PERFORMANCE OPTIMISATION ON XEON PHI

Adrian Jackson, Michele Weiland, Fiona Reid, Manos Farsarakis, David Scott
adrianj@epcc.ed.ac.uk
PERFORMANCE DEOPTIMISATION ON XEON PHI

Adrian Jackson, Michele Weiland, Fiona Reid, Manos Farsarakis, David Scott

adrianj@epcc.ed.ac.uk
Intel’s IPCC program

• Collaboration between Intel and leading Universities around the world

  “Intel® Parallel Computing Centers are universities, institutions, and labs that are leaders in their field, focusing on modernizing applications to increase parallelism and scalability through optimizations that leverage cores, caches, threads, and vector capabilities of microprocessors and coprocessors.”
IPCC Proposal

• Two main aims
  • Port and optimise codes for Xeon Phi
  • Optimise codes for Xeon for ARCHER system

• Target ‘Grand Challenges’ codes which are also heavily used in the UK
  • codes we have strong knowledge of/working relationship with
Intel Xeon Phi

- Intel Larrabee: “A Many-Core x86 Architecture for Visual Computing”
  - Release delayed such that the chip missed competitive window of opportunity.
  - Larrabee was not released as a competitive product, but instead a platform for research and development (Knight’s Ferry).
- Knights Corner derivative chip
  - Intel Xeon Phi – co-processor
  - Many Integrated Cores (MIC) architecture. No longer aimed at graphics market
    - Instead “Accelerating Science and Discovery”
- PCIe Card
  - 60 cores/240 threads/1.054 GHz
  - 8 GB/320 GB/s
  - 512-bit SIMD instructions
- Hybrid between GPU and many-core CPU
## Intel Xeon Phi

<table>
<thead>
<tr>
<th></th>
<th>3100 series</th>
<th>5100 series</th>
<th>7100 series</th>
</tr>
</thead>
<tbody>
<tr>
<td>cores</td>
<td>57</td>
<td>60</td>
<td>61</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>1.100 GHz</td>
<td>1.053 GHz</td>
<td>1.238 GHz</td>
</tr>
<tr>
<td>DP Performance</td>
<td>1 Tflops</td>
<td>1.01 TFlops</td>
<td>1.2 TFlops</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>240 GB/s</td>
<td>320 GB/s</td>
<td>352 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>6 GB</td>
<td>8 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

- Usable in different ways
  - Offload kernels
  - “Native” direct run applications
Achievable Performance

• 1 to 1.2 TFlop/s double precision performance
  • Dependent on using 512-bit vector units
  • And FMA instructions

• 240 to 352 GB/s peak memory bandwidth

• ~60 physical cores
  • Each can run 4 threads
  • Must run at least 2 threads to get full instruction issue rate
  • Don’t think of it as 240 threads, think of it as 120 plus more if beneficial

• 2.5x speedup over host is good performance
  • Highly vectorised code, no communications costs

• MPI performance
  • Can be significantly slower than host
Serial code

**The Serial Factor**

Serial Factor = Clock Factor * ILP Factor * Issue Factor

Where

Clock Factor = 2.6 / 1.09

For FMA type calculations
ILP Factor*** = 2 / 2 = 1

For non-FMA type calculations
ILP Factor = 2 / 1

Issue factor = Num cycles to issue instruction on Phi / Num cycles to issue instruction on Xeon = 2 / 1

Note: in single threaded code Xeon Phi uses two cycles to issue an instruction (in threaded mode it takes just one cycle)

** FMA: source code is capable of using Fused Multiple Add when built for Xeon Phi

FMA**

x4.77 slower

Non-FMA

x9.54 slower
PingPong Bandwidth

![Graph showing PingPong Bandwidth]
PingPong Latency

The graph shows the latency of ping pong operations in terms of message size (in bytes) and latency ratio. The lines represent different scenarios:

- Xeon Phi/host latency ratio 2 procs
- Xeon Phi/host latency ratio full node

The y-axis represents the latency ratio, while the x-axis shows the message size in bytes.
MPI_Allreduce

![Graph showing MPI_Allreduce performance for different message sizes and architectures. The graph plots average time in microseconds against message size. There are four lines representing different configurations: Host 16 procs, Xeon Phi 60 procs, Xeon Phi 120 procs, and Xeon Phi 240 procs.]
General approach

- Work on 3 codes
  - GS2, COSA, and CP2K
- All FORTRAN
- Investigate and optimise vectorisation of codes
  - Use profiler and compiler tools to evaluate vectorisation
  - Modify computationally expensive code to improve vectorisation
- Improve/implement hybrid parallelisation
  - Can help for both standard and phi systems
  - Reduce memory footprint
- Reduce serial code
  - I/O etc....
GS2

