Wave Equation Based Stencil Optimizations on Multi-core CPU

Muhong Zhou and William W. Symes, Rice University*

SUMMARY

As the engine for seismic imaging algorithms, stencil kernels modeling wave propagation 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.

INTRODUCTION

Finite Difference (FD) stencil computations are very easy to implement, but it is also compute- and memory-intensive. To make full use of the computing power of modern multi-core X86 based CPU architectures, FD stencil code has to be parallelized at various levels. In particular, data-level parallelization, .i.e., vectorization, generated by SIMD packed instructions, plays a key role in optimizing serial code. With a good optimized serial code, OpenMP and MPI then can add intra and inter-node parallelizations. However, to get good SIMD vectorization without manually inserting SIMD assembly instructions or intrinsics (Intel, 2014), loop structure and memory access pattern have to meet certain criteria (Intel, 2011). Sometimes, vector pragmas and memory alignment hints need to be inserted before the loop to assist compiler to generate vectorization with desired vector length (Intel, 2011; Borges, 2011).

Even if the stencil codes are effectively parallelized and vectorized, their performance may still be constrained by the limited bandwidth from RAM to cache. For example, our test device, a Xeon E5-2660 processor, has a *machine balance* (the ratio between peak machine GFLOPs/sec and maximum memory-to-cache bandwidth (McCalpin, 1995)) of 8.68 FLOPs/byte, while the *operation intensity* (floating operations per byte transferred from RAM to cache (Williams et al., 2009)) of the stencil code varies from 1.67 FLOPs/byte (4-th order) to 6.17 FLOPs/byte (16-th order), both of which are smaller than the machine balance of the test device, indicating that the performance bottleneck for FD stencil codes comes from the limited memory bandwidth.

To reduce the impact of this hardware deficiency on code performance, it is essential to mitigate memory traffic by reducing cache misses, or alternatively, increasing cache hits, by exploiting spatial and temporal locality of the cached data. This task is often accomplished by performing loop blocking along spatial dimensions or time dimension (Etgen and O'Brien, 2007; Imbert et al., 2011).

In this paper, we will demonstrate that with appropriate loop structure re-arrangement, memory alignment, and insertion of compiler hints, both icc 14.0.0 and gcc 4.8.2 compilers can generate auto-vectorized codes with any input stencil order, having comparable performances to that of manually vectorized SIMD codes. Then we will show that loop interchanging (C. Stork, personal communication, 2013), blocking thread executions using OpenMP pragma options, and time-skewing can significantly reduce cache misses and further improve stencil code performance.

WAVE EQUATION STENCIL CODES

Acoustic-wave propagation in constant density fluid medium can be captured by the following equation:

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} = c^2(x,y,z)\nabla^2 u(x,y,z,t) \tag{1}$$

Here $(x,y,z) \in \Omega$, a 3D computational domain, u is the pressure field, $c^2(x,y,z)$ is the squared velocity field, equaling to $\kappa(x,y,z)/\rho(x,y,z)$, where κ and ρ are the bulk modulus and density fields respectively.

We assume c to be a constant, then the corresponding `r`-th order in space and 2nd order in time stencil codes based on central FD scheme (Moczo et al., 2007) is:

```
//Time loop{
for(k=0; k<nz; k++)//k-loop
 for(j=0; j<ny; j++)//j-loop
  for(i=0; i <nx; i++){//i-loop
   U_out[k][j][i]=c0*U_in[k][j][i]-U_out[k][j][i];
   for(ixyz = 1; ixyz <= r/2; ixyz++)//ixyz-loop
    U_out[k][j][i]+=cx[ixyz]*(U_in[k][j][ixyz+i]+U_in[k][j
         ][-ixyz+i]);
   for(ixyz = 1; ixyz <= r/2; 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/2; ixyz++)
    U_out[k][j][i]+=cz[ixyz]*(U_in[ixyz+k][j][i]+U_in[-
         ixyz+k][j][i]);}   // + process boundary data}
```

Listing 1: NAIVE – stencil kernel with no optimization.

Here two arrays `U_out` and `U_in` keep the pressure field values at previous and current time levels, and the values computed for the next time level are kept in `U_out`. In our 3D indexing scheme, `U_out[k][j][i]` denotes `U_out` value at grid point (`k`,`j`,`i`), where `i`, `k` are the most and least contiguous dimensions.

We have imposed Dirichlet boundary conditions by allocating extra array memory to build up ghost elements around the computational domain boundary, so that near-boundary stencil computations can stay the same as central stencil computations, eliminating possible slow down caused by branching statements.

