# Wave Equation Stencil Optimization on Multi-core CPU

Muhong Zhou

TRIP 14 Annual Review Meeting

May 1, 2015

# Education Background

- BS in Math and Applied Math, Zhejiang University, 2007-2011
- MS in Computational and Applied Math, Rice University, 2011-2014
- PhD in Computational and Applied Math, Rice University, 2014-

# Contents

- Summary of my master thesis work.
- Ongoing PhD project.

# Acoustic Wave Equation and FDTD Stencil Kernel

Acoustic wave equation in fluid media:

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

Where $u$ is the pressure field and $c$ is the velocity (constant).

Approximate all the derivatives using the CFD scheme (2nd order in time and 2r-th (r=2,8) order in all three spatial dimensions):

$$u(x,y,z,t+\Delta t) = c_0 u(x,y,z,t) - u(x,y,z,t-\Delta t)$$
$$+ \sum_{i=1}^{r} c_i [u(x+i\Delta x,y,z,t) + u(x-i\Delta x,y,z,t)]$$
$$+ \sum_{i=1}^{r} c_i [u(x,y+i\Delta y,z,t) + u(x,y-i\Delta y,z,t)]$$
$$+ \sum_{i=1}^{r} c_i [u(x,y,z+i\Delta z,t) + u(x,y,z-i\Delta z,t)]$$

# Acoustic Wave Equation and FDTD Stencil Kernel

Acoustic wave equation in fluid media:

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

Where $u$ is the pressure field and $c$ is the velocity (constant). Approximate all the derivatives using the CFD scheme (2nd order in time and 2r-th (r=2,8) order in all three spatial dimensions):

$$\begin{aligned} u(x,y,z,t+\Delta t) = {} & c_0 u(x,y,z,t) - u(x,y,z,t-\Delta t) \\ & + \sum_{i=1}^{r} c_i [u(x+i\Delta x,y,z,t) + u(x-i\Delta x,y,z,t)] \\ & + \sum_{i=1}^{r} c_i [u(x,y+i\Delta y,z,t) + u(x,y-i\Delta y,z,t)] \\ & + \sum_{i=1}^{r} c_i [u(x,y,z+i\Delta z,t) + u(x,y,z-i\Delta z,t)] \end{aligned}$$

# NAIVE – stencil kernel with no optimization.

`U_out[x][y][z]` stores $u(x, y, z, t - \Delta t)$, $u(x, y, z, t + \Delta t)$;
`U_in[x][y][z]` stores $u(x, y, z, t)$.

```
for (t = 1; t <= NT; t++)
 for (k = 1; k <= NZ; k++)
  for (j = 1; j <= NY; j++)
   for (i = 1; i <= NX; i++){
    U_out[k][j][i]=c0*U_in[k][j][i]-U_out[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]);}}}
```

# Optimization Overview

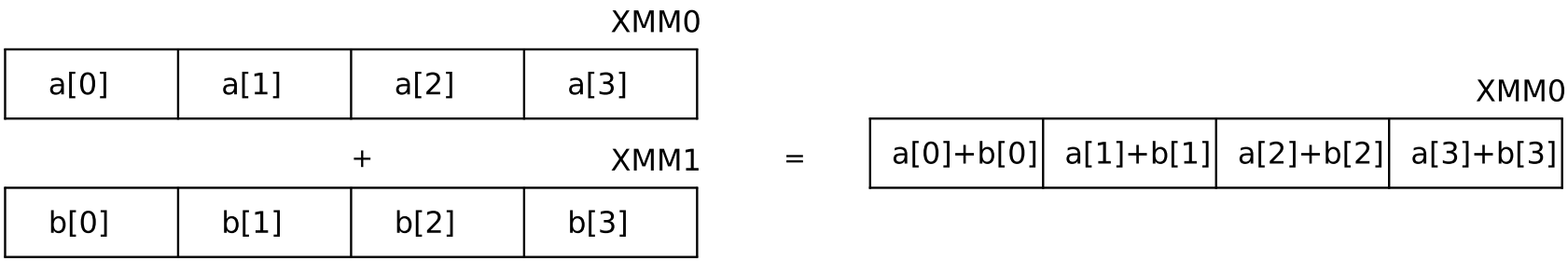
Based on wave equation stencil kernels, I apply:

- CPU SIMD vectorization techniques.
- CPU cache optimization techniques (work together with OpenMP parallelization).

# SIMD Technology

SIMD = Single Instruction, Multiple Data.

Figure : Packed SSE instruction `ADDPS` performing vector addition.




Packed SIMD (desired!) vs. Scalar SIMD

Figure : Scalar SSE instruction `ADDSS` performing scalar addition.

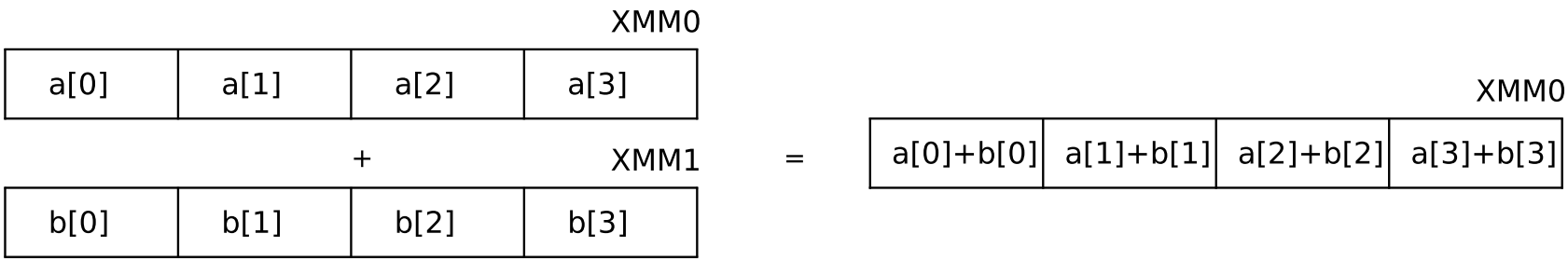



SIMD instruction sets available on Sandy Bridge: SSE, AVX

# SIMD Technology

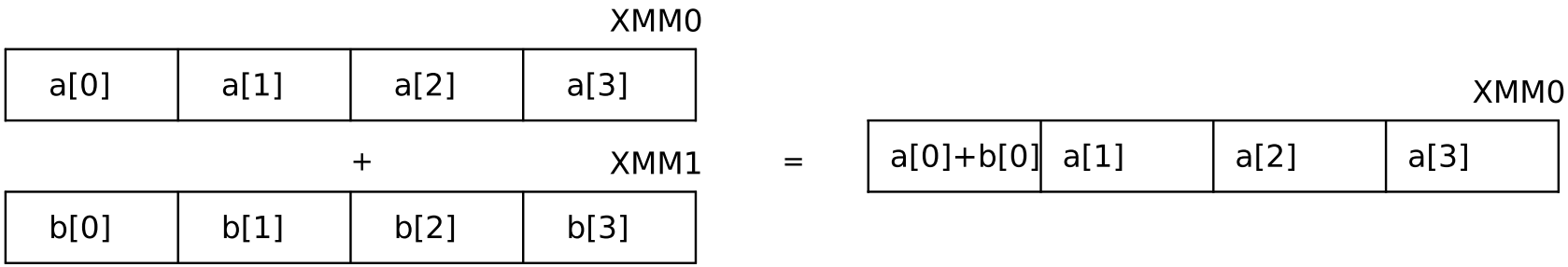
SIMD = Single Instruction, Multiple Data.