- Flux-tube gyrokinetic code
  - Initial value code
  - Solves the gyrokinetic equations for perturbed distribution functions together with Maxwell’s equations for the turbulent electric and magnetic fields
  - Linear (fully implicit) and Non-linear (dealiased pseudo-spectral) collisional and field terms
  - 5D space – 3 spatial, 2 velocity
  - Different species of charged particles
- Advancement of time in Fourier space
- Non-linear term calculated in position space
  - Requires FFTs
  - FFTs only in two spatial dimensions perpendicular to the magnetic field
- Heavily dominated by MPI time at scale
  - Especially with collisions
Initial hybrid version

- Performance of existing version
New hybrid implementation

- Still funnelled communication model
- OpenMP done at a higher level in the code
- Single parallel region per time step
  - Better can be achieved (single parallel region per run)
- Some code excluded but computationally expensive code all hybridised

<table>
<thead>
<tr>
<th>MPI processes</th>
<th>OpenMP threads</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>1</td>
<td>16.54</td>
</tr>
<tr>
<td>96</td>
<td>2</td>
<td>18.34</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>16.46</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>30.86</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>28.3</td>
</tr>
</tbody>
</table>
Port to Xeon Phi

- Pure MPI code performance:
  - ARCHER (2x12 core Xeon E5-2697, 16 MPI processes): 3.08 minutes
  - Host (2x8 core Xeon E5-2650, 16 MPI processes): 4.64 minutes
  - 1 Phi (176 MPI processes): 7.34 minutes
  - 1 Phi (235 MPI processes): 6.77 minutes
  - 2 Phis (352 MPI processes): 47.71 minutes

- Hybrid code performance
  - 1 Phi (80 MPI processes, 3 threads each): 7.95 minutes
  - 1 Phi (120 MPI processes, 2 threads each): 7.07 minutes
Vector (de)optimisations

- Vector optimising work unsuccessful
  - A number of poorly vectorising targets identified
  - Code restructuring and directives not able to improve performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Compiler flags</th>
<th>Compiler directives</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-O2 (original)</td>
<td>-O2 (original)</td>
<td>16.46</td>
</tr>
<tr>
<td>invert_rhs_1</td>
<td>-align array64byte</td>
<td>attributes align, vector aligned</td>
<td>16.73</td>
</tr>
<tr>
<td>get_source_term</td>
<td>-align array64byte</td>
<td>attributes align, vector aligned</td>
<td>16.99</td>
</tr>
<tr>
<td>get_source_term</td>
<td>-align array64byte</td>
<td>attributes align, vector aligned (only for variables gexp1, gexp2 and gexp3)</td>
<td>16.72</td>
</tr>
</tbody>
</table>
Complex number optimisation

• Much of GS2 uses FORTRAN Complex numbers
  • However, often imaginary and real parts are treated separately
  • Can affect vectorisation performance

• Work underway to replace with separate arrays
  • Initial performance numbers demonstrate performance improvement on Xeon Phi
  • 2-3% for a single routine when using separate arrays
COSA

- Fluid dynamics code
  - Harmonic balance (frequency domain approach)
  - Unsteady navier-stokes solver
  - Optimise performance of turbo-machinery like problems
  - Multi-grid, multi-level, multi-block code
  - Parallelised with MPI and with MPI+OpenMP
## Xeon Phi Performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Number of hardware elements</th>
<th>Occupancy</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MPI processes</td>
<td>1/2</td>
<td>8/16</td>
<td>2105.71</td>
</tr>
<tr>
<td>16 MPI processes</td>
<td>2/2</td>
<td>16/16</td>
<td>1272.54</td>
</tr>
<tr>
<td>64 MPI processes</td>
<td>1/2</td>
<td>64/240</td>
<td>3874.45</td>
</tr>
<tr>
<td>64 MPI processes 3 OpenMP threads</td>
<td>1/2</td>
<td>192/240</td>
<td>2963.58</td>
</tr>
<tr>
<td>118 MPI processes 4 OpenMP threads</td>
<td>2/2</td>
<td>472/480</td>
<td>2118.05</td>
</tr>
<tr>
<td>128 MPI processes 3 OpenMP threads</td>
<td>2/2</td>
<td>384/480</td>
<td>1759.30</td>
</tr>
</tbody>
</table>

- **Hardware:**
  - 2 x Xeon Sandy Bridge 8-core E5-2650 2.00GHz
  - 2 x Xeon Phi 5110P 60-core 1.05GHz

- **Test case**
  - 256 blocks
  - Maximum 7 OpenMP threads
Serial optimisations

- Manual removal of floating point loop invariants divisions
  
  ```
  do ipde = 1,4
    fac1 = fact * vol(i,j)/dt
  end do
  ```

  ```
  recip = 1.0d / dt
  do ipde = 1,4
    fact1 = fact * vol(i,j) * recip
  end do
  ```

- Provides ~15% speedup so far on Xeon Phi
  - No real benefit noticed on host
  - Changes the results
I/O

• Identified that reading input is now significant overhead for this code
  • Output is done using MPI-I/O, reading is done serially
  • File locking overhead grows with process count
  • Large cases ~GB input files

• Parallelised reading data
  • Reduce file locking and serial parts of the code

• One or two orders of magnitude improvement in performance at large process counts
  • 1 minute down to 5 seconds
Future work

