Wave Equation Stencil Optimization on A Multi-core CPU

Muhong Zhou

TRIP 2014

May 6, 2014

Engine of Seismic Imaging – FDTD Wave Propagation Kernel

Wave propagation traveling in fluid media is governed by the following equation:

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} = \kappa(x,y,z)\nabla \cdot (\frac{1}{\rho(x,y,z)}\nabla u(x,y,z,t))$$

- $(x,y,z) \in \Omega$, a 3D computational domain.
- κ: bulk modulus field; ρ: density field.
- u is the pressure field.

Acoustic Constant Density Wave Equation

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

- $c^2(x,y,z) = \kappa(x,y,z)/\rho(x,y,z)$, squared velocity field.
- This work further assumes c is constant.

CFD Discretization

Discretize each derivative with *Central Finite Difference Scheme* [Moczo et al., 2007]:
2-nd order CFD on u_{tt}

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

4-th order CFD on u_{xx}

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

The stencil kernels evaluated in this work use 2-4 and 2-16 CFD scheme, i.e., r (stencil radius) = 2, 8. Corresponding stencil schemes (SC) are denoted as SC4, SC16.

NAIVE Stencil Kernel

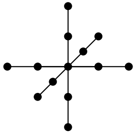
- U_out: pressure field values at previous and next time step, i.e., $u(x, y, z, t - \Delta t)$ and $u(x, y, z, t + \Delta t)$.
- U_in: pressure field values at current time step, i.e., $u(x, y, z, t)$.