Figure : Packed SSE instruction `ADDPS` performing vector addition.




Packed SIMD (desired!) vs. Scalar SIMD

Figure : Scalar SSE instruction `ADDSS` performing scalar addition.




SIMD instruction sets available on Sandy Bridge: SSE, AVX

# Implement SIMD with descending difficulty

1. **Assembly languages**
2. **SIMD intrinsics**: function interfaces that can provide direct control over the generation of SIMD instructions.
   - Most of the literatures on stencil (Jacobi-type) vectorization use SIMD intrinsics. [Datta, 2009, Dursun et al., 2009, Henretty et al., 2011, Strzodka et al., 2011, Dursun et al., 2012, Zumbusch, 2012, Zumbusch, 2013].
   - Detailed work is shown in `flexSIMD.h` (Dr. Stork, personal communication)
   - `U_out[k][j][i]=c0*U_in[k][j][i]-U_out[k][j][i];`

```
__m256 out_simd=_mm256_load_ps(&U_out[k][j][i]);
__m256 in_simd=_mm256_load_ps(&U_in[k][j][i]);
__m256 c_simd=_mm256_broadcast_ss(&c0);
in_simd=_mm256_mul_ps(in_simd, c_simd);
out_simd=_mm256_sub_ps(in_simd, out_simd);
_mm256_store_ps(&U_out[k][j][i], out_simd);
```

3. **Auto-vectorization by compilers**: gcc, icc.
   - Borges demonstrated how to auto-vectorize a **fixed order** stencil (r=4). [Borges, 2011]

# Auto-vectorization By Compilers

Common violations prevent compiler auto-vectorization: [Intel, 2012]

- Not innermost loop
- Low loop count ($\leq 4$)
- Not unit-stride accessing
- Existence of potential data dependencies, e.g. a[i]=a[i-1]+1

NAIVE kernel has coefficient loops as innermost loops, it can not be auto-vectorized because:

- low loop count (if r$\leq$4)
- not unit-stride accessing when shifting along j, k dimensions.

# Auto-vectorization By Compilers

Steps:

1. Interchange i-loop and ixyz-loop to ensure long unit-stride
2. Add compiler hint (`#pragma ivdep`) to remove potential vector dependencies for icc.
3. Double-check with -S.

```
//t, k, j, i-loops wrapped outside.
U_out[k][j][i]=-U_out[k][j][i]+c0*U_in[k][j][i];
for(ixyz=1; ixyz<=r; ixyz++){
  #pragma ivdep
  for(i=1; i <= nx; i++)
   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])
        ;}}
```

