Designing and Optimizing LQCD code using OpenACC

E Calore, S F Schifano, R Tripiccione

Enrico Calore

University of Ferrara and INFN-Ferrara, Italy

GPU Computing in High Energy Physics

Pisa, Sep. 10th, 2014
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
GPUs and MICs performances are growing

Theoretical Peak Performance, Double Precision

Courtesy of Dr. Karl Rupp, Technische Universität Wien

E. Calore (INFN of Ferrara)

LQCD using OpenACC

Pisa, Sep 10th, 2014 4 / 27
GPUs and MICs use in HPC is growing

Accelerator architectures in the Top500 Supercomputers
Outline

1. Introduction
   - Hardware trends
   - **Software needs**
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
How to get our code ready for future HPC systems?

Given that:

- available parallelism in CPUs is increasing
- accelerator architectures are quickly evolving
- CPUs and Accelerators are getting closer
- is hard to predict if one architecture will prevail and, if it is the case, which one will

Code has to:

- be able to exploit hardware parallelism at different levels
- be portable across different architectures
- not be subject to (excessive) performance degradation due to its portability
**OpenCL (Open Computing Language):**

- The same code can be run on CPUs, GPUs, MICs, etc.
- Functions to be offloaded on the accelerator have to be explicitly programmed (as in CUDA)
- Data movements between host and accelerator has to be explicitly programmed (as in CUDA)
- NVIDIA do not support it anymore

**OpenACC (for Open Accelerators):**

- The same code (will probably) run on CPUs, GPUs, MICs, etc.
- Functions to be offloaded are “annotated” with `#pragma` directives
- Data movements between host and accelerator could be managed automatically or manually
- Support is still limited, but seems to be quickly growing
Why it is worth to use OpenACC

**Code modifications could be minimal**

- Thanks to the annotation of pre-existing C code using `#pragma` directives.
- Programming efforts needed mainly to re-organize the data structures and to efficiently design data movements.

**If it will be superseded, programming efforts would not be lost**

- OpenMP community is working towards the native support for accelerators in the language (maybe in several years).
- Switching between directive based languages should be just a matter of changing the `#pragma` clauses.
- Also other directive based languages would benefit from data re-organization and efficiently designed data movements.

NVIDIA is pushing for its adoption and is strongly committed to develop PGI
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
OpenACC example: the Saxpy function

```c
{  
  my_saxpy(x, y);
}
```

```c
void my_saxpy(float * x, float * y) {

    #pragma acc kernels loop
    for (int i = 0; i < N; ++i)
        y[i] = a*x[i] + y[i];
}
```

OpenACC code computing a saxpy function on vectors x and y. `#pragma` clause identify the region to run on the accelerator.
OpenACC code computing a saxpy function on vectors \(x\) and \(y\). `#pragma` clauses identifies the region to run on the accelerator and how to manage data transfers.
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
AoS vs SoA in a 3D Lattice Boltzmann Application

d3q19 CUDA / OpenACC Propagate execution time

- CUDA SoA
- OACC SoA
- CUDA AoS
- OACC AoS

Time [ms]

Block/Gang size

16 32 64 128 256 512 1024
Memory layout for LQCD: AoS vs SoA

// fermions stored as AoS:
typedef struct {
    double complex c1; // component 1
    double complex c2; // component 2
    double complex c3; // component 3
} vec3_aos_t;

vec3_aos_t fermions[sizeh];

AoS: corresponding components of different sites are interleaved, causing strided memory-access and leading to coalescing issues.

// fermions stored as SoA:
typedef struct {
    double complex c0[sizeh]; // components 1
    double