VECTORIZATION

Current X86 based CPU architectures have SIMD packed instructions that can perform four or eight single precision floating point (SPFP) operations simultaneously on wide vector registers. On the other hand, the throughput and latency of issuing one SIMD packed instruction are the same as the corresponding old X87 instruction or SIMD scalar instruction, both of which only perform one arithmatic SPFP operation at a time (Fog, 2013). Therefore, code performance can be improved significantly by processing more floating point operations with SIMD packed instructions.

Using SIMD intrinsics (Intel, 2014) can directly lead the compiler to generate corresponding SIMD packed instructions, but requires the programmer to take care of data movements between vector registers and memory, and express the normal arithmetic operations in terms of SIMD intrinsic functions. An alternative approach is to let the compiler automatically generate SIMD packed instructions from C source code. Compilers only auto-vectorize the innermost loop, and its array references must be contiguous. To accommodate this fact, we have rearranged the loop structure of NAIVE kernel. The resulting AUTOVEC kernel is in Listing 2.

```
//Time loop{
for(k=0; k<nz; k++)//k-loop
 for(j=0; j<ny; j++){//j-loop
  for(i=0; i<nx; i++)//i-loop-1
   U_out[k][j][i]=-U_out[k][j][i]+c0*U_in[k][j][i];
  for(ixyz=1; ixyz<=r/2; ixyz++){//ixyz-loop
   #pragma ivdep
   for(i=0; 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 boundary data}
```

Listing 2: AUTOVEC, based on NAIVE kernel, with rearranged loop structure to assist compiler auto-vectorization.

For the icc compiler, the ivdep pragma is used to remove data dependencies assumed by the default compiler heuristics. Vectorization reports from both gcc and icc compilers suggest that this kernel can be fully vectorized by both compilers.

Both compilers will generate fewer instructions moving data between vector registers and memory when memory references are aligned. Therefore we have aligned the first addresses of all contiguous loop segments (U_out[k][j][0] and U_in[k][j][0]) on 32-byte boundaries by padding (over-allocating) memory. We have also notified compilers about memory alignment using compiler hints: __assume_aligned() for icc and __builtin_assume_aligned() for gcc.

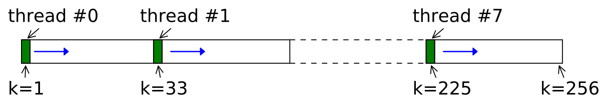
REDUCING L3 CACHE MISSES

Memory traffic results from cache misses. In practice, the size of two array usually exceeds cache capacity, and even processor RAM (Etgen and O'Brien, 2007). As a result, U_in arrays may be loaded multiple times per time step. However, the limited RAM to cache bandwidth may prevent U_in from being loaded into CPU right when CPU needs it, which slows down CPU execution. In this section, we presented three methods to reduce cache misses in the context of OpenMP parallelization. The first two methods reduce cache misses by exploiting data spatial locality, the third method is a combination of the first method and the traditional time-skewing method that exploits data temporal locality to reduce cache misses.

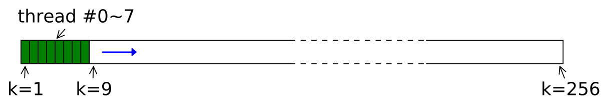
Blocking thread accesses

If (r+4) i-j planes can reside in the cache concurrently, then for serial program, there is no point in performing spatial loop blocking. However, for OpenMP parallelized program, as Figure 1(a) shows, without specifying schedule(static,1) in omp pragma, each thread will work on a chunk of contiguous planes, so the planes being updated simultaneously are discretely located along Z direction. If Z dimension is large enough that U_in planes used by adjacent threads do not overlap, and if the cache can not hold NP(r+4) planes, where NP is the number of OpenMP threads, then cache misses will occur.

The OpenMP directive schedule(static,1) tells each thread to work on only one plane at a time, forcing all threads to work on a chunk of contiguous planes simultaneously, as illustrated in Figure 1(b). Thus the spatial locality of U_in planes will be improved. For instance, the U_in plane assigned to the 0th thread will also be used by the 1st to 7th threades for 8-th order stencils. In addition, if the cache can hold (r+4NP) planes, then when these threads update the next chunk of U_out planes, they re-use some U_in planes already in cache.



(a) Omp without schedule(static,1)



(b) Omp with schedule(static,1)

Figure 1: Memory access pattern of different threads. The horizontal axis is k-axis. Each green chunk represents an i-j plane processed by one thread. The blue arrows denote the update direction of the thread(s).