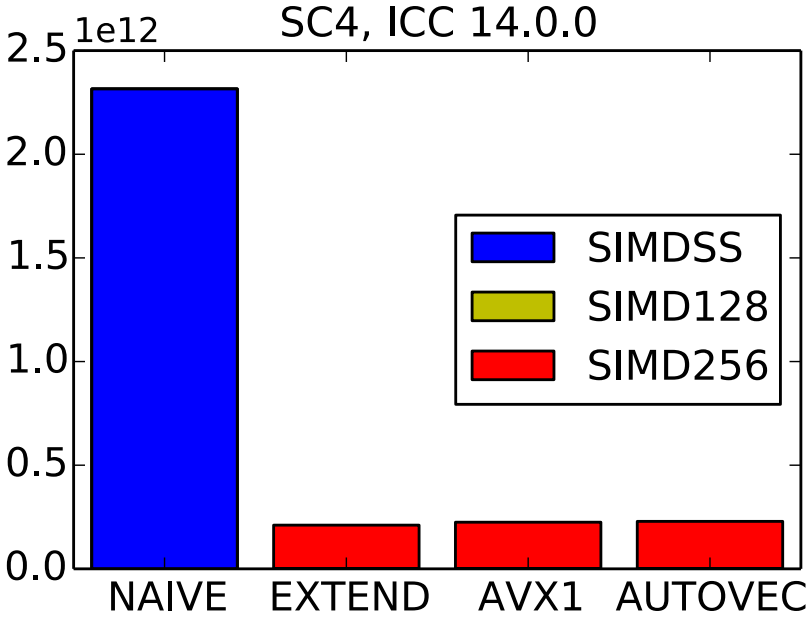
Listing 1: AUTOVEC

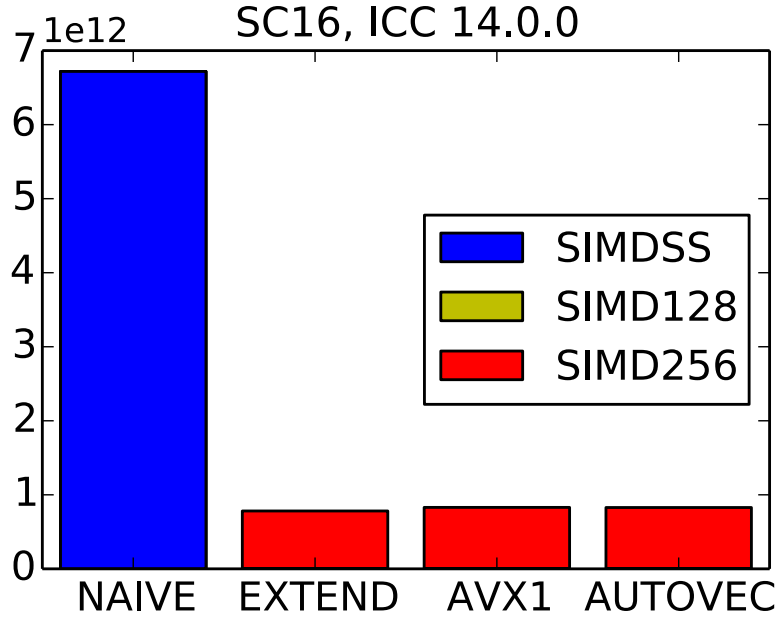
# Auto-vectorization Results

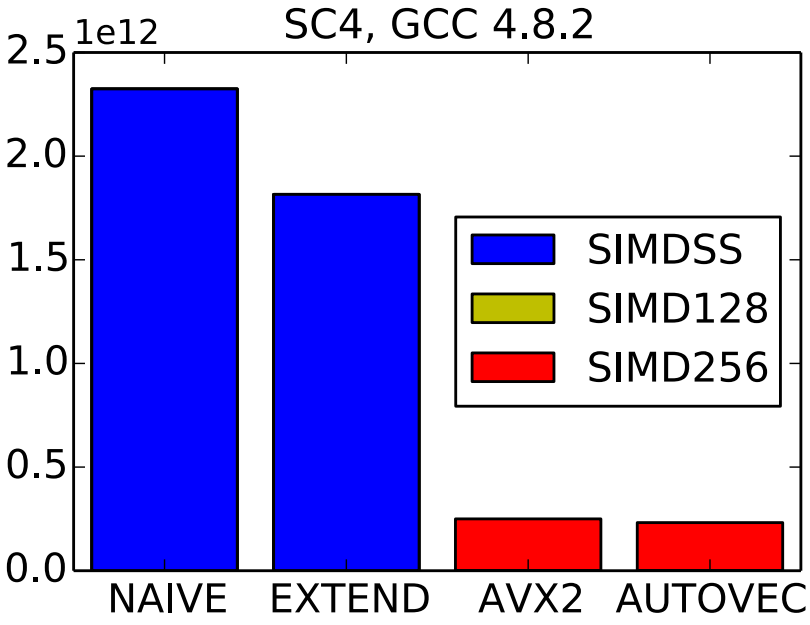
- Problem: $256^3$ single precision floating point arrays, 5001 time steps.
- Test Device: a Xeon E5-2660 Sandy Bridge processor: 8 x 2.2GHz cores.
- Desired SIMD instruction sets: packed AVX.
- Parallelization: OpenMP (outside k loop), 1 thread/core.
- Kernels:
  - NAIVE: with no optimizations.
  - EXTEND: based on NAIVE, with the coefficient loop unrolled manually, coded in the old IWAVE/acd.
  - AVX[`n`]: based on NAIVE, coded with SIMD intrinsic, with i loops manually unrolled `n` times.
  - AUTOVEC: based on NAIVE, with re-arranged code structure and compiler pragmas so that the compiler can auto-vectorize this kernel.

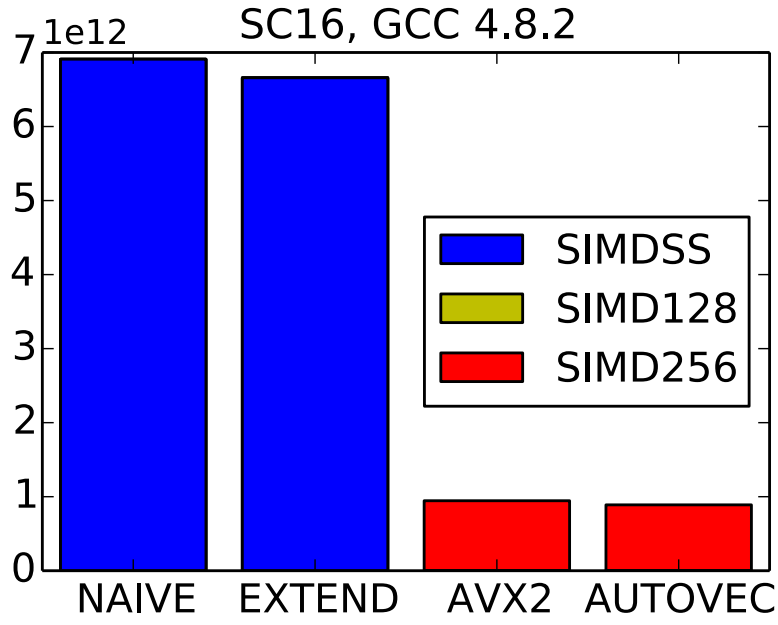
# Auto-vectorizaiton Results

Number of SIMD FLOP Instructions Executed:



# Auto-vectorization Results

Table : Run time results [seconds]. icc=icc 14.0.0, gcc=gcc 4.8.2.

| | SC4 | | SC16 | |
|---|---|---|---|---|
| Kernel | icc | gcc | icc | gcc |
| NAIVE | 138 | 196 | 543 | 940 |
| EXTEND | 36 | 152 | 202 | 838 |
| AVX | 36 | 47 | 198 | 244 |
| AUTOVEC | 36 | 47 | 200 | 241 |

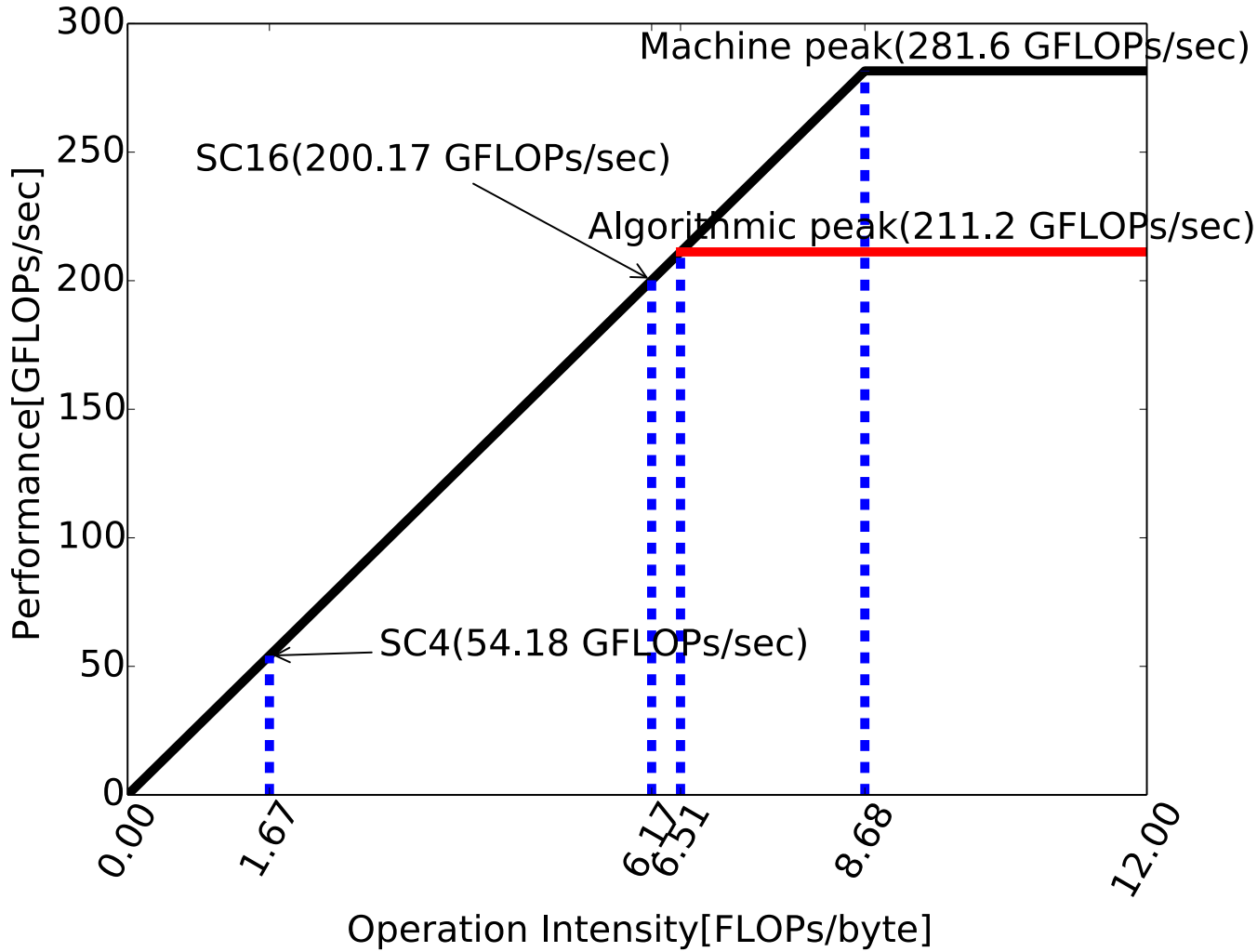
- Current compilers make auto-vectorized codes have comparable performance as intrinsic codes, so writing intrinsics code manually is unnecessary.

