HBM code modernization - insights from a Xeon Phi experiment

Jacob Weismann Poulsen, DMI, Denmark
Per Berg, DMI, Denmark
Karthik Raman, Intel, USA
# Timeloop profile on 2S-IVB 2697-v2, 48 threads

<table>
<thead>
<tr>
<th>6H simulation of HBM (BB2nm)</th>
<th>Time [s]</th>
<th>Fraction [%]</th>
<th>SLOC [K]</th>
<th>Fraction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFLOW – advection</td>
<td>380.6</td>
<td>45.0</td>
<td>5.5</td>
<td>57.3</td>
</tr>
<tr>
<td>HD – smagorinsky</td>
<td>89.7</td>
<td>10.6</td>
<td>0.3</td>
<td>3.1</td>
</tr>
<tr>
<td>HD – momeqs</td>
<td>74.3</td>
<td>8.8</td>
<td>0.8</td>
<td>8.3</td>
</tr>
<tr>
<td>HD – tflow_up</td>
<td>62.0</td>
<td>7.3</td>
<td>0.2</td>
<td>2.1</td>
</tr>
<tr>
<td>HD – turbulence</td>
<td>59.6</td>
<td>7.0</td>
<td>0.9</td>
<td>9.4</td>
</tr>
<tr>
<td>TFLOW – diffusion</td>
<td>48.2</td>
<td>5.7</td>
<td>0.3</td>
<td>3.1</td>
</tr>
<tr>
<td>HD – sumuvwi</td>
<td>45.1</td>
<td>5.3</td>
<td>0.1</td>
<td>1.0</td>
</tr>
<tr>
<td>MISC Density</td>
<td>37.6</td>
<td>4.4</td>
<td>0.6</td>
<td>6.3</td>
</tr>
<tr>
<td>HD – bclprs</td>
<td>27.8</td>
<td>3.3</td>
<td>0.6</td>
<td>6.3</td>
</tr>
<tr>
<td>HD - maseqs</td>
<td>21.2</td>
<td>2.5</td>
<td>0.3</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**Sum**

<table>
<thead>
<tr>
<th>Time [s]</th>
<th>Fraction [%]</th>
<th>SLOC [K]</th>
<th>Fraction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>846.0</td>
<td>100.0</td>
<td>9.6</td>
<td>100.0</td>
</tr>
</tbody>
</table>

**Fraction of total timeloop**

<table>
<thead>
<tr>
<th>Time [s]</th>
<th>Fraction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.8</td>
<td>11.3</td>
</tr>
</tbody>
</table>

NB! Time reported is the **slowest** thread time measured using `omp_get_wtime()`
Outline

- Data structures
- Node performance
  - Thread parallelization
  - SIMD vectorization
- Performance results
  - For a full 6h simulation
  - For some of kernels used during the analysis
The data is sparse and highly irregular
Data layout (serial, indirect addressing)

do iw = 1,iw2
  i = ind(1,iw)
  j = ind(2,iw)
  ! all surface wet-points (i,j) reached with 
  ... u(iw) ...
enddo

doiw = 1,iw2
  kb = kh(iw)
  if (kb < 2) cycle
  i = ind(1,iw)
  j = ind(2,iw)
  mi0 = mcol(iw)-2
  do k = 2, kb
    ! all subsurface wet-points (k,i,j) are reached with 
    mi = mi0 + k
    ... u(mi) ...
  enddo
enddo
Data layout (serial)

- Data layout revisited:
  - Horizontally (unstructured) columns
  - Indirect addressing in the horizontal: 
    msrf(0:,0:), ind(1:2,:), mcol(0:), kh(0:)
  - Direct addressing in the vertical
- Observation

- Any enumeration of the surface points and any enumeration of the subsurface points imposes a unique cache pattern (D1, L2, L3, TLB) and some are obviously better than others. A true challenge to formulate a well-posed problem, though. Even if we managed that then finding the infimum would still be an NP-hard problem
- But space-filling-curves could be used to guide various heuristic approaches.
Data layout for threads (or tasks + explicit halo)

- Each thread will handle a subinterval of columns:

- Another layout of the columns will impose another decomposition for the threads (and the tasks).

```fortran
!$OMP PARALLEL DEFAULT(SHARED)
call foo(...); call bar(...); ...
!$OMP BARRIER
call halo_update(...)
!$OMP BARRIER
call baz(...); call quux(...); ...
!$OMP END PARALLEL
...
subroutine foo(...)
...  
call domp_get_domain(kh, 1, iw2, nl, nu, idx)
do iw=nl,nu  
i = ind(1,iw)
j = ind(2,iw)
  ! all threadlocal wet-points (:,:,,:) are reached here
...  
enddo
end subroutine foo
```
Thread (and task) load balancing

Formal definition:

Let $I = \{1, \ldots, m\}$ be the column index set and let $\{w_1, \ldots, w_m\}$ be the weights associated with the individual columns. Let $n$ denote the number of threads/tasks. A disjoint subinterval $I_i = \{[l_i; u_i]\}_{i=1,\ldots,n}$ covering of $I$ induces a cost vector $(c_1, \ldots, c_n)$ with $c_i = \sum_{j=l_i}^{u_i} w_j$. The cost $c$ of the covering is defined as $\max_i c_i$. The balance problem is to find a covering that minimizes $c$.

Observation: The **NP-hard problem** is reduced to the **integer partition problem** which provides an exact solution within time complexity: $O(m^2n)$.

Heuristics: Greedy approach or alternating greedy approach with runtime complexity: $O(n)$.

The weights can be a sum of sub weights while retaining problem complexity!
Thread parallelism - insights

- SPMD based (like MPI) and *not* loop based in order to minimize synchronization. A single openMP block with orphaned barriers surrounding synchronization points such as MPI haloswaps will do (nice side-effect: *No explicit scoping*).

- Consistent loop structures and consistent data layout and usage throughout the whole code implying that it is very easy to ensure a proper NUMA layout.

- Proper balancing is very important at scale (Amdahl). It can be done either offline (exact) or online (heuristic).

- Tuning options for balancing: Linear regression based on profiles, cf. DMI technical report tr12-20.
Refactoring for SIMD

Actually not as simple as it may sound....
SIMD target loops

Loops in the advection module are structured like this:

```fortran
do iw= ! horizontal - mpi/openmp parallelization
    do k= ! vertical   - vectorization
        do ic= ! innermost loop (in advection) with number of tracers
            ...
        enddo
    enddo
enddo
```

- Could vectorize at the iw-level but hardware is not ready. Thus, the aim is to vectorize all the k-loops
- None of the loops in the advection module vectorized initially and the trivial obstacles to vectorization were:
  - Indirections
  - Assumed-shape (F2008 contiguous attribute)
  - Nesting of branches (min/max/sign)
SIMD target loops

- All loops in HBM (despite the ones in advection) looks like:

```fortran
  do iw=  ! horizontal - mpi/openmp parallelization
    do k=  ! vertical    - vectorization
        enddo
      enddo
```

- Could vectorize at the iw-level but hardware is not ready. Thus, the aim is to vectorize all the k-loops
SIMD target loops

- Design choice for stencil codes: columns one-by-one using work arrays (tune for \textit{tripcount}) or whole stencil in one go (tune for \textit{intensity}) plus required remainder loops.

- Refactor strategy using \textit{computational intensity} and D1 pressure as the guidelines.

- The main challenges:
  - Serial algorithms (e.g. tridiagonal system)
  - Long latency operations
  - Indirect addressing and varying tripcounts implies unaligned accesses and generation of peel and remainder loops
  - Reductions
Premature abstraction is the root of all evil
(a hands on experience)
Premature abstraction is the root of all evil

- This topic may not coincide with your expectations:
  - I will **not** talk about how one can lose a leg with OOD (google it, e.g. Mike Acton).
  - I will **not** talk about how one may loose performance by using the HW abstraction that cores within a node have distributed memory.
  - …
  - Instead I will describe how the most simple HW abstraction (a 2D-array) will result in more than 2x performance loss on Xeon Phi and this should serve as a warning against using even the most simple abstractions without a prior analysis of consequences.
Premature abstraction is the root of all evil

The design idea was to hold all tracers in one 2D-array and treat all tracers in a similar fashion in one go like this (simplified illustration of the obstacle):

With dynamic \texttt{nc} the compiler vectorizes \texttt{nc}-loop:

(4): (col. 7) \textit{remark: LOOP WAS VECTORIZED}

With static \texttt{nc}, the compiler vectorizes the \texttt{k}-loop:

(1): (col 7) \textit{remark: LOOP WAS VECTORIZED}

Alas, assembler inspection revealed that gather operations were generated and runtime experiences confirm this.

```fortran
1  do k=2,kmax
2    k1 = k+off1
3    k2 = k+off2
4  t(1:nc,k) = t(1:nc,k) + A(k)*(B(1:nc,k1)-B(1:nc,k2))
5  enddo
```
Premature abstraction is the root of all evil

- The obstacle in a nutshell: A static nc (2) implies unrolling:

```fortran
  do k=1,kmax
    k1 = k+off1
    k2 = k+off2
    t(1,k) = t(1,k) + A(k)*(B(1,k1)-B(1,k2))
    t(2,k) = t(2,k) + A(k)*(B(2,k1)-B(2,k2))
  enddo
```

- And the unrolling implies that the optimizer sees the loop as a stride-2 loop but we know better so let's state what the compiler should have done (next slide)

- **LESSON LEARNED:** Within loops one should always ensure that all data-structures used are of the same dimension.
The compiler transformation that we hoped for:

- Proper handling of a mix of 2D and 1D (load with nc=2):

<table>
<thead>
<tr>
<th>zmm1←</th>
<th>t(1,1)</th>
<th>t(2,1)</th>
<th>t(1,2)</th>
<th>t(2,2)</th>
<th>t(1,3)</th>
<th>t(2,3)</th>
<th>t(1,4)</th>
<th>t(2,4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>zmm2←</td>
<td>t(1,5)</td>
<td>t(2,5)</td>
<td>t(1,6)</td>
<td>t(2,6)</td>
<td>t(1,7)</td>
<td>t(2,7)</td>
<td>t(1,8)</td>
<td>t(2,8)</td>
</tr>
<tr>
<td>zmm3←</td>
<td>B(1,1+k1)</td>
<td>B(2,1+k1)</td>
<td>B(1,2+k1)</td>
<td>B(2,2+k1)</td>
<td>B(1,3+k1)</td>
<td>B(2,3+k1)</td>
<td>B(1,4+k1)</td>
<td>B(2,4+k1)</td>
</tr>
<tr>
<td>zmm4←</td>
<td>B(1,5+k1)</td>
<td>B(2,5+k1)</td>
<td>B(1,6+k1)</td>
<td>B(2,6+k1)</td>
<td>B(1,7+k1)</td>
<td>B(2,7+k1)</td>
<td>B(1,8+k1)</td>
<td>B(2,8+k1)</td>
</tr>
<tr>
<td>zmm5←</td>
<td>B(1,1+k2)</td>
<td>B(2,1+k2)</td>
<td>B(1,2+k2)</td>
<td>B(2,2+k2)</td>
<td>B(1,3+k2)</td>
<td>B(2,3+k2)</td>
<td>B(1,4+k2)</td>
<td>B(2,4+k2)</td>
</tr>
<tr>
<td>zmm6←</td>
<td>B(1,5+k2)</td>
<td>B(2,5+k2)</td>
<td>B(1,6+k2)</td>
<td>B(2,6+k2)</td>
<td>B(1,7+k2)</td>
<td>B(2,7+k2)</td>
<td>B(1,8+k2)</td>
<td>B(2,8+k2)</td>
</tr>
<tr>
<td>zmm7←</td>
<td>A(1)</td>
<td>A(1)</td>
<td>A(2)</td>
<td>A(2)</td>
<td>A(3)</td>
<td>A(3)</td>
<td>A(4)</td>
<td>A(4)</td>
</tr>
</tbody>
</table>

- Trick

- Proper handling of a mix of 2D and 1D (arithmetic):

\[
\begin{align*}
zm9 & = zmm1 + zmm7 * (zmm3 - zmm5) ! k=1,4; nc=1,2 \\
zmm10 & = zmm2 + zmm8 * (zmm4 - zmm6) ! k=5,8; nc=1,2
\end{align*}
\]

- But did not get so we need to drop the 2D abstraction if performance matters to us.
Performance numbers

After dedicated work on the advection module and some general optimization efforts on the remaining parts.
## Performance results (work in progress)