Separate-and-interchange method

Listing 3 shows separate-and-interchange method (C. Stork, personal communication, 2013), which reduces cache misses by changing array access pattern. The idea is to isolate the partial update to U_out along Z direction, then interchange the two outermost loops for this partial update.

```
//Time loop{
for(k=0; k<nz; k++)//k-loop
 for(j=0; j<ny; j++){//j-loop
  for(i=0; i<nx; i++)//i-loop-1
   U_out[k][j][i]=-U_out[k][j][i]+c0*U_in[k][j][i];
  for(ixyz=1; ixyz<=r/2; ixyz++){//ixyz-loop
   #pragma ivdep
   for(i=0; 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])}}
for(j=0; j<ny; j++)//j-loop
 for(k=0; k<nz; k++)//k-loop
  for(ixyz=1; ixyz<=r/2; ixyz++)//ixyz-loop
   for(i=0; i<nx; i++)//i-loop
    U_out[k][j][i]+=cz[ixyz]*(U_in[ixyz+k][j][i]+U_in[-
         ixyz+k][j][i]); // + process boundary data}
```

Listing 3: AUTOVECz, based on AUTOVEC kernel and separate-and-interchange method.

With this method, the first partial update to `U_out[k][0:ny-1][0:nx-1]` only requires the plane itself and `U_in[k][0:ny-1][0:nx-1]`. The cache capacity may allow 2`NP` planes reside in it concurrently. So the thread may load each array to cache only once for the first sweep over the entire computational domain. During the second sweep, updating `U_out[0:nz-1][j][0:nx-1]` still requires the plane itself and `U_in[0:nz-1][j][0:nx-1]`. Therefore, two arrays may have been loaded into cache only twice every time step.

Both methods discussed so far (blocking thread access, loop interchange) are effective only if L3 cache can load sufficient entire array planes. Otherwise, the outer loops will need to be blocked as well.

Time-skewing method

No matter how we exploit data spatial locality by changing array access pattern, or by performing spatial loop blocking, two arrays must have been loaded into cache at least once per time step. To mitigate this bottleneck, it is necessary to use temporal loop blocking, i.e., time-skewing (Etgen and O'Brien, 2007; Datta et al., 2009).

Our time-skewing scheme is based on the first method in this section. As illustrated in Figure 2, at each time step, all the OpenMP threads together update a chunk containing `NP` contiguous `i-j` planes, then in the next time step, they will update `NP` planes on the next upper stair. In Figure 2, right before the update in the current round, if the cache is large enough that it still holds `U_in[0:31][0:ny-1][0:nx-1]` and `U_out[8:23][0:ny-1][0:nx-1]`, then to get `U_out[8:15][0:ny-1][0:nx-1]` at (t+3) time level, it suffices to load `U_out[24:31][0:ny-1][0:nx-1]` and `U_in[32:39][0:ny-1][0:nx-1]` into the cache. In short, it only needs to load two plane chunks in order to get a plane chunk over 3 time steps.

Theoretically, if the cache can hold $2\mathtt{NTS}*\mathtt{NP}+\mathtt{r}$ planes, where `NTS` is the number of time steps leaped over per round, then every round the program will read both arrays only once from RAM, and write one array once to RAM, which is equivalent to about 1/`NP` loads of either array and 1/`NP` stores of only one array per time step.

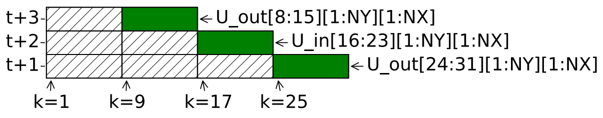


Figure 2: Using time-skewing method to update data values at three time levels per iteration. The hatched chunks represent data already processed by time-skewing method, the green chunks are data being updated at current round. Here assume OpenMP forks 8 threads in total, and each data chunk contains 8 planes that can be assigned to these OpenMP threads equally.

RESULTS

Experiment setup

Our test device is an Intel Xeon E5-2660 processor, containing 8 cores of 2.2 GHz. Each core has a L1 cache of 32KB, a L2 cache of 256KB, and a shared L3 cache of 20MB. Theoretical peak GFLOPs/sec of the machine is 281.6 GFLOPs/sec. Computational domain size is 256x256x256, and total time steps is 5001. The compilers in use are icc 14.0.0 and gcc 4.8.2. Parallelization are provided by OpenMP, which forks 8 threads in total, with one thread per core.