# The second part solves the memory problem.

- **Memory Bound**

  Due to the limited memory bandwidth between cache and DRAM, if the application is memory intensive, then the performance of the application is likely to be bounded by the size of the memory bandwidth.
- L3 cache ($\sim$20MB) $<$ Array size (e.g. 134MB in our test), so the entire array will be loaded into cache at least once per time step.
- Cache optimization aims to mitigate the memory bottleneck by reusing the cached data.

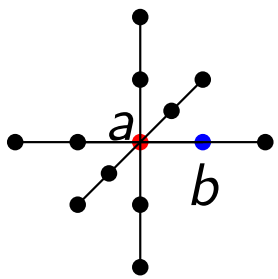
# Roofline model shows the impact of memory bound on the performance of SC.

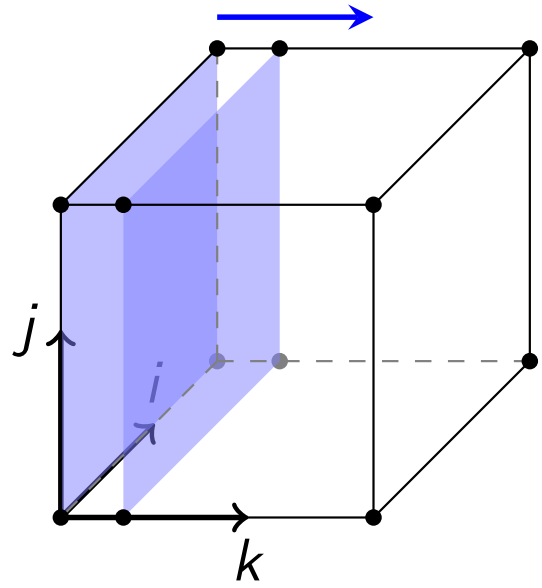



*Operation Intensity* (OI): FLOPs executed per byte transferred.

# Multiple loads per time step for SC16

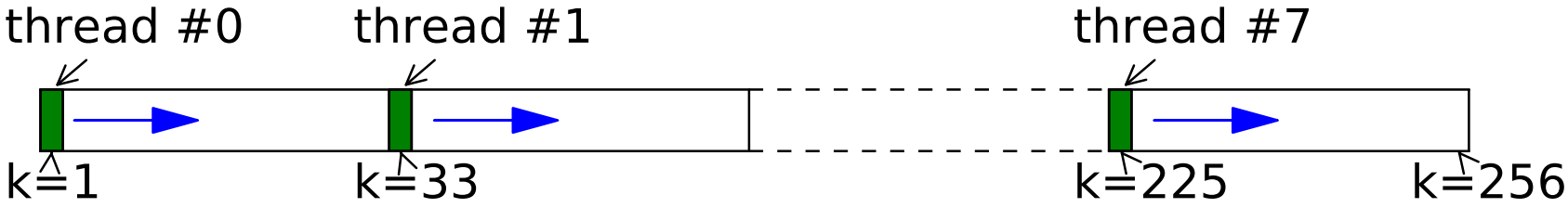
- Before reusing `U_in` at $b$, the thread needs to do [(`NY`-r)`NX`-1] stencil operations. Denote related memory by $mem(SC)$.




- Each thread will process `NZ`/`NT` contiguous `k`-planes.

Figure : OpenMP parallelization without using `schedule(static, 1)`.



# Multiple loads per time step for SC16

- For $256^3$ SC, $mem(SC)$ is $\sim$6 `k`-planes (SC4), $\sim$17 `k`-planes (SC16).
- Xeon E5-2660 has 20MB L3 cache, which can hold at most 8 `k`-planes/core (SC16), 9 `k`-planes/core (SC4).
- In SC16, every time $\mathtt{k}_1$ is treated as a finite difference term along `k` axis, it has to be loaded from DRAM.

| #loads | #stores |
|---|---|
| 17×U_in + 1×U_out | 1×U_out |

Table : Loads and stores per time step for SC16

- *Operation intensity* $\downarrow$ 0.97 FLOPs/byte, Peak GFLOPs/sec $\downarrow$ 31.59 GFLOPs/sec. (200.17 GFLOPs/sec if load/store only once.)

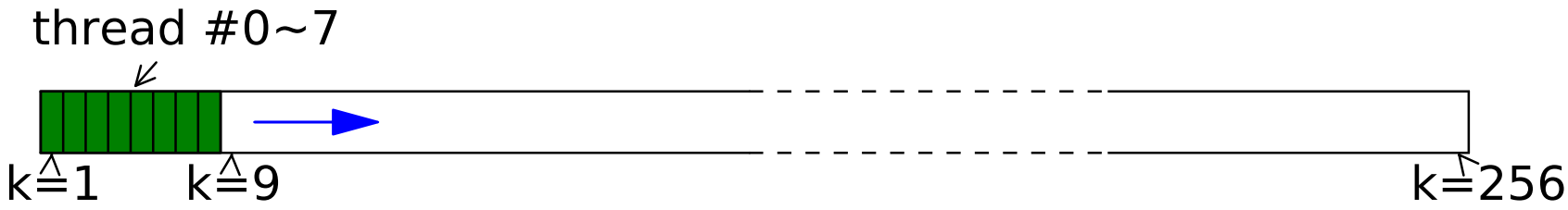
# Cache optimization methods:

| Solutions | reduce multiple loads | blocking in time |
|---|---|---|
| Thread-blocking | ✓ | |
| Separate-and-exchange | ✓ | |
| Parallelized Time-skewing | ✓ | ✓ |

- Blocking in time makes it possible to reuse the same cached data for several time steps.

