Optimizing a Seismic Imaging Code on Intel Xeon Phi

Interesting Insights on How to Optimize for Cache on Xeon vs Xeon Phi

G Civario\textsuperscript{1}, S Delaney\textsuperscript{2}, M Lysaght\textsuperscript{1}

\textsuperscript{1}Irish Centre for High End Computing (ICHEC)
\textsuperscript{2}Tullow Oil plc
Ireland
Background

- **TORTIA: A Reverse Time Migration (RTM) code:**
  - Developed in-house at Tullow Oil plc (Oil & Gas exploration)
  - An explicit FD scheme of variable spatial order, and first order in time, to model wave propagation in the three isotropy cases
  - Based on an unconventional rotated staggered grid (RSG) method [1]
  - We describe cache-blocking methods that provide interesting insights on Xeon vs Xeon Phi optimizations

- **Verticals/ Domains:**
  - Energy, Geophysics

- **Modes of Execution:**
  - This in an MPI + OpenMP code (with SIMD directives too)
  - Use a small pure OpenMP benchmark code for the investigation

- **Tools Used:**
  - C/C++, OpenMP 4.0, MPI, Intel Vector Advisor, VTune, Roofline Model

---

TORTIA Kernel

• Each grid point (vertex) holds 1 vector quantity and 1 scalar quantity.
• Use the scalars to update the vectors, and vice versa, iteratively.
• To update the 'pivot' (target vertex), stencil specifies relative positions of vertices that will contribute.

```c
#pragma omp for
for (int i=0; i<Ni; i++)
  for (int j=0; j<Nj; j++) {
    #pragma omp simd
    for (int k=0; k<Nk, k++) {
      // Load stencil elements
      #pragma unroll
      for(int n=0; n<Nstencil; n++)
        stencil[n] = s[i][j][k + offsets[n]];
      // FD computation
      calcUpdate(stencil); // inline vector function
      v1[i][j][k] += A * stencil[0] + B * stencil[1];
      v2[i][j][k] += C * stencil[2] + C * stencil[3];
    }
  }
```

• The stencil is incomplete for pivots near domain extremities.
• Additional external vertex values are required. This arises in:
  1. Physical boundary conditions at the edge of the grid.
  2. 'Halo' zones around MPI sub-domains.
  4. Overlap regions of adjacent cache blocks.
Theoretical data reuse

- As iteration proceeds along the fastest varying dimension the stencil sweeps out an extruded ‘X’ shape through the grid.
- Vertices contribute twice as the stencil sweeps through k.
- This delivers 1 cache reuse per vertex.

- As iteration proceeds in the next dimension we can imagine the extruded stencil in figure 2 sweeping to the right.
- Again, vertices are hit twice as we sweep through j.
- The data volume processed between these 2 hits is O(Nk).
- Cache reuse may occur if Nk is small enough not to evict data too soon.

- Finally, we sweep through the last dimension.
  - Again there are 2 hits per vertex, separated by O(Nk*Nj) other updates.
  - This is typically too much data to fit in cache.
  - Tackling the grid in ‘blocks’, we can limit the amount of data processed between reuse opportunities, effectively reducing Nk and Nj.
  - Blocking in both the k and j dimensions could yield benefits.
  - Blocking in k reduces the unit-stride sweep, reducing performance.
  - Modified prefetching could possibly alleviate this.
  - Blocking in ‘j’ should yield performance benefits.
Boundary layers

- Outer layers (halo) of the block facilitate consistency at block boundaries.
  - These halos are accessed sub-optimally.
  - Hence, the surface to volume ratio of the cache block must be minimized.
- This would motivate cubic blocks filling LLC.
- However, this can disrupt other optimizations (e.g. long ‘k’ sweep).