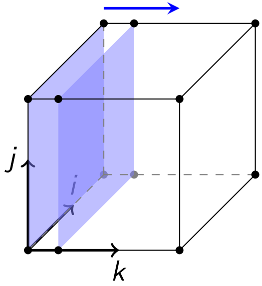
```
for (t = 1; t <= NT; t++)
 for (k = 1; k <= NZ; k++)
  for (j = 1; j <= NY; j++)
   for (i = 1; i <= NX; i++){
    U_out[k][j][i]=c0*U_in[k][j][i]-U_out[k][j][i
        ];
    for(ixyz = 1; ixyz <= r; ixyz++)
     U_out[k][j][i]+=cx[ixyz]*(U_in[k][j][ixyz+i]+
         U_in[k][j][-ixyz+i]);
    for(ixyz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

Listing 1: NAIVE – stencil kernel with no optimization.

Stencil and Traversing Pattern Inside the Computation Domain

3D Stencil Operator (2-4 scheme)





- Each U_out[k][j][i] (center of the stencil) is updated by using itself and the nearby U_in values.
- Stencil operator sweeps over the entire computational domain according to certain order and updates every U_out[k][j][i] over one time step.

Optimization Overview

Based on wave equation stencil kernels, I apply:

- CPU SIMD vectorization techniques.
- CPU cache optimization techniques (in the context of OpenMP parallelization).

SIMD Technology

- SIMD = Single Instruction, Multiple Data.
- SIMD technology on modern x86-based (AMD, Intel) CPUs
  1. vector registers: 128-bit `XMM`, 256-bit `YMM`
  2. SIMD instruction sets: SSE, AVX (Sandy Bridge, 2011)
- Scalar SIMD instructions vs. Packed SIMD instructions.

  They have same latency and throughput.

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

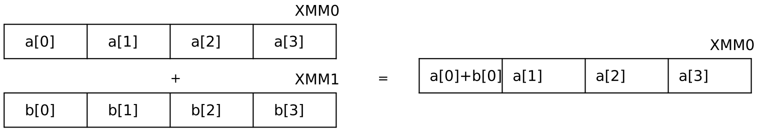
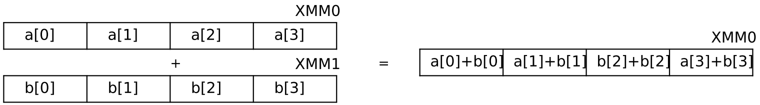


Figure : Packed SSE instruction `ADDPS` performing vector addition.



SIMD Technology

- How to use it on my stencil kernel?
- SIMD load/store preference: contiguous references will render less load/store instructions.

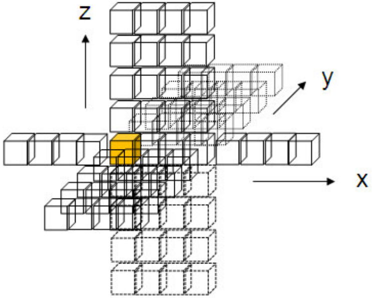


Figure : Performing stencil operations using SSE. [Borges, 2011]

- Basic principle: group several stencils along unit-stride dimension and then SIMDize them.

SIMD Technology

- Implementation – three approaches: (Descending difficulty)
  1. Assembly languages
  2. SIMD intrinsics.
     ```
     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.

Auto-vectorization By Compilers

Common violations prevent compiler auto-vectorization: [Intel, 2011]

- Not innermost loop
- Low loop count
- Not unit-stride accessing
- Existence of potential data dependencies

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

- low loop count
- 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 -vec-report or -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 2: AUTOVEC

Auto-vectorization By Compilers

Other tricks applied:

- Align memory on 32-byte boundary:
  - reduce generated instructions per statement
  - increase unrolling factors
  - employ cache by-passing stores when using instrinsics
- Compiler options:

  `-funroll-loops`: increase unrolling factors.