# #1: Thread-blocking Method

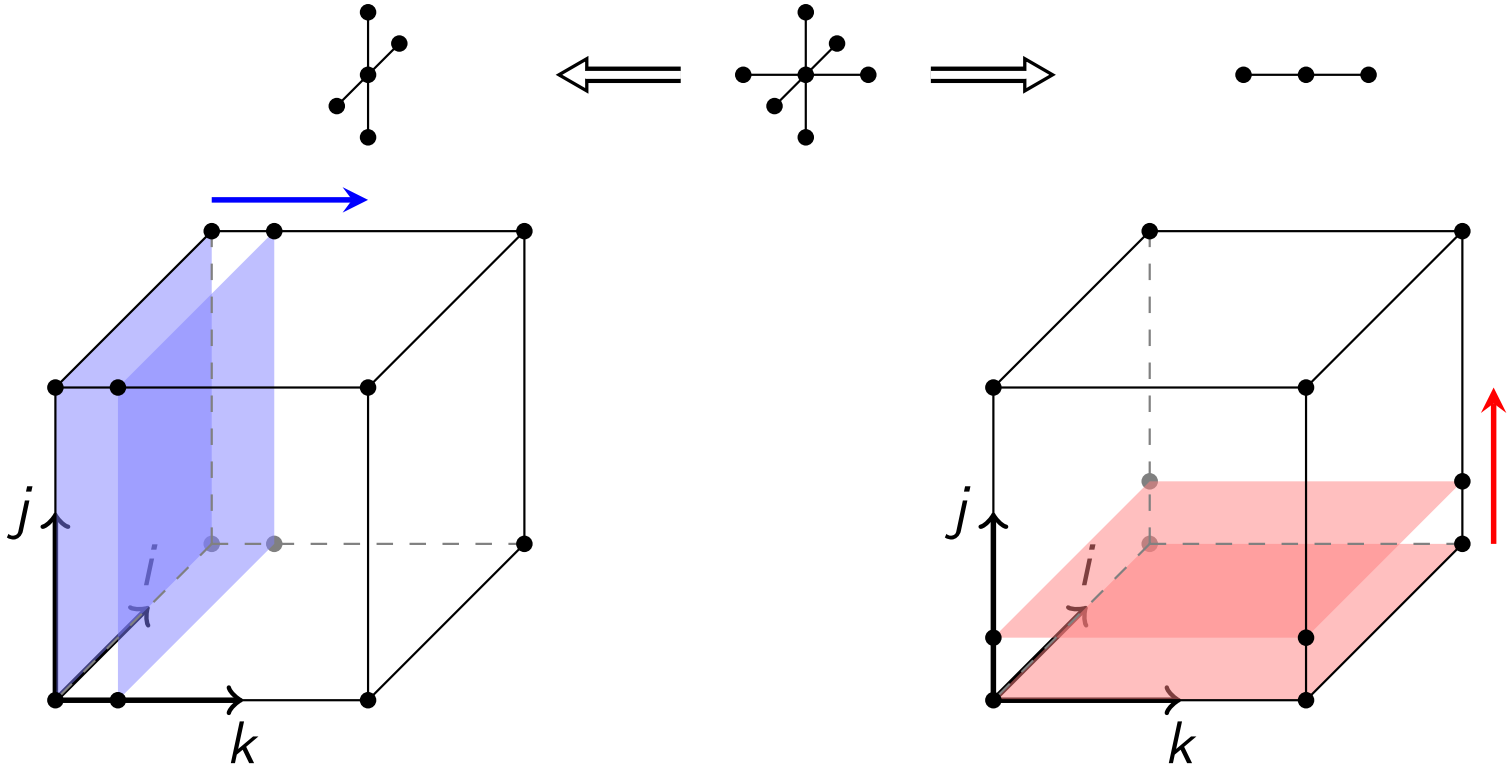
Figure : Using `schedule(static, 1)`.




- Every thread will only get one `k`-plane each round.
- `U_in` data at point $b$ may be re-used by other threads simultaneously (at least $min(\mathtt{NT}, r)$ times).

| #loads | #stores | $lb$(cache) |
|---|---|---|
| $1\times$`U_in` $+ 1\times$`U_out` | $1\times$`U_out` | $(4\mathtt{NT} + 2\mathtt{r})$ `k`-planes |

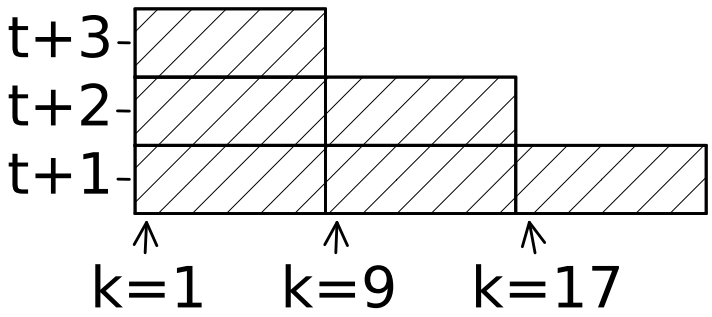
# Sol2: Separate-and-exchange Method [Stork]




| #loads | #stores | *lb*(cache) |
|---|---|---|
| 2×U_in + 2×U_out | 2×U_out | ((4 + 2r) × NX + 4r) × NT |

- Hurt the performance for lower-order stencils.

# Sol3: Parallelized Time-skewing Method =Thread-blocking + time-skewing




- Temporal blocking factor (NTS) is determined by minimizing #loads/stores.

| #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 | |

# Cache Optimization Results

Table : L3 cache misses.

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 2.27e9 | 2.38e9 | 1.23e10 | 1.12e10 |
| +T-b | 2.05e9 | 2.13e9 | 2.73e9 | 2.91e9 |
| +S&E | 4.95e9 | 5.32e9 | 8.09e9 | 5.95e9 |
| +Parallelized TS | 7.17e8 | 7.36e8 | 1.07e9 | 1.15e9 |

Table : Run time results [seconds].

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 36 | 47 | 200 | 241 |
| +T-b | 35 | 41 | 84 | 133 |
| +S&E | 65 | 72 | 97 | 156 |
| +Parallelized TS | 31 | 39 | 82 | 131 |

# Cache Optimization Results

Table : L3 cache misses.