Vectorization

Run time results of AUTOVEC kernel for both low- and high-order stencil codes are listed in Table 1. For comparison purpose, the table has also listed run time results of the following kernels:

- EXTEND: based on NAIVE, with its coefficient loops unrolled, making i-loop the innermost loop. We have also aligned its array memory, and added vector pragmas to assist auto-vectorization. Vectorization reports showed that icc can fully vectorized this kernel while gcc cannot.
- AVX: SIMD intrinsics codes based on NAIVE, with aligned array memory. In particular, we have also explicitly unrolled i-loop twice when using gcc compiler. Based on our experiment, this unrolling factor will give the shortest run time for gcc AVX codes.

	SC4		SC16	
Kernel	[I]Time(s)	[G]Time(s)	[I]Time(s)	[G]Time(s)
NAIVE	138	196	543	940
EXTEND	36	152	202	838
AVX	36	47	198	244
AUTOVEC	36	47	200	241

Table 1: Run time results of different kernels. SC# represents using #-th order stencil. [I]Time, [G]Time are the run time when the kernel is compiled by icc 14.0.0, gcc 4.8.2 respectively.

When using either compiler, and either low-order or high-order stencil codes, AUTOVEC is about 4 times faster than NAIVE kernel, and it is also a little faster than intrinsics codes in most cases. Also, with the help of vectorization pragmas, icc compiler has already vectorized EXTEND kernel quite well. But compared with AUTOVEC and AVX kernels, EXTEND lacks the coding flexibility of changing stencil order `r`, and the gcc compiler does not vectorize it.

Reducing L3 cache misses

The kernels below are designed to test the effectiveness of the three methods in reducing L3 cache misses and run time.

- AUTOVECz: Based on AUTOVEC and separate-and-interchange method.
- AUTOVEComp: Based on AUTOVEC and blocking thread accesses method.
- AUTOVECts: Based on time-skewing and blocking thread accesses methods. The temporal blocking factor that produces least amount of observed L3 cache misses is 3 for 4-th order stencil codes and 2 for 8-th order stencil codes.

Table 2, 3 give the cache line misses and timing results.

	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 2: L3 cache misses of different kernels. Notations are the same as the ones used in Table 1. Miss is the number of L3 cache line misses measured by Perf.

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

Table 3: Run time results of different kernels. Notations are the same as the ones used in Table 1.

Both AUTOVEC and AUTOVEComp have successfully reduced cache misses for high-order stencil codes, but for low-order stencil codes, AUTOVEComp has barely reduced cache misses, while AUTOVECz has generated more cache misses than AUTOVEC, probably because the reduction of cache misses due to re-arranged access pattern can not compensate for the increase in cache misses due to extra stencil traversal over the computational grid. The run time results also reflect the same trend. Compared with AUTOVEC run time results in Table 1, these two methods reduce the run time of each kernel about by half in high-order stencil cases, but only AUTOVEComp uses less time than AUTOVEC in 4-th order stencil codes.

After coupled with time-skewing method, the cache misses of AUTOVEComp dropped to about one third of its original value, but the run time didn't drop as significantly. The more time steps required by the program, the more significant improvement in the run time. As we increased the number of total time steps to 50001, about 10x of 5001, we observed that the run time difference between AUTOVEComp and AUTOVECts is 10x of it when total number of time steps is 5001. In addition, the run time reduction is more significant with low-order stencils: smaller operation intensity means that limited memory bandwidth affects it more.

Evaluation of different methods

In terms of two kinds of code performance measurements, Figure 3 shows the evaluation of the following stencil optimization methods: compiler automatic SIMD vectorization, blocking thread accesses, time-skewing based on previous blocking method. The plot reveals that both low- and high-order stencil codes have benefited a lot from automatic vectorization, but only high-order stencil codes have shown big improvement with cache misses reducing methods, especially the blocking method.

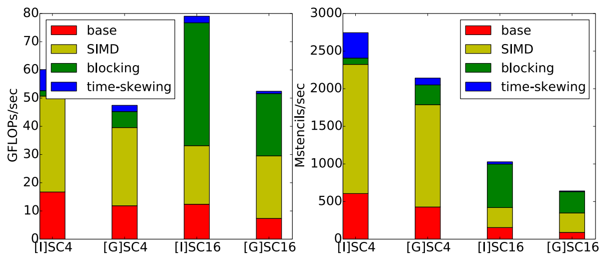


Figure 3: Stacked bar plot of contribution of each method to GFLOPs/sec and million stencils/sec of stencil codes. Notations are the same as the ones used in Table 1.

CONCLUSION

We have shown how to vectorize wave equation stencil code of any order. With appropriate compiler directives, pragmas, and memory alignment, applications compiled from our C source perform as well as those built out of explicit SIMD implicit function calls. We have documented further performance improvements through simple optimizations that significantly reduce L3 cache misses. These optimizations almost double the performance of high-order stencil code, to approximately 30% of peak floating point throughput on contemporary x86-based platforms. This performance is comparable to the best reported by other authors (Imbert et al., 2011), but at the cost of somewhat less intrusive code modification. With the expected increase in machine balance in the near future, even the most agressive memory optimizations, such as time-skewing, may become profitable for high-order stencil computations.

