$omp parallel do collapse(3)
  do p4=1,p4d
  do p5=1,p5d
  do p6=1,p6d
  do h1=1,h1d
  do h7=1,h7d
  !dec$ vector aligned
  do h2h3=1,h2d*h3d
   triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
   1 - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
  enddo
  enddo
  enddo
  enddo
  enddo
$omp end parallel do
## Agenda

### Tuesday 24th

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:00</td>
<td><strong>Course introduction</strong></td>
</tr>
<tr>
<td>10:00-10:15</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>10:15-11:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>11:00-12:00</td>
<td><strong>SIMD optimization</strong></td>
</tr>
<tr>
<td>12:00-12:45</td>
<td>Lunch break</td>
</tr>
<tr>
<td>12:45-13:45</td>
<td>Exercises</td>
</tr>
<tr>
<td>13:45-14:45</td>
<td>Performance analysis methodology</td>
</tr>
<tr>
<td>14:45-15:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>15:00-16:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>16:00-18:00</td>
<td>Code optimization workshop (voluntary)</td>
</tr>
</tbody>
</table>

### Wednesday 25th

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:00</td>
<td><strong>Memory optimization</strong></td>
</tr>
<tr>
<td>10:00-10:15</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>10:15-11:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>11:00-12:00</td>
<td><strong>Microarchitectural analysis</strong></td>
</tr>
<tr>
<td>12:00-12:45</td>
<td>Lunch break</td>
</tr>
<tr>
<td>12:45-13:45</td>
<td>Exercises</td>
</tr>
<tr>
<td>13:45-14:45</td>
<td>Advanced OpenMP</td>
</tr>
<tr>
<td>14:45-15:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>15:00-16:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>16:00-18:00</td>
<td>Code optimization workshop (voluntary)</td>
</tr>
</tbody>
</table>

### Thursday 26th

<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00-10:00</td>
<td><strong>OpenMP optimization</strong></td>
</tr>
<tr>
<td>10:00-10:15</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>10:15-11:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>11:00-12:00</td>
<td><strong>OpenMP target extensions</strong></td>
</tr>
<tr>
<td>12:00-12:45</td>
<td>Lunch break</td>
</tr>
<tr>
<td>12:45-13:45</td>
<td>Exercises</td>
</tr>
<tr>
<td>13:45-14:45</td>
<td><strong>OpenMP and MPI</strong></td>
</tr>
<tr>
<td>14:45-15:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>15:00-16:00</td>
<td>Exercises</td>
</tr>
<tr>
<td>16:00-16:15</td>
<td>Course wrap-up</td>
</tr>
</tbody>
</table>
Reference material

Title: Optimizing HPC Applications with Intel® Cluster Tools

Authors: Alexander Supalov, Andrey Semin, Michael Klemm, Chris Dahnken

Table of Contents:
Foreword by Bronis de Supinski (CTO LLNL)
Preface
Chapter 1: No Time to Read this Book?
Chapter 2: Overview of Platform Architectures
Chapter 3: Top-Down Software Optimization
Chapter 4: Addressing System Bottlenecks
Chapter 5: Addressing Application Bottlenecks: Distributed Memory
Chapter 6: Addressing Application Bottlenecks: Shared Memory
Chapter 7: Addressing Microarchitecture Bottlenecks
Chapter 8: Application Design Implications

Advanced Threading and Optimization exercises

Hardware

CSC cluster Taito (taito.csc.fi) has 10 full nodes reserved for the course both from the Haswell partition (two Intel Xeon E5-2690v3) and from the Xeon Phi partition (two Intel Xeon E5-2620-v2 and two Intel Xeon Phi 7120X). The names of the reservations are **advtao_hsw** (Taito) and **advtao_mic** (Taito Xeon Phi), respectively. Note that only the course training accounts are listed in the reservation, i.e., you cannot use your own CSC user account in conjunction with the reservation.

Batch job queue system

All the jobs must be submitted through the Slurm batch job queuing system. **All access** (including batch jobs) to the Xeon Phi partition is via node **m1**, accessible with **ssh** from a regular Taito login node.

To submit a single node job to the Taito Haswell partition (ten minute runtime at most):

```
srun -N1 -n1 -c24 --cpu_bind=none --reservation=advtao_hsw -p parallel -t00:10:00 ./executable
```

To submit a single node job with two Xeon Phi cards to Taito Xeon Phi partition (ten minute runtime at most):

```
srun -N1 -n1 -c12 --cpu_bind=none --reservation=advtao_mic --gres=mic:2 -p mic -t00:10:00 ./executable
```

For benchmarking purposes, it is preferred to use the Slurm option **--exclusive**.

To have an interactive shell session for 30 minutes to the Taito Haswell partition with X11 connection enabled:

```
srun -N1 -n1 -c1 --reservation=advtao_hsw -p parallel -t00:30:00 --x11 --pty $SHELL
```

Software

Latest Intel compiler stack is installed on Taito. To use it (software environment defaults to an older compiler), load the modules **intel/15.0.2 intelmpi/5.0.2**
and mkl/11.2.2. With the Intel software stack, it is recommended not to use cpu binding provided by Slurm, i.e., provide option \texttt{--cpu\_bind=none} at job launch time. When using Intel compilers, threads should be bound to cores by using the OpenMP and MPI affinity settings provided by the Intel software stack.

1. Remote desktop environment (ex1)

By following the instructions given in \texttt{nxkajaani\_howto.txt} set up a remote desktop connection to CSC’s datacenter in Kajaani. The connection will be used later during the course for profiling applications with VTune and other Intel tools.