- Further serial optimisation
  - Cache blocking

- 3D version of the code now developed
  - Porting optimised and hybrid version to this

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Number of hardware elements</th>
<th>Occupancy</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MPI processes</td>
<td>1/2</td>
<td>8/16</td>
<td>2105.71</td>
</tr>
<tr>
<td>16 MPI processes</td>
<td>2/2</td>
<td>16/16</td>
<td>1272.54</td>
</tr>
<tr>
<td>128 MPI processes</td>
<td>1/2</td>
<td>128/240</td>
<td>1903.51</td>
</tr>
<tr>
<td>64 MPI processes            3 OpenMP threads</td>
<td>1/2</td>
<td>192/240</td>
<td>2214.56</td>
</tr>
<tr>
<td>128 MPI processes            3 OpenMP threads</td>
<td>2/2</td>
<td>384/480</td>
<td>1503.45</td>
</tr>
</tbody>
</table>
CP2K

• Atomistic and molecular simulations of solid state, liquid, molecular, and biological system
• MPI and hybrid parallelisations implemented
• Heavily uses internal and external libraries for core computations
• Other sites working on Xeon Phi
  • Offload functionality
  • Investigating compiler optimisations
• EPCC has previously worked on a native mode Xeon Phi port
  • Performance not great, 50% compared to CPU version (16 cores), low memory requirement restricts accuracy
  • This work identified a number of vectorisation targets
CP2K Performance

- H2O-64 benchmark for 1 time step on NEC
- Percentage of runtime as reported by Perftools-lite
Performance of CP2K H2O-64 benchmark on the Xeon Phi

- **MPI**
- **OpenMP**
- **MPI/OpenMP - original placement**
- **MPI/OpenMP - optimal placement**

![Graph showing the performance of CP2K H2O-64 benchmark on the Xeon Phi](image-url)
CP2K Test Suite

- Regression test suite and continuous integration testing important to ensure CP2K maintains correctness
  - Important as it informs which compilers and libraries users can build the application with
- Ported test suite and framework to enable use of Intel compilers and MKL libraries
  - Build both on host and Xeon Phi
- Essential task before code modification could be undertaken
  - Picked up a number of bugs with the code and Intel compilers/libraries
Vector (de)optimisations

- Vector optimising work unsuccessful
- CP2K uses auto-tuning library routines for core kernels
- Vectorising these routines struggled due to code structure

<table>
<thead>
<tr>
<th>Code version</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original code</td>
<td>2.423632</td>
</tr>
<tr>
<td>Adding !DIR$ IVDEP to loop over ig</td>
<td>2.472624</td>
</tr>
<tr>
<td>Attempt 1: Array syntax</td>
<td>2.439629</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.437631</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ VECTOR ALWAYS on loop over ig</td>
<td>2.436625</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + ISOMP SIMD privates(lo) on loop over ig</td>
<td>2.484623</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + align map and pol_x</td>
<td>2.479622</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + align map and pol_x + ISOMP SIMD on loop over ig</td>
<td>2.524676</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax</td>
<td>2.477623</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.473624</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.580608</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + ISOMP SIMD privates(s) on loop over ig</td>
<td>2.620602</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + localmap 1d array used to compute I</td>
<td>2.475624</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(iib) to stop(iib) for each block of contiguous iterations</td>
<td>2.626000</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(iib) to stop(iib) for each block of contiguous iterations + !DIR$ IVDEP on loop over i</td>
<td>2.625601</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(iib) to stop(iib) for each block of contiguous iterations + !DIR$ VECTOR ALWAYS on loop over i</td>
<td>2.627600</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(iib) to stop(iib) for each block of contiguous iterations + ISOMP SIMD on loop over i</td>
<td>2.582607</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(iib) to stop(iib) for each block of contiguous iterations + ISOMP SIMD privates(s) on loop over i</td>
<td>2.634599</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc</td>
<td>2.769579</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ IVDEP on loop over i</td>
<td>2.760580</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ VECTOR ALWAYS on loop over i</td>
<td>2.755581</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + ISOMP SIMD on loop over i</td>
<td>2.754581</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + ISOMP SIMD privates(s) on loop over i</td>
<td>2.757580</td>
</tr>
</tbody>
</table>
Summary

- Working on large FORTRAN MPI (or hybrid) simulation codes
  - Already heavily optimised, no real low hanging fruit
- Single code base work highly favoured
  - Large scale codes won’t maintain mixed source versions
  - Favours native mode parallelisation
- Hybrid parallelisations will help elsewhere
  - Obvious target for many MPI programs
- Intel compilers v15 has impacted performance across the board for our codes
  - Slower with v15 vs v14
- MPI across Xeon Phi’s can heavily impact performance
  - Global comms dominated codes don’t currently scale
  - Local comms codes can scale well
Acknowledgements

• Work supported by Intel Parallel Computing Centre program
• Some work also supported by PRACE and EPSRC Plasma HEC grant