Scientific Simulations on Thousands of GPUs with Performance Portability

Alan Gray and Kevin Stratford
EPCC, The University of Edinburgh
CORAL procurement

- Three “pre-exascale” machines have been announced in the US, each in the region of 100-300 petaflops
- *Summit* at ORNL and *Sierra* at LLNL will use **NVIDIA GPUs** (with IBM CPUs).
- *Aurora* at Argonne will use **Intel Xeon Phi many-core CPUs** (Cray system)
- *Performance Portability* is the key issue for the programmer
Outline

• Applications: Ludwig and MILC

• Performance Portability with targetDP

• Performance results on GPU, CPU and Xeon Phi
  ▪ Using same source code for each

• Scaling to many nodes with MPI+targetDP
Ludwig Application

- **Soft matter** substances or **complex fluids** are all around us
- Ludwig: uses lattice Boltzmann and finite difference methods to simulate a wide range of systems

- Improving the understanding of, and ability to manipulate, **liquid crystals** is a very active research area

- But required simulations can be extremely computationally demanding, due to range of scales involved

- targetDP developed in co-design with Ludwig

---


MILC application

- Lattice QCD simulations provide numerical studies to help understand how quarks and gluons interact to form protons, neutrons and other elementary particles.

- The Unified European Application Benchmark Suite (UEABS) is a set of 12 application codes designed to be representative of EU HPC usage:
  - including Lattice QCD component, derived from MILC codebase
  - [http://www.prace-ri.eu/ueabs/](http://www.prace-ri.eu/ueabs/)

- targetDP applied to this application benchmark to enable for GPU and Xeon Phi
Multi-valued data

- For most scientific simulations the bottleneck is **memory bandwidth**
- Simulation data consists of **multiple values at each site**
- In memory, we have a choice of how to store this
  - \( | rgb | rgb | rgb | rgb | \) (Array of Structs AoS)
  - \( | rrrr | gggg | bbbb | \) (Struct of Arrays SoA)
  - Most general case is Array of Structs of (short) Arrays (AoSoA)
  - E.g. \( | | rr | gg | bb | | | rr | gg | bb | | \) has SA length of 2
  - Major effect on bandwidth. Best layout architecture-specific
- **Solution:**
  - De-couple memory layout from application source code
  - Can simply be done with macro, e.g.
    \[
    \text{field}[\text{INDEX}(\text{iDim}, \text{iSite})]
    \]
targetDP

- Simple serial code example: loop over N grid points
  - With some operation ... at each point

```c
int iSite;
for (iSite = 0; iSite < N; iSite++)
{
    ...
}
```
• **OpenMP**

```c
int iSite;
#pragma omp parallel for
for (iSite = 0; iSite < N; iSite++)
{
    ...
}
```

• **CUDA**

```c
__global__ void scale(double* field) {

    int iSite;
    iSite=blockIdx.x*blockDim.x+threadIdx.x
    if(iSite<N)
    {
        ...
    }
    return;
}
```

• **targetDP**

```c
__targetEntry__ void scale(double* field){

    int iSite;
    __targetTLP__(iSite, N)
    {
        ...
    }
    return;
}
```
• **PROBLEM**: to fully utilise modern CPUs, compiler must vectorize innermost loops to create vector instructions.

• **SOLUTION**: TLP can be strided, such that each thread operates on chunk of VVL lattice sites
  - VVL must be 1 for above example to work
  - But we can set VVL>1, and add a new innermost loop

```c
__targetEntry__ void scale(double* t_field) {

    int index;
    __targetTLP__(iSite, N) {

        int iDim;
        for (iDim = 0; iDim < 3; iDim++) {

            t_field[INDEX(iDim,iSite)] = t_a*t_field[INDEX(iDim,iSite)];

        }
    }
    return;
}
```
ILP can map to loop over chunk of lattice sites, with OpenMP SIMD directive

- Easily vectorizable by compiler
- VVL can be tuned specifically for hardware, e.g. VVL=8 will create single IMCI instruction for 8-way DP vector unit on Xeon Phi
  - Without this, performance is several times worse on Xeon Phi
- We can just map to an empty macro, when we don’t want ILP

```c
__targetEntry__ void scale(double* t_field) {
    int baseIndex;
    __targetTLP__(baseIndex, N) {
        int iDim, vecIndex;
        for (iDim = 0; iDim < 3; iDim++) {
            __targetILP__(vecIndex) \n            t_field[INDEX(iDim,baseIndex+vecIndex)] = \n            t_a*t_field[INDEX(iDim,baseIndex+vecIndex)];
        }
        return;
    }
}
```
• Function called from host code using wrappers to CUDA API
  
  - That can alternatively map to regular CPU (malloc, memcpy etc)

    ```c
    targetMalloc((void **) &t_field, datasize);
    
    copyToTarget(t_field, field, datasize);
    copyConstDoubleToTarget(&t_a, &a, sizeof(double));
    
    scale __targetLaunch__(N) (t_field);
    targetSynchronize();
    
    copyFromTarget(field, t_field, datasize);
    targetFree(t_field);
    ```
## Results