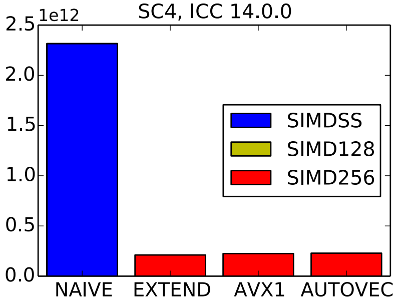
  `-march=native` (gcc only): using desired length of SIMD registers.

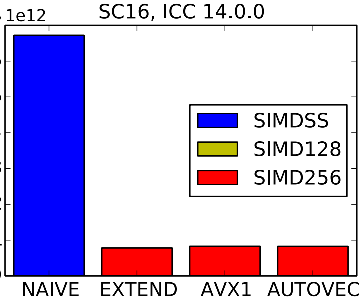
Auto-vectorization Results

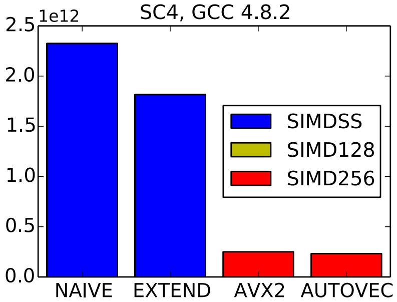
- Problem: 256^3 single precision floating point arrays, 5001 time steps.
- Test Device: a Xeon E5-2660 Sandy Bridge processor: 8 x 2.2GHz cores.
- Desired SIMD instruction sets: AVX.
- Parallelization: OpenMP (outside k loops), 1 thread/core.
- Kernels:
  - NAIVE: with no optimizations.
  - EXTEND: explicitly unroll the coefficient loops.
  - AVX*n*: SIMD intrinsic codes, with i loops unrolled *n* times.
  - AUTOVEC: Auto-vectorized codes by the compiler.

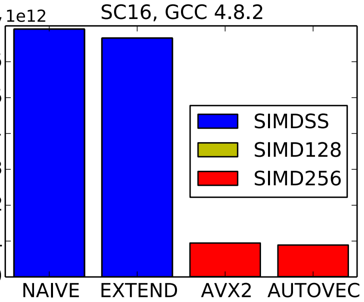
Auto-vectorizaiton Results

Number of SIMD FLOP Instructions Executed:









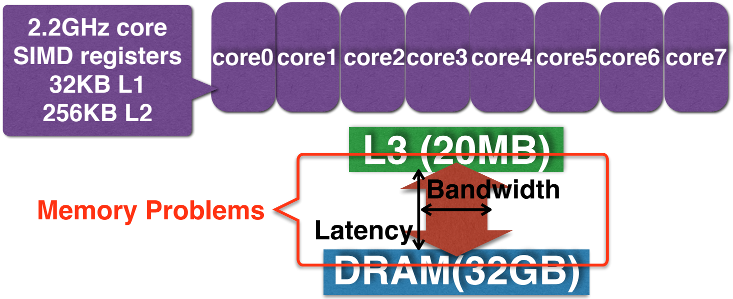
Auto-vectorization Results

Table : Run time results. [I]Time, [G]Time are the run time when the kernel is compiled by icc 14.0.0, gcc 4.8.2 respectively.

	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

- icc fails to vectorize NAIVE, gcc fails to vectorize NAIVE, EXTEND.
- Both intrinsic codes and AUTOVEC codes are fully vectorized.
- AUTOVEC achieves comparable performance as intrinsic codes.

Memory Bottleneck



Problem size: 256x256x256 SPFP arrays ∼ 134MB

Hardware deficiencies:

1. Slow DRAM to cache memory speed
2. Limited DRAM to cache memory bandwidth

Table : Yearly growth rate. [Demmel, 2014]

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

Both SCs are Memory Bandwidth Bound on Xeon E5-2660

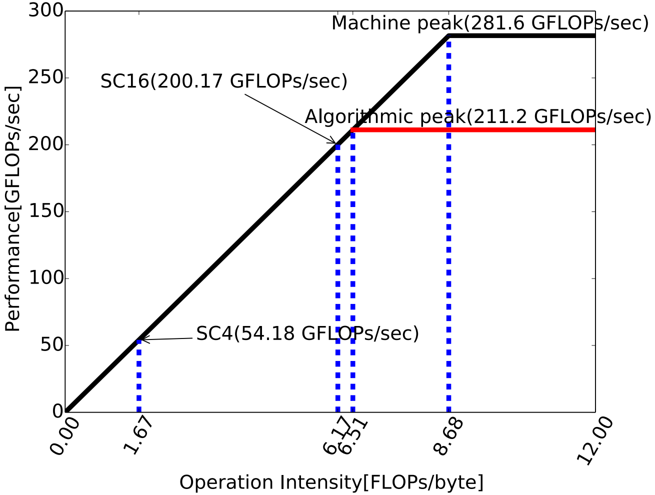
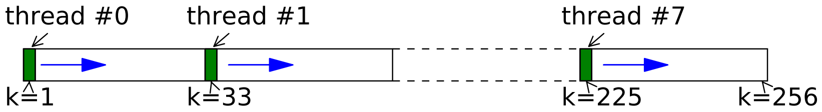


Figure : Roofline Model [Williams et al., 2009] based on Xeon E5-2660

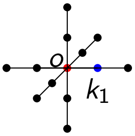
Operation Intensity (OI): FLOPs executed per byte transferred.

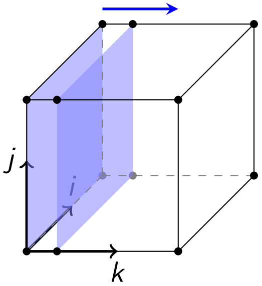
Multiple Loads Per Time Step

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



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





- Before reusing `U_in` at k_1, the thread needs to do [(`NY-r`)`NX-1`] stencil operations. Denote related memory by $mem(SC)$.

Multiple Loads Per Time Step

- 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).

#loads	#stores
17×`U_in` + 1×`U_out`	1×`U_out`

Table : Loads and stores per time step for SC16

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

Reduce Memory Traffic between Cache and DRAM

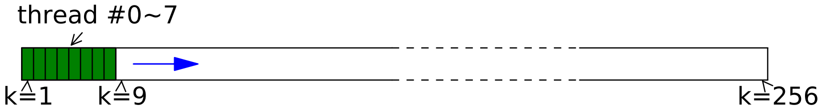
Solutions:

1. Thread-blocking Method
2. Separate-and-exchange Method [Stork, personal communication, 2013]
3. Parallelized Time-skewing Method

- #1, #2 prevent multi-loads per time step, i.e., prevent *operation intensity* from left shifting.
- #3 involves temporal-blocking, to load/store only once over several time steps, i.e., right shift *operation intensity*, break the bandwidth bottleneck.