ACKNOWLEDGEMENTS

The authors thank Christof Stork of ION Geophysical for providing examples and advice. We are grateful to the sponsors of The Rice Inversion Project for their continued support.

APPENDIX A

THE SOURCE OF THE BIBLIOGRAPHY

```
@misc{AVX2,
author = "Borges, L",
title = "3D Finite Differences on Multi-core Processors",
note = "available online at \url{http://software.intel.com/en-us/articles/3d-finite-differences
-on-multi-core-processors}",
urldate = "2011-12-16",
year = "2011"
}

@article{Datta,
author = "Datta, K. and Kamil, S. and Williams, S. and Oliker, L. and Shalf, J. and Yelick, K.",
title = "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors",
journal = "SIAM Review",
pages = "129-159",
year = "2009",
volume = "51",
url = "http://dx.doi.org/10.1137/070693199",
doi = "10.1137/070693199",
publisher = "Society for Industrial and Applied Mathematics",
}

@article{TSBP,
author = "Etgen, J. and O'Brien, M.",
title = "Computational methods for large-scale 3D acoustic finite-difference modeling: A tutorial",
journal = "Geophysics",
volume = "72",
pages = "SM223-SM230",
year = "2007",
doi = "10.1190/1.2753753",
url = "http://library.seg.org/doi/abs/10.1190/1.2753753"
}

@misc{Fog13,
author = "Fog, A.",
title = "Instruction tables Lists of instruction latencies, throughputs and micro-operation
 breakdowns for Intel, AMD and VIA CPUs",
year = "2013",
note = "available online at \url{http://www.agner.org/optimize/instruction_tables.pdf}",
urldate = "2013-04-03",
pages = "145,158"
}

@misc{AVX1,
author = "Intel",
title = "Using AVX Without Writing AVX",
year = "2011",
note = "available online at \url{http://software.intel.com/en-us/articles/using-avx-without-
writing-avx-code}",
urldate = "2013-10-27"
}

@misc{intrinsics,
author = "Intel",
title = "The Intel Intrinsics Guide",
year = "2014",
note = "available online at \url{http://software.intel.com/sites/landingpage/IntrinsicsGuide/}",
urldate = "2014-02-06"
}

@article{mb,
author = "McCalpin, J.~D.",
title = "Memory Bandwidth and Machine Balance in Current High Performance Computers",
journal = "IEEE Technical Committee on Computer Architecture (TCCA) Newsletter",
year = "1995"
}

@INCOLLECTION{FDTDMoczo,
author = "Moczo, P. and Robertsson, J.~O.A.  and Eisner, L.",
title = "The Finite-Difference Time-Domain Method for Modeling of Seismic Wave propagation",
booktitle = "Advances in wave propagation in heterogeneous Earth",
volume = "48",
pages = "421-516",
series = "Advances in Geophysics",
publisher = "Elsevier",
year = "2007",
}

@article{will,
author = "Williams, S. and Waterman, A. and Patterson, D.",
title = "Roofline: An Insightful Visual Performance Model for Multicore Architectures",
journal = "Communications of the ACM",
volume = "52",
number = "4",
pages = "65-76",
month = "April",
year = "2009",
publisher = "ACM",
}

@INPROCEEDINGS{RTMFWITIPS,
author = "Imbert, D. and Imadoueddine, K. and Thierry, P. and Chauris, H. and Borges, L.",
title = "Tips and tricks for Finite difference and i/o-less {FWI}",
booktitle = "Proc. 81st Annual International Meeting",
year = "2011",
organization = "Society of Exploration Geophysicists",
pages = "3174-3178",
note = "Expanded abstract"
}
```

REFERENCES

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`).

Datta, K., S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, 2009, Optimization and performance modeling of stencil computations on modern microprocessors: SIAM Review, **51**, 129–159.

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

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`).

Imbert, D., K. Imadoueddine, P. Thierry, H. Chauris, and L. Borges, 2011, Tips and tricks for finite difference and i/o-less FWI: Proc. 81st Annual International Meeting, Society of Exploration Geophysicists, 3174–3178. (Expanded abstract).

Intel, 2011, Using avx without writing avx. (available online at `http://software.intel.com/en-us/articles/using-avx-without-writing-avx-code`).

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

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

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, volume **48** *of* Advances in Geophysics, 421–516.

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