<table>
<thead>
<tr>
<th>Processor</th>
<th>Product Details</th>
<th>Peak Perf. (Dbl. Prec.)</th>
<th>Stream Triad (Measured)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ivy-Bridge</td>
<td>Intel Xeon E5-2697 v2 12-core CPU @ 2.70GHz</td>
<td>259 Gflops</td>
<td>49.8 GB/s</td>
</tr>
<tr>
<td>Haswell</td>
<td>Intel Xeon E5-2630 v3 8-core CPU @ 2.40GHz</td>
<td>154 Gflops</td>
<td>40.9 GB/s</td>
</tr>
<tr>
<td>Interlagos</td>
<td>AMD Opteron 6274 16-core CPU @ 2.20GHz</td>
<td>141 Gflops</td>
<td>32.4 GB/s</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>Intel Xeon Phi 5110P 60-core CPU @ 1.053GHz</td>
<td>1.01 Tflops</td>
<td>158.4 GB/s</td>
</tr>
<tr>
<td>K20X</td>
<td>Nvidia Tesla K20X GPU</td>
<td>1.31 Tflops</td>
<td>181.3 GB/s</td>
</tr>
<tr>
<td>K40</td>
<td>Nvidia Tesla K40 GPU</td>
<td>1.43 Tflops</td>
<td>192.1 GB/s</td>
</tr>
</tbody>
</table>

- **Same** performance-portable targetDP **source code** on all architectures
<table>
<thead>
<tr>
<th></th>
<th>Intel Ivybridge 12-core CPU</th>
<th>Intel Haswell 8-core CPU</th>
<th>AMD Interlagos 16-core CPU</th>
<th>Intel Xeon Phi</th>
<th>NVIDIA K20X GPU</th>
<th>NVIDIA K40 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>time [s]</td>
<td>200</td>
<td>300</td>
<td>600</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Best Config:</td>
<td>AoSoA, VVL=4</td>
<td>AoS, VVL=1</td>
<td>AoS, VVL=1</td>
<td>AoSoA, VVL=8</td>
<td>SoA, VVL=1</td>
<td>SoA, VVL=1</td>
</tr>
</tbody>
</table>

**Full Ludwig Liquid Crystal 128x128x128 Test Case**

- Ludwig Remainder
- Advect. Bound.
- Advection
- LC Update
- Chemical Stress
- Order Par. Grad.
- Collision
- Propagation
Full MILC Conjugate Gradient 64x64x32x8 Test Case

- MILC Remainder
- Shift
- Scalar Mult. Add
- Insert
- Insert & Mult.
- Extract & Mult.
- Extract

Best Config:
- AoSoA, VVL=4
- AoS, VVL=1
- AoS, VVL=1
- AoSoA, VVL=8
- SoA, VVL=1
- SoA, VVL=1

- Intel Ivybridge 12-core CPU
- Intel Haswell 8-core CPU
- AMD Interlagos 16-core CPU
- Intel Xeon Phi
- NVIDIA K20X GPU
- NVIDIA K40 GPU
Comparing with capability of hardware

- Use “Roofline” model

- It can be shown that all our kernels are memory-bandwidth bound
  - Compare kernel bandwidth with STREAM benchmark
# MPI+targetDP Supercomputer Scaling

<table>
<thead>
<tr>
<th></th>
<th>Titan</th>
<th>Archer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>Oak Ridge National Laboratory</td>
<td>University of Edinburgh</td>
</tr>
<tr>
<td>Product</td>
<td>Cray XK7</td>
<td>Cray XC30</td>
</tr>
<tr>
<td>Per Node</td>
<td>One Interlagos &amp; one K20X</td>
<td>Two Ivy-Bridge</td>
</tr>
<tr>
<td>Nodes</td>
<td>18,688</td>
<td>4,920</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Cray Gemini</td>
<td>Cray Aries</td>
</tr>
</tbody>
</table>
Ludwig Liquid Crystal: 128x128x128

- **Titan CPU**
  - (One 16-core Interlagos per node)

- **Archer CPU**
  - (Two 12-core Ivybridge per node)

- **Titan GPU**
  - (One K20X per node)
Ludwig Liquid Crystal: 1024x1024x512

- **Titan CPU**
  (One 16-core Interlagos per node)

- **Archer CPU**
  (Two 12-core Ivy-bridge per node)

- **Titan GPU**
  (One K20X per node)
MILC Conjugate Gradient: 64x64x32x8

- **Titan CPU**
  (One 16-core Interlagos per node)

- **Archer CPU**
  (Two 12-core Ivy-bridge per node)

- **Titan GPU**
  (One K20X per node)
MILC Conjugate Gradient: 64x64x64x192

- Titan CPU (One 16-core Interlagos per node)
- Archer CPU (Two 12-core Ivy-bridge per node)
- Titan GPU (One K20X per node)
Summary

• targetDP is a simplistic framework that allows grid-based codes to perform well on modern multi/many-core CPUs as well as GPUs
  ▪ By abstracting parallelism and memory spaces
  ▪ Express TLP and ILP. We can see that exposing ILP is crucial on Xeon Phi today, and vector units will continue to get wider on future CPUs
  ▪ It is also crucial to de-couple memory layout by abstracting memory accesses.
• We demonstrated performance portability across multiple modern architectures
• GPUs and Xeon Phi are significantly faster than CPUs, because they offer higher memory bandwidth
  ▪ GPUs have advantage over Xeon Phi
• MPI+targetDP is suitable for large-scale supercomputing
  ▪ NVLINK should help with strong multi-GPU scaling
• We have been concentrating on structured grid-based applications, but similar thinking may be fruitful for other areas
• targetDP is freely available
  ▪ http://ccpforge.cse.rl.ac.uk/svn/ludwig/trunk/targetDP/README
Acknowledgements