<table>
<thead>
<tr>
<th>Single node performance</th>
<th>2S E5-2697-v2</th>
<th>KNC 7120A</th>
<th>2S-IVB/1KNC</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>6H simulation of HBM (BB2nm)</strong></td>
<td>Time [s]</td>
<td>Time [s]</td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>TFLOW – advection</td>
<td>380.6</td>
<td>330.0</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>HD – smagorinsky</td>
<td>89.7</td>
<td>56.6</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td>HD – momeqqs</td>
<td>74.3</td>
<td>111.3</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>HD – tflow_up</td>
<td>62.0</td>
<td>37.0</td>
<td>1.68</td>
<td></td>
</tr>
<tr>
<td>HD – turbulence</td>
<td>59.6</td>
<td>99.4</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>TFLOW – diffusion</td>
<td>48.2</td>
<td>41.8</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>HD – sumuvwi</td>
<td>45.1</td>
<td>29.0</td>
<td>1.55</td>
<td></td>
</tr>
<tr>
<td>MISC Density</td>
<td>37.6</td>
<td>36.6</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>HD – bclprs</td>
<td>27.8</td>
<td>27.2</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>HD - masseqs</td>
<td>21.2</td>
<td>14.7</td>
<td>1.44</td>
<td></td>
</tr>
<tr>
<td><strong>Timeloop took</strong></td>
<td><strong>865.5</strong></td>
<td><strong>819.5</strong></td>
<td><strong>1.06</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Extrapolation (all incl.)

**5days forecast:**

1.05

NB! Time reported for the timeloop subcomponents is the *slowest* thread time measured using `omp_get_wtime()`. The timeloop time is measured using `omp_get_wtime()` but the timer is outside an openMP block. The extrapolation includes initialization and finalization.
Sustained kernel performance, fraction of peak

<table>
<thead>
<tr>
<th>Name</th>
<th>Speedup</th>
<th>Fraction BW[?]</th>
<th>Fraction Flops[?]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2S E5-2697-v2</td>
<td>KNC 7120A</td>
<td>2S E5-2697-v2</td>
</tr>
<tr>
<td>Advection</td>
<td>1.2</td>
<td>100</td>
<td>75</td>
</tr>
<tr>
<td>Density</td>
<td>1.1</td>
<td>41</td>
<td>34</td>
</tr>
<tr>
<td>Bclprs</td>
<td>1.1</td>
<td>50</td>
<td>42</td>
</tr>
<tr>
<td>Turbulence</td>
<td>0.7</td>
<td>41</td>
<td>26</td>
</tr>
<tr>
<td>Momeqs</td>
<td>0.6</td>
<td>75</td>
<td>23</td>
</tr>
</tbody>
</table>

- The kernels have sloccouts in the 10K-40K range and the single thread loop in each have sloccounts in the 0.5K-6K range.
- Peak performance is defined as sustained performance using stream triad and HPL, respectively.
More information

- Older version of the code and testcase is available online: http://lotsofcores.com

- The preparation work is documented in a technical report: http://www.dmi.dk/fileadmin/user_upload/Rapporter/tr12-20.pdf
Benchmark systems

- Intel Xeon E5-2697 v2 (30Mb cache, 2.70 GHz)
  - Launched Q3, 2013
  - Number of cores/threads on 2 sockets: 24/48
  - DDR3-1600 MHz, 8*8 GB
  - Peak flops (HPL: 543 GF/s, 450 Watt)
  - Peak BW (Stream: 88 GB/s, 408 Watt)
  - Theoretical Peak BW: 8*1.6*4*2= 102 GB/s

- Intel Xeon Phi 7120A (30.5Mb cache, 1.238 GHz)
  - Launched Q2, 2013
  - Number of cores/threads: 60/240
  - GDDR5, 5.5 GT/s, 16 GB
  - Peak flops (HPL: 999 GF/s, 313 Watt)
  - Peak BW (Stream: 177 GB/s, 283 Watt)
Testcase for all benchmark timings presented here

- 2nm BaffinBay, freely available at http://lotsofcores.com
Acknowledgement

- Michael Greenfield, Intel
- Larry Meadows, Intel
- John Levesque, Cray
- Bill Long, Cray