory Alignment and Event Counters

Memory alignment only reduces the number of data movement instructions. The number of arithmetic floating point instructions (`vaddps`, `vsubps` and `vmulps`) stays the same. Since *Perf* `SIMD256` event counts only the number of arithmetic operations, `SIMD256` value will not decrease due to memory alignment.

Appendix F

Perf Setup

Perf is a profiling tool available from linux kernel source code. It can gather event counts from the hardware counters.

The main reason for choosing *Perf* instead of *PAPI* for our profiling analysis is that *Perf* can give event counts for 128-bit and 256-bit vectorizations separately, while *PAPI* bundles these two different events together to a single event (`PAPI_VEC_SP`), making it impossible to tell the portion of 256-bit vectorization in total vectorizations.

Nikolic described here ([Nikolic, 2013a], [Nikolic, 2013b]) on how to use `showevtinfo` and `check_events` to find the *Perf* code names for hardware events. We are particularly interested in the following events:

Code Name	Event Name	Description
r530110	`FP_COMP_OPS_EXE:X87`	Number of X87 uops
r532010	`FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE`	Number of 128-bit scalar SPFP uops
r534010	`FP_COMP_OPS_EXE:SSE_PACKED_SINGLE`	Number of 128-bit packed SPFP uops
r530111	`SIMD_FP_256:PACKED_SINGLE`	Number of 256-bit packed SPFP uops
r53012e	`L3_LAT_CACHE:MISS`	Number of L3 cache misses

Then run the following command to gather event counts:

```
perf stat -e r530110,r532010,r534010,r530111,r53012e ./main
```

Denote the counts gathered for previous events by `X87`, `SCALAR`, `SIMD128`, `SIMD256`, `L3MISS`, then the total FLOPs for `main` is: $\texttt{X87} + \texttt{SCALAR} + 4 * \texttt{SIMD128} + 8 * \texttt{SIMD256}$. Since `X87` is order of magnitude smaller than the sum of remaining terms, this term is omitted in FLOPs computations in this work.

References

[Borges, 2011] Borges, L., 2011, 3d finite differences on multi-core processors. (available online at `http://software.intel.com/en-us/articles/3d-finite-differences-on-multi-core-processors`). 2, 3, 4.2

[Datta, 2009] Datta, K., 2009, Auto-tuning stencil codes for cache-based multicore platforms: PhD thesis, University of California at Berkeley. 2

[Datta et al., 2008] Datta, K., M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, 2008, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 1–12. 2

[Demmel, 2014] Demmel, J., 2014, Case study: Tuning matrix multiply. (available online at `http://www.cs.berkeley.edu/~demmel/cs267_Spr14/`). 1.1

[Dursun et al., 2012] Dursun, H., M. Kunaseth, K. Nomura, J. Chame, R. Lucas, C. Chen, M. Hall, R. Kalia, A. Nakano, and P. Vashishta, 2012, Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters: J Supercomput, 946–966. 2

[Dursun et al., 2009] Dursun, H., K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. Kalia, A. Nakano, and P. Vashishta, 2009, In-core optimization of high-order stencil computations: In PDPTA, 533–538. 2

[Etgen and O'Brien, 2007] Etgen, J., and M. O'Brien, 2007, Computational methods for large-scale 3d acoustic finite-difference modeling: A tutorial: Geophysics, **72**, SM223–SM230. 1, 1, 2, 3.2

[Fog, 2013] Fog, A., 2013, Instruction tables lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. (available online at `http://www.agner.org/optimize/instruction_tables.pdf`). 2, D

[Henretty et al., 2011] Henretty, T., K. Stock, L. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, 2011, Data layout transformation for stencil computations on short-vector simd architectures: Proceedings of the 20th International Confer-

ence on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, Springer-Verlag, 225–245. 2

[Intel, 2012] Intel, 2012, Using avx without writing avx. (available online at `http://software.intel.com/en-us/articles/using-avx-without-writing-avx-code`). 1, 2, 3, 4.3

[Intel, 2014] ———, 2014, The intel intrinsics guide. (available online at `http://software.intel.com/sites/landingpage/IntrinsicsGuide/`). 2, 2

[Lomont, 2012] Lomont, C., 2012, Introduction to intel advanced vector extensions. (available online at `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`). 2

[McCalpin, 1995] McCalpin, J. D., 1995, Memory bandwidth and machine balance in current high performance computers: IEEE Technical Committee on Computer Architecture (TCCA) Newsletter. 2, 3

[McCalpin, 2013] ———, 2013, Stream: Sustanable memory bandwidth in high-performance computers. (available online at `www.cs.virginia.edu/stream`). 3

[McVoy and Staelin, 1996] McVoy, L., and C. Staelin, 1996, lmbench: Portable tools for performance analysis: Presented at the Proceedings of USENIX 1996 Annual Technical Conference. A

[Moczo et al., 2007] Moczo, P., J. O. Robertsson, and L. Eisner, 2007, The finite-difference time-domain method for modeling of seismic wave propagation, *in* Advances in wave propagation in heterogeneous Earth: Elsevier - Academic Press, London, UK, volume **48** *of* Advances in Geophysics, 421–516. 1, 3.2, 3.3

[Nguyen et al., 2010] Nguyen, A., N. Satish, J. Chhugani, C. Kim, and P. Dubey, 2010, 3.5-d blocking optimization for stencil computations on modern cpus and gpus: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 1–13. 2

[Nikolic, 2013a] Nikolic, B., 2013a. (available online at `http://www.bnikolic.co.uk/blog/hpc-prof-events.html`). F

[Nikolic, 2013b] ———, 2013b. (available online at `http://www.bnikolic.co.uk/blog/hpc-howto-measure-flops.html`). F

[Strzodka et al., 2011] Strzodka, R., M. Shaheen, D. Pajak, and H. Seidel, 2011, Cache accurate time skewing in iterative stencil computations: Proceedings of the International Conference on Parallel Processing (ICPP), IEEE Computer Society, 571–581. 2

[Symes, 2013] Symes, W., 2013, Using iwave. (available online at `http://www.ahay.org/RSF/book/trip/iwave/paper.pdf`). 3.3

[Terentyev, 2009] Terentyev, I. S., 2009, A software framework for finite difference simulation: Technical Report TR09-07, Department of Computational and Applied Mathematics, Rice University. 3.3

[Williams et al., 2009] Williams, S., A. Waterman, and D. Patterson, 2009, Roofline: An insightful visual performance model for multicore architectures: Communications of the ACM, **52**, 65–76. 2, 5.1, 4

[Wonnacott, 1999] Wonnacott, D., 1999, Achieving scalable locality with time skewing: International Journal of Parallel Programming, **30**, 2002. 2

[Wonnacott and Strout, 2012] Wonnacott, D. G., and M. M. Strout, 2012, On the scalability of loop tiling techniques: Technical Report HC-CS-TR-2012-01, Department of Computer Science, Haverford College. 2

[Zumbusch, 2012] Zumbusch, G., 2012, Tuning a finite difference computation for parallel vector processors: ISPDC, 63–70. 2

[Zumbusch, 2013] ———, 2013, Vectorized higher order finite difference kernels, *in* Applied Parallel and Scientific Computing: Springer Berlin Heidelberg, volume **7782** *of* Lecture Notes in Computer Science, 343–357. 2