- Shared caches outperform exclusive caches (of same aggregate size), by avoiding data replication and associated bandwidth overhead.
- Threads may reuse data loaded by others, but contend for bandwidth.
- For optimal reuse, regardless of sharing, we must always aim to maximise the locality of a limited data pool. This means blocking.
- Multi-threading over i and j is somewhat analogous to blocking, by starting new iterations of outer loops before all earlier iterations complete (and evict data).
- Hence, multi-threading with a shared cache has similarities with blocking in an exclusive cache.
Inter-thread data reuse

General case: \( N \) threads, stencil diameter \( D \).
Total uses = \( 2 \times N \times D \)
Total loads = \( N + D - 1 \)
Optimal usage: \( 2 \times D \) uses per load
Efficiency = \( \frac{\text{Total uses}}{\text{Total loads}} / \frac{\text{Optimal usage}}{} \)
\[ = \frac{N}{N + D - 1} \]

- Efficiency improves asymptotically with increasing \( N \).
- More threads with shared cache gives better performance.
- Inter-thread synchronised coherent locality replaces intra-thread spatial locality and spares cache space.
- Correct OpenMP scheduling is the key.
Roofline models

- Red and green cache “walls” enclose the domain and the roof covers it
- Algorithm still memory-bound, even at the right-wall level (i.e., with maximum effective use of cache)
- Actual performance proves effective reuse of data
- Best performance is without tiling in I (thread synchronised locality) and K (vectorisation)
- **Xeon**
  - 55% of (infinite cache) peak performance ➔ not much more to gain
- **Xeon Phi**
  - Extra tiling in J yields some more performance (same tiling doesn’t affect the CPU performance)
  - 24% of (infinite cache) peak performance
  - 77% of equivalent 4x data reuse (ie 2 dimensions out of 3 fully cached)
  - Current limit is L2-L2 cache traffic ➔ maybe some hope here
Conclusion

- We used an uncommon method to maximize data reuse, based on synchronised locality across threads and shared cache
  - OpenMP scheduling central to the method: “dynamic” or “static,1” does the trick

- Extremely effective on Xeon, thanks to L3 shared cache

- Somewhat effective on Xeon Phi, due to per-core L2 private caches
  - 4 threads per core share the L2 cache
  - A lot of L2-L2 data transfers across cores
  - Same impact of this traffic as would have had memory to L2 data transfers

- Overall relative performance increase due to memory optimisations here is significant
  - 1.22x on Xeon
  - 1.46x on Xeon Phi

- Exact same code base for both architectures
  - One single fully standard / portable source code

- Opens up the possibility of genuine production accelerated code on KNL
Supplementary material
Memory-related Workshop
# Memory Latency on KNC

**Elapsed Time:** 9.411s

- **Clockticks:** 1,068,695,000,000
- **Instructions Retired:** 172,120,000,000
- **CPI Rate:** 6.209

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

**Cache Usage:**

- **L1 Misses:** 8,849,950,000
- **L1 Hit Ratio:** 0.987

The L1 cache hit ratio should be as close to 1 as possible. A low value for this ratio may mean that the application does not use the cache effectively.

**Estimated Latency Impact:** 97.021

---

## General Exploration

**General Exploration viewpoint (change)**

- **Analysis Target:**
- **Analysis Type:**
- **Collection Log:**
- **Summary:**
- **Bottom-up**
- **Top-down Tree**
- **Tasks and Frames**

**Grouping:** Function / Call Stack

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>Clockticks</th>
<th>Instructions Retired</th>
<th>CPI Rate</th>
<th>Cache Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L1 Misses</td>
</tr>
<tr>
<td>updateVelocityACOUSTICCompSampParallel@157</td>
<td>267,925,000,000</td>
<td>36,480,000,000</td>
<td>7.344</td>
<td>1,976,900,000</td>
</tr>
<tr>
<td>updateStressACOUSTICLoop&lt;int&gt;1&gt;</td>
<td>175,500,000,000</td>
<td>30,510,000,000</td>
<td>5.752</td>
<td>1,883,900,000</td>
</tr>
<tr>
<td>updateStressACOUSTICLoop&lt;int&gt;0&gt;</td>
<td>175,375,000,000</td>
<td>29,955,000,000</td>
<td>5.895</td>
<td>1,895,375,000</td>
</tr>
<tr>
<td>updateStressACOUSTICLoop&lt;int&gt;2&gt;</td>
<td>171,595,000,000</td>
<td>29,235,000,000</td>
<td>5.857</td>
<td>1,850,275,000</td>
</tr>
<tr>
<td>computeF&lt;%int&gt;3&gt;</td>
<td>23,220,000,000</td>
<td>1,905,000,000</td>
<td>12.178</td>
<td>313,500,000</td>
</tr>
<tr>
<td>computeF&lt;%int&gt;1&gt;</td>
<td>14,335,000,000</td>
<td>1,235,000,000</td>
<td>11.607</td>
<td>275,000,000</td>
</tr>
<tr>
<td>computeF&lt;%int&gt;0&gt;</td>
<td>14,190,000,000</td>
<td>1,635,000,000</td>
<td>8.679</td>
<td>290,500,000</td>
</tr>
<tr>
<td>computeF&lt;%int&gt;2&gt;</td>
<td>13,785,000,000</td>
<td>1,690,000,000</td>
<td>8.457</td>
<td>273,025,000</td>
</tr>
<tr>
<td>[import thunk__kmpc_for Static&lt;%int&gt;4]</td>
<td>30,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[import thunk__kmpc_for Static&lt;%int&gt;]</td>
<td>15,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[import thunk__kmpc_barrier]</td>
<td>10,000,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Memory Bandwidth on KNC
# Memory Latency on IB