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 2.27e9 | 2.38e9 | 1.23e10 | 1.12e10 |
| +T-b | 2.05e9 | 2.13e9 | 2.73e9 | 2.91e9 |
| +S&E | 4.95e9 | 5.32e9 | 8.09e9 | 5.95e9 |
| +Parallelized TS | 7.17e8 | 7.36e8 | 1.07e9 | 1.15e9 |

Table : Run time results [seconds].

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 36 | 47 | 200 | 241 |
| +T-b | 35 | 41 | 84 | 133 |
| +S&E | 65 | 72 | 97 | 156 |
| +Parallelized TS | 31 | 39 | 82 | 131 |

# Cache Optimization Results

Table : L3 cache misses.

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 2.27e9 | 2.38e9 | 1.23e10 | 1.12e10 |
| +T-b | 2.05e9 | 2.13e9 | 2.73e9 | 2.91e9 |
| +S&E | 4.95e9 | 5.32e9 | 8.09e9 | 5.95e9 |
| +Parallelized TS | 7.17e8 | 7.36e8 | 1.07e9 | 1.15e9 |

Table : Run time results [seconds].

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 36 | 47 | 200 | 241 |
| +T-b | 35 | 41 | 84 | 133 |
| +S&E | 65 | 72 | 97 | 156 |
| +Parallelized TS | 31 | 39 | 82 | 131 |

# Cache Optimization Results

Table : L3 cache misses.

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 2.27e9 | 2.38e9 | 1.23e10 | 1.12e10 |
| +T-b | 2.05e9 | 2.13e9 | 2.73e9 | 2.91e9 |
| +S&E | 4.95e9 | 5.32e9 | 8.09e9 | 5.95e9 |
| +Parallelized TS | 7.17e8 | 7.36e8 | 1.07e9 | 1.15e9 |

Table : Run time results [seconds].

| Kernel | SC4 | | SC16 | |
|---|---|---|---|---|
| | icc | gcc | icc | gcc |
| AUTOVEC | 36 | 47 | 200 | 241 |
| +T-b | 35 | 41 | 84 | 133 |
| +S&E | 65 | 72 | 97 | 156 |
| +Parallelized TS | 31 | 39 | 82 | 131 |

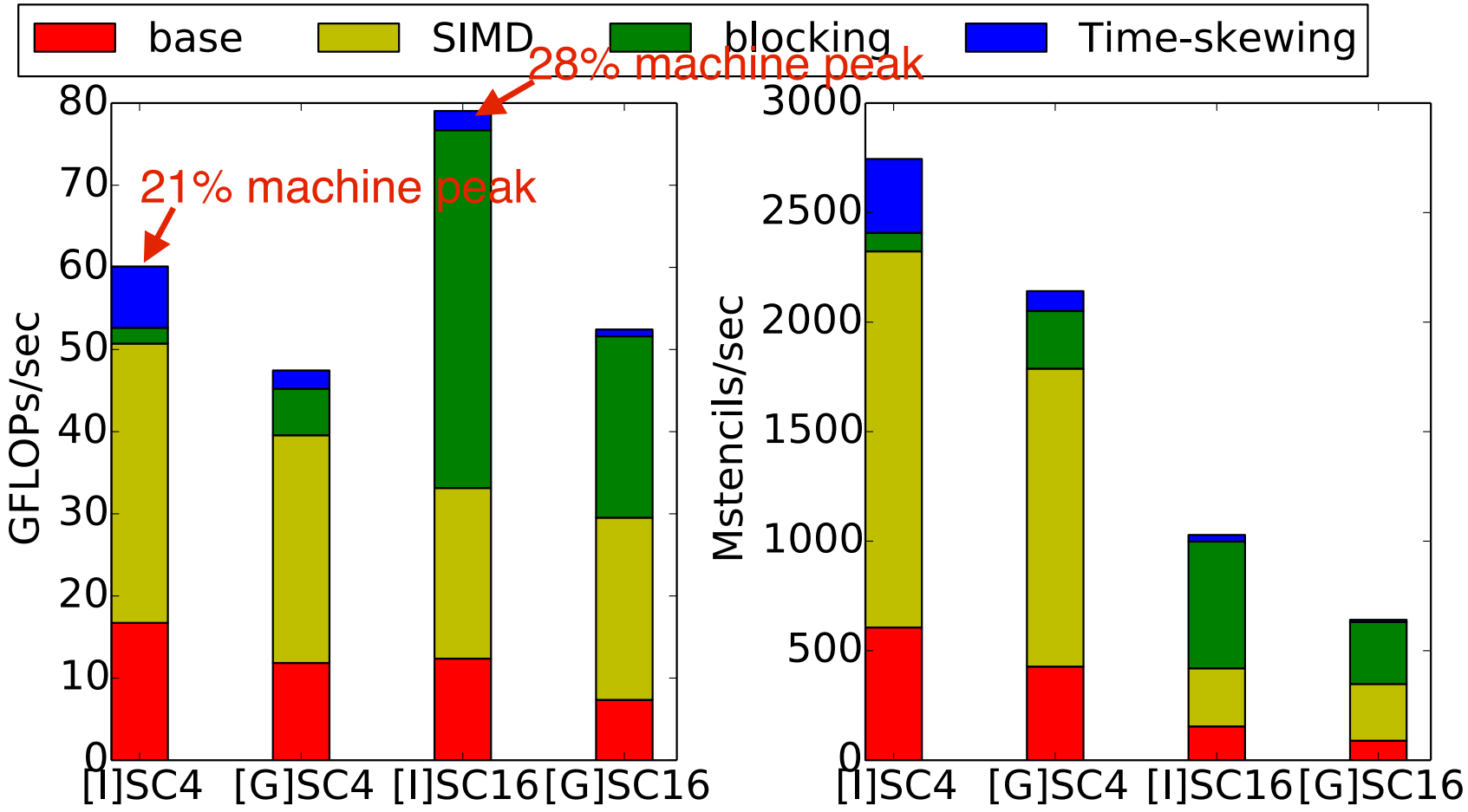
# Evaluation of Two Optimization Approaches

base = NAIVE, SIMD = AUTOVEC

blocking = Thread-blocking

Time-skewing = Parallelized time-skewing

[I] = icc, [G] = gcc.



## Conclusion

- By modifying the code structure, both icc and gcc compilers can provide fully-vectorized stencil code of any order with performance comparable to that of SIMD intrinsic code.
- Both SC4 and SC16 are memory bandwidth bound by analyzing the corresponding roofline model.
- Among the three cache optimizations provided to mitigate the memory issue, separate-and-interchange method works efficiently only for high order stencils (e.g., SC16). Parallelized time-skewing (upgraded from thread-blocking) gives a 1.2x (2x) further speedup for fully-vectorized SC4 (SC16).