#1: Thread-blocking Method

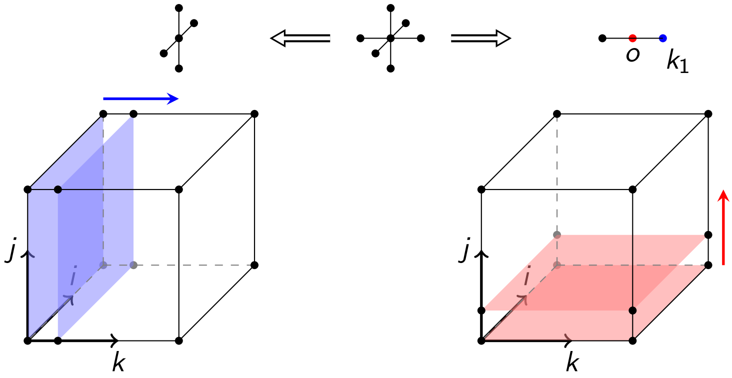
Figure : Using `schedule(static, 1)`.



- Every thread will only get one `k`-plane each round.
- `U_in` data at point k_1 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

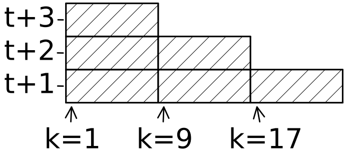
Sol2: Separate-and-exchange Method [Stork]



- Only takes NX-1-r stencil operations until the thread reuses `U_in` at point k_1.

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

Sol3: Parallelized Time-skewing Method



- Conventional time-skewing method + thread-blocking method: set spatial blocking factor as `NT`.
- 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

- AUTOVEComp: Thread-blocking Method
- AUTOVECz: Separate-and-exchange Method
- AUTOVECts: Parallelized Time-skewing Method (NTS=3 for SC4, NTS=2 for SC16).

Table : L3 cache misses.

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

Cache Optimization Results

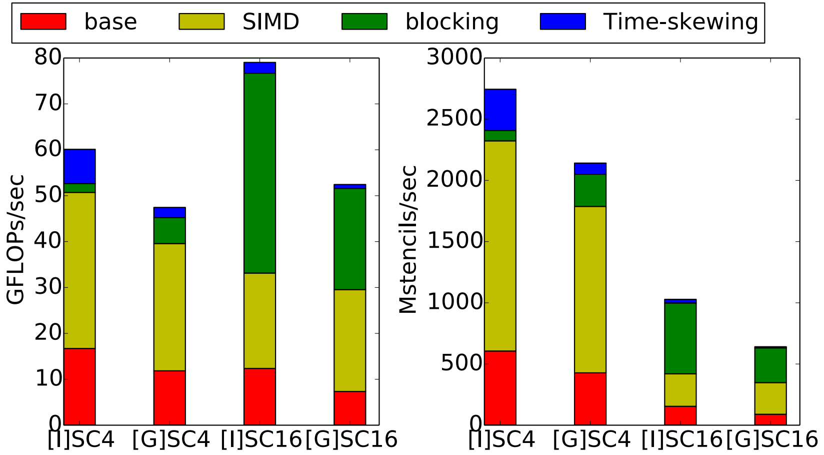
- AUTOVECOmp: Thread-blocking Method
- AUTOVECz: Separate-and-exchange Method
- AUTOVECts: Parallelized Time-skewing Method

Table : Run time results.

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

Evaluation of Two Optimization Approaches

base = NAIVE
SIMD = AUTOVEC
blocking = Thread-blocking
Time-skewing = Parallelized time-skewing



Summary and Future Work

Summary:

- Fully vectorize stencil kernel of any order, and obtain performance comparable to intrinsics codes.
- Propose two OpenMP parallelized cache optimization methods that further increases stencil kernel performances by 1.2X to 2X.
- The final optimized kernel are both icc and gcc friendly.

Future Work:

- Increase problem domain and extend to multi-sockets.
- Experiments with IWAVE.

Thanks!
Q&A

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.

Demmel, J. (2014).

Case study: Tuning matrix multiply.

available online at
http://www.cs.berkeley.edu/~demmel/cs267_Spr14/.

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.

Intel (2011).

Using avx without writing avx.

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

McVoy, L. and Staelin, C. (1996).
lmbench: Portable tools for performance analysis.
In *Proceedings of USENIX 1996 Annual Technical Conference*, San Diego, California.

Moczo, P., Robertsson, J. O., and Eisner, L. (2007).
The finite-difference time-domain method for modeling of seismic wave propagation.
In *Advances in wave propagation in heterogeneous Earth*, volume 48 of *Advances in Geophysics*, pages 421–516. Elsevier - Academic Press, London, UK.

Williams, S., Waterman, A., and Patterson, D. (2009).
Roofline: An insightful visual performance model for multicore architectures.
Communications of the ACM, 52(4):65–76.