## Unfilled Pipeline Slots (Stalls):

### Back-End Bound: 0.748

Identify stalls where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.

### Memory Bound: 0.375

This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of cycles where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline.

### L1 Bound: 0.282

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

- DTLB Overhead: 0.039
- Loads Blocked by Store Forwarding: 0.000
- Split Loads: 0.041
- 4K Aliasing: 0.024

### L3 Bound: 0.121

This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 miss/L3 hit) improves the latency and increases performance.

- Contested Accesses: 0.000
- Data Sharing: 0.000
- L3 Latency: 0.017

### DRAM Bound: 0.028

- Memory Bandwidth: 0.415
- Memory Latency: 0.515
- Local DRAM: 0.004
- Remote DRAM: 0.000
- Remote Cache: 0.000

### Store Bound: 0.000

**Core Bound: 0.335**

This metric shows how core non-memory issues limit the performance when you run out of OOO resources or are saturating certain execution units (for example, using FP-chained long-latency arithmetic operations).

### Port Utilization: 0.335

This metric represents a fraction of cycles during which an application was stalled due to Core non-die-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports.

- Cycles of 0 Ports Utilized: 0.379
- The number of cycles during which no port was utilized.
- Cycles of 1 Port Utilized: 0.309
- Cycles of 2 Ports Utilized: 0.187
- Cycles of 3+ Ports Utilized: 0.069

### Front-End Bound: 0.011
Memory Bandwidth on IB
Vectorisation-related Workshop
Vectorisation rationales

- FD kernels take all the time that isn’t IO or MPI
- Initial code not satisfactory
  - FD order hard-coded
    - We needed arbitrary FD order (run-time input from user)
  - Not vectorisation friendly
    - Auto-vectorisation wasn’t working at all
    - Adding OpenMP SIMD pragmas had only a very limited impact
- Kernels fully re-written
  - Arbitrary FD order
  - Fully vectorisable
    - Aligned memory whenever possible
    - OpenMP pragma SIMD
    - But even without that, the Intel compiler’s auto-vectoriser does the right thing
## Vectorisation metrics on Xeon

<table>
<thead>
<tr>
<th>Xeon</th>
<th>Total vectorisation gain</th>
<th>Total vectorisation efficiency</th>
<th>Number of vectorised kernel</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original workload baseline with disabled auto-vectorisation</td>
<td>1.00x</td>
<td></td>
<td>0</td>
<td>20010</td>
</tr>
<tr>
<td>Original workload baseline</td>
<td>1.00x</td>
<td>0.00%</td>
<td>0</td>
<td>20010</td>
</tr>
<tr>
<td>Simple manual vectorisation</td>
<td>1.55x</td>
<td>19.43%</td>
<td>4</td>
<td>12870</td>
</tr>
<tr>
<td>Code refactored with vectorisation disabled</td>
<td>1.00x</td>
<td></td>
<td>0</td>
<td>11491</td>
</tr>
<tr>
<td>Code refactored with vectorisation enabled</td>
<td>3.63x</td>
<td>45.38%</td>
<td>2</td>
<td>3165</td>
</tr>
</tbody>
</table>
## Vectorisation metrics on Xeon Phi