2. SIMD vectorization (ex2)

The file \texttt{Vectorization\_Lab\_Linux.pdf} contains instructions for a step-by-step SIMD optimization of a simple matrix-vector kernel. The example is credited to Georg Zitzlsberger from Intel. There is an additional example concerning SIMD functions in the workbook as well.

3. Performance analysis methodology (ex3)

The file \texttt{Finding\_Hotspots\_Linux.pdf} contains instructions for a step-by-step profiling of a ray tracing application. Note that in order use VTune, Slurm reservation must have X11 plugin enabled (\texttt{--x11} –parameter) and the VTune module (\texttt{intel-vtune/15.1.1}) must be loaded to the environment.

4. Memory optimization (ex4)

Xeon Phi is able to attain about 170GB/sec bandwidth in the Stream triad benchmark, which computes a daxpy –like operation for two vectors. For the rules and results see \texttt{http://www.cs.virginia.edu/stream/}. The benchmark has been created by Dr. John D. McCalpin.

\begin{itemize}
  \item[a)] The file \texttt{stream\_ref(.c|.F90)} contains a reference version of the benchmark. Compile the benchmark (\texttt{stream\_ref}) for the language of your choice with the given \texttt{Makefile\_c|ftn}. Run the benchmark on a Xeon Phi (native mode) to replicate the stream triad results. It is recommended to use 60 cores and scatter thread affinity for the task.
  \item[b)] The file \texttt{stream(.c|.F90)} contains a slightly modified version of the stream benchmark. Compile the benchmark (\texttt{stream}) for the language of your choice
with the given `Makefile_(c|ftn)`. Run the benchmark on a Xeon Phi (native mode) to compare the results to the previously computed reference results. By only modifying the code (look for TODO -tags), try to match the results of the stream triad portion of the reference benchmark.

5. Microarchitectural analysis (ex5)

The purpose of this exercise is to perform microarchitectural analysis with VTune to the kernels from previous exercises, i.e., “2. SIMD vectorization” and “4. Memory optimization”.

a) The directory `matvec/` contains unoptimized and optimized C (`matvec/c/`) and Fortran (`matvec/fortran/`)-versions of matrix-vector multiply kernel of the exercise 2. Compile the code for the language of your choice with the given `Makefile` to generate `matvec` (unoptimized) and `matvec_opt` (optimized) binaries. By using VTune, run a Microarchitectural Analysis with General Exploration for both binaries. Compare the resulting number of instructions retired, CPI rate and also other indicators between the unoptimized and optimized versions of the code. Then investigate and compare the most time consuming hotspot of the code. Does having a low CPI value correlate to good performance? Then, using the assembly view functionality of VTune, try to determine if the hotspot vectorized or not.

b) The directory `stream/` contains unoptimized and optimized C (`stream/c/`) and Fortran (`stream/fortran/`)-versions of the stream benchmark of exercise 2. Compile the code for the language of your choice with the given `Makefile` to generate `stream` (unoptimized) and `stream_ref` (optimized) binaries. By using VTune, run a Microarchitectural Analysis with Bandwidth for both binaries. Compare the resulting number of instructions retired, CPI rate, LLC Miss and also other indicators between the unoptimized and optimized versions of the code. How does the sampled bandwidth measured by VTune compare to the bandwidth actual bandwidth measured by the stream benchmark?

6. Advanced OpenMP (ex6)

The file `nbody(.c|.f90)` implements a simple N-body iteration. Compile the code for the language of your choice with the given `Makefile` (by typing `make c` or `make fortran`). The resulting binary `nbody` takes the number of bodies to simulate as the first argument. Try running a simulation by using 262144 bodies in total.
By using VTune or compiler vectorization reports, investigate if the code vectorizes well. Try to improve the vectorization properties and to speed up the code by using the OpenMP SIMD construct and any optimization techniques you are aware of.

7. Optimization for threads (ex7)

The code implements a very simplistic matrix-vector multiplication that uses OpenMP for multi-threading. Compile the code (by typing `make`) and two binaries will be created: `mxv.seqinit` and `mxv.parinit`. Although they are named differently, each of the binaries allocates the matrix and the vector sequentially and thus suffer from a NUMA problem. Try running the codes with `make runs` and each binary will run with different thread distribution and different thread numbers. What average runtimes do you observe?

As the next step, find the initialization of the matrix and the vector, and parallelize it according so it matches the parallelization of the matrix-vector multiplication. Run the binaries again (make runs) and check the performance. Do you spot a difference?

8. OpenMP target features (ex8)

The file `ex_jacobi(.c|.F90)` implements a simple Jacobi iteration.

a) Add OpenMP directives to the code in such a way that the computational work is done in parallel.

b) Modify the OpenMP implementation in such a way that the computation is done entirely on the device, i.e., no additional data transfers take place between the host and the device.

c) (Bonus *) Optimize the performance of the implementation.

9. OpenMP and MPI (ex9)

Benchmark hybrid MPI+OpenMP kernels by using EPCC’s OpenMP/MPI Mixed-Mode Microbenchmarks. Both C and Fortran versions of the source code have been already downloaded to the directories `EPCC_mixedMode_C/` and `EPCC_mixedMode_Fortran/`, respectively. Compile the benchmarks (of the source code of your language choice) with an Intel MPI compiler. Instructions on the compilation and use of the benchmarks can be acquired from `http://www2.epcc.ed.ac.uk/~markb/mpiopenmpbench/intro.html`.

A sample input file `multipingpong.bench` performs a two node multipingpong benchmark and can be used to replicate the results given in the course handouts.