# Ongoing PhD work

1. 
   - Identify the performance bottleneck of variable-order staggered-grid anisotropic elastic (TI, crack-induced orthorhombic, or more general types) wave simulation codes.
   - Extend cache optimizations to these modeling codes, and implement them in the context of RTM. The codes will be written directly in IWAVE, with hybrid MPI+OpenMP.
2. Add multi-grid option, extend the idea of conservative energy interpolation to high-order anisotropic elastic codes.

   **The energy-conservative interpolation** [Petersson and Sjögreen, 2010] for ghost points can yield a more accurate, more stable, faster solution than the non-conservative intuitive interpolation when source is near the grid interface.

Thanks!
Q&A

# Backup Slide 1

1. Why vectorization doesn't give you 8x speedup?

   Sandy Bridge can only do 128-bit load and 128-bit store per cycle. With perfect unrolling, SSE and AVX can attain the same performance. The performance is load/store bound.

2. Why GCC is slower than ICC?

   [Stated in the appendix E:] 1. The AVX instruction generated by ICC can directly operates on memory address, GCC needs another load instruction; 2. Once the coefficient variable is loaded into register, ICC can reuse it multiple times among different *i*s, while GCC loads it again and again for different *i*s.

3. The difference between the cache optimizations used in this thesis and ones used in that CUDA paper?

   One impressive feature of that paper is to maximize the register locality, while my work doesn't have this feature. The GPU process the data plane by plane along the contiguous dimension. Loading a next plane means locally loading a front datum into register and throwing out the tail datum out of the register, and some data shuffling among registers.

# Backup Slide 1

1. Why vectorization doesn't give you 8x speedup?

   Sandy Bridge can only do 128-bit load and 128-bit store per cycle. With perfect unrolling, SSE and AVX can attain the same performance. The performance is load/store bound.

2. Why GCC is slower than ICC?

   [Stated in the appendix E:] 1. The AVX instruction generated by ICC can directly operates on memory address, GCC needs another load instruction; 2. Once the coefficient variable is loaded into register, ICC can reuse it multiple times among different *i*s, while GCC loads it again and again for different *i*s.

3. The difference between the cache optimizations used in this thesis and ones used in that CUDA paper?

   One impressive feature of that paper is to maximize the register locality, while my work doesn't have this feature. The GPU process the data plane by plane along the contiguous dimension. Loading a next plane means locally loading a front datum into register and throwing out the tail datum out of the register, and some data shuffling among registers.

# Backup Slide 1

1. Why vectorization doesn't give you 8x speedup?

   Sandy Bridge can only do 128-bit load and 128-bit store per cycle. With perfect unrolling, SSE and AVX can attain the same performance. The performance is load/store bound.

2. Why GCC is slower than ICC?

   [Stated in the appendix E:] 1. The AVX instruction generated by ICC can directly operates on memory address, GCC needs another load instruction; 2. Once the coefficient variable is loaded into register, ICC can reuse it multiple times among different *i*s, while GCC loads it again and again for different *i*s.

3. The difference between the cache optimizations used in this thesis and ones used in that CUDA paper?

   One impressive feature of that paper is to maximize the register locality, while my work doesn't have this feature. The GPU process the data plane by plane along the contiguous dimension. Loading a next plane means locally loading a front datum into register and throwing out the tail datum out of the register, and some data shuffling among registers.

# Backup Slide 2

4. Latest Intel/GCC compiler version?

   icc 14.0 and gcc 4.9.

# Backup Slide 3

Other tricks applied:

- Align memory on 32-byte boundary [Appendix E]
  - reduce generated instructions per statement
  - increase unrolling factors
  - employ cache by-passing stores when using intrinsics.

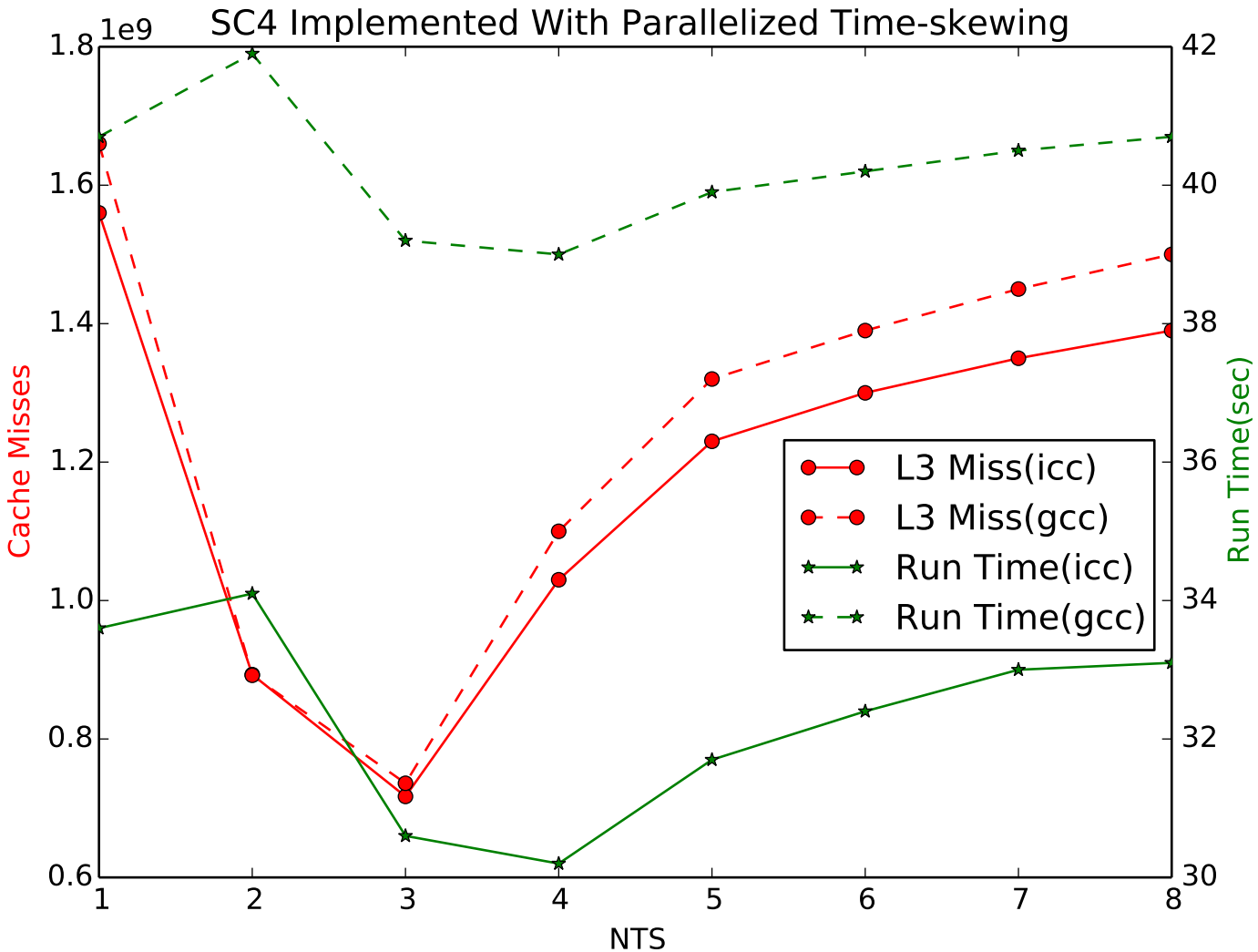
  But my experiments showed that there was no apparent run time reduction after alignment. 1. Pad the array to obtain alignment
  2. Notify the compiler about the alignment (e.g., `__assume_aligned()`)
- Compiler options:

  `-funroll-loops`: increase unrolling factors.