<table>
<thead>
<tr>
<th>Xeon Phi</th>
<th>Total vectorisation gain</th>
<th>Total vectorisation efficiency</th>
<th>Number of vectorised kernel</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original workload baseline with disabled auto-vectorisation</td>
<td>1.00x</td>
<td>0.00%</td>
<td>0</td>
<td>37263</td>
</tr>
<tr>
<td>Original workload baseline</td>
<td>1.00x</td>
<td>0.00%</td>
<td>0</td>
<td>37263</td>
</tr>
<tr>
<td>Simple manual vectorisation</td>
<td>4.61x</td>
<td>28.79%</td>
<td>4</td>
<td>8090</td>
</tr>
<tr>
<td>Code refactored with vectorisation disabled</td>
<td>1.00x</td>
<td>0.00%</td>
<td>0</td>
<td>37514</td>
</tr>
<tr>
<td>Code refactored with vectorisation enabled</td>
<td>13.84x</td>
<td>86.49%</td>
<td>2</td>
<td>2711</td>
</tr>
</tbody>
</table>
Optimisation insights

- Code portability and standard-compliance are deemed essential
  - Only use of OpenMP parallelisation and SIMD pragmas
  - No proprietary intrinsics or language extensions
- Portable optimisations are important too
  - Ok for pre-processor macros targeted for specific hardware

```c
#if defined(__MIC__) 
    #define VLENGTH 64 
#elif defined(__AVX__) 
    #define VLENGTH 32 
#else 
    #define VLENGTH 16 
#endif 
```
Optimisation insights

- Use of vectorisation tools
  - Auto-vectoriser works very well indeed
    - But vectorisation reports weren’t very explicit with old compilers
    - So we had no clear view on our level of vectorisation
  - Also vectorisation effectiveness is limited on Xeon
    - We tired hard to push it further
    - Vector advisor showed that vectorisation was actually as good as could be
    - The effectiveness is limited by something else, and that’s where we should focus our efforts

- Remaining bottlenecks
  - Not vectorisation-related
  - Possibly related to TLB and L2-L2 cache transfers on Xeon Phi
  - Although current vectorisation is perfectly fine, changing the vectorisation strategy might help (on-going study)
Memory alignment
How to align memory

- Code portability and standard-compliance are deemed essential
  - No proprietary intrinsics or language extensions
- With our data layout, what is important is that the last dimensions of our arrays are aligned to the end of the halo layer
  - We want “a[i][j][haloSize]” aligned to the size of a vector register for all i and j indexes
  - We need padding before the full array to shift the start for aligning with “haloSize”
  - We need padding after the Nz size to ensure next line stays aligned
  - We use a standard allocation function: posix_memalign
Sample code

```c
static int padBeforeNz, padAfterNz;

void computePadding( int Nz, int haloSize ) {
    int vIdx = VLENGTH / sizeof( float );
    for ( padBeforeNz=0; (padBeforeNz+haloSize) % vIdx != 0; padBeforeNz++ )
        for ( padAfterNz=0; (Nz+2*haloSize+padAfterNz) % vIdx != 0; padAfterNz++ )
}

float ***alloc3dAlignedHalo( size_t Nx, size_t Ny, size_t Nz ) {
    float ***dummy = (float***)malloc( Nx*sizeof( float** ) );
    dummy[0] = (float**)malloc( Nx*Ny*sizeof( float* ) );
    posix_memalign( (void**)&dummy[0][0], VLENGTH,
                  (Nx*Ny*(Nz+padAfterNz)+padBeforeNz)*sizeof( float ) );
    dummy[0][0] += padBeforeNz;
    for ( size_t i=1; i < Nx; i++ ) dummy[i] = dummy[i-1]+Ny;
    for ( size_t i=1; i < Nx*Ny; i++ ) dummy[0][i] = dummy[0][i-1]+Nz+padAfterNz;
    return dummy;
}

void free3dAlignedHalo( float ***dummy ) {
    free( dummy[0][0]-padBeforeNz );
    free( dummy[0] );
    free( dummy );
}```