  `-march=native` (gcc only): using desired length of SIMD registers.

# B4: Sol3: Parallelized Time-skewing Method

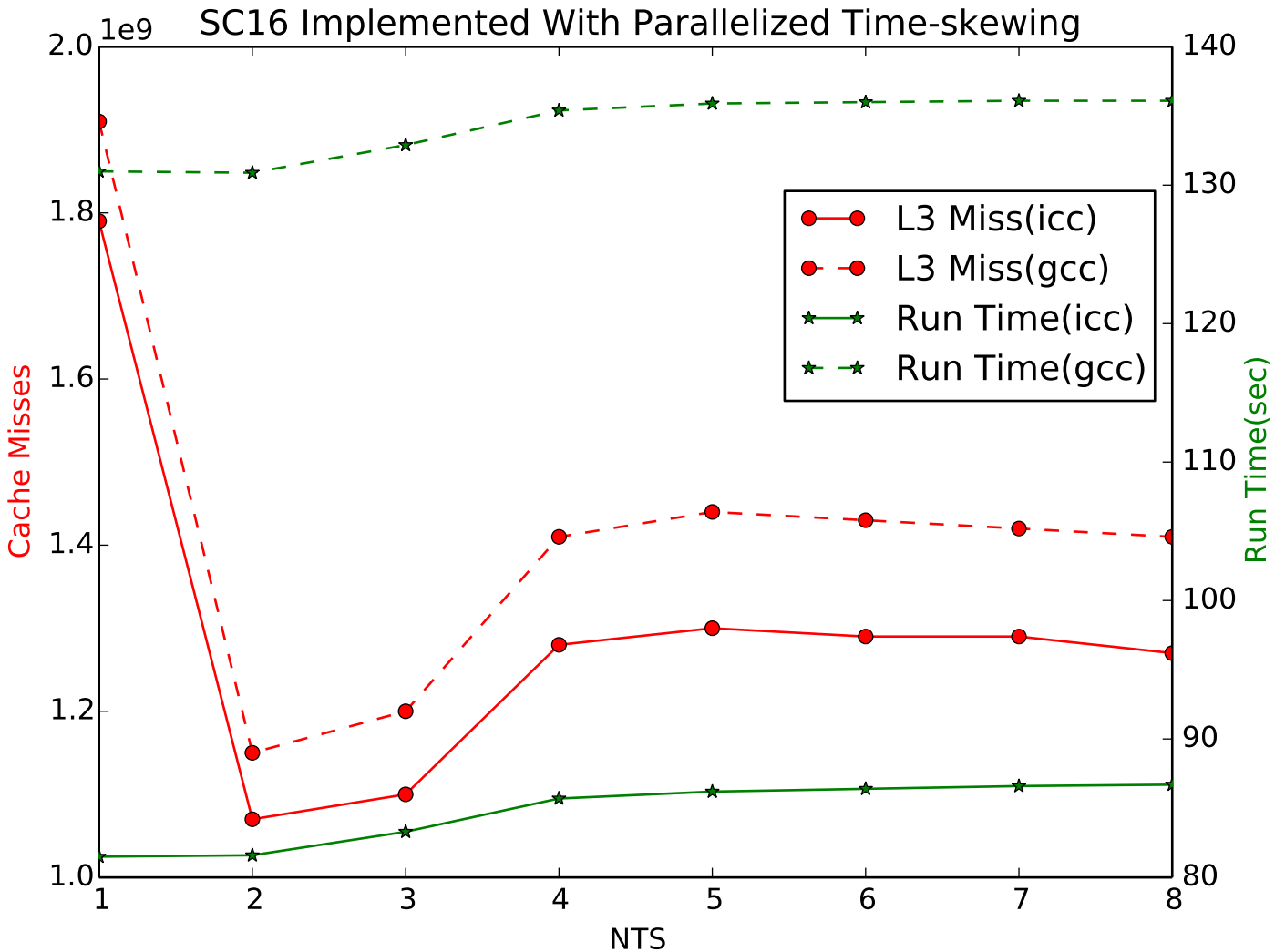
For SC4, $NTS_{opt}$=3.




When NTS=1, the code degrades to the thread-blocking code.

# B5: Sol3: Parallelized Time-skewing Method

For SC16, $NTS_{opt}$=2.




When NTS=1, the code degrades to the thread-blocking code.

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. (2009).

*Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*.

PhD thesis, University of California at Berkeley.

Dursun, H., Kunaseth, M., Nomura, K., Chame, J., Lucas, R., Chen, C., Hall, M., Kalia, R., Nakano, A., and Vashishta, P. (2012).

Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters.

In *J Supercomput*, number 2, pages 946–966.

Dursun, H., Nomura, K., Wang, W., Kunaseth, M., Peng, L., Seymour, R., Kalia, R., Nakano, A., and Vashishta, P. (2009).

In-core optimization of high-order stencil computations.

In *In PDPTA*, pages 533–538.


Henretty, T., Stock, K., Pouchet, L., Franchetti, F., Ramanujam, J., and Sadayappan, P. (2011).

Data layout transformation for stencil computations on short-vector simd architectures.

In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245. Springer-Verlag.


Intel (2012).

Using avx without writing avx.

available online at `http://software.intel.com/en-us/articles/using-avx-without-writing-avx-code`.


Petersson, N. A. and Sjögreen, B. (2010).

Stable grid refinement and singular source discretization for seismic wave simulations.

*Communications in Computational Physics*, 8:1074–1110.


Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H. (2011).

Cache accurate time skewing in iterative stencil computations.

In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 571–581. IEEE Computer Society.

Zumbusch, G. (2012).

Tuning a finite difference computation for parallel vector processors.

In *ISPDC*, pages 63–70.

Zumbusch, G. (2013).

Vectorized higher order finite difference kernels.

In *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 343–357. Springer Berlin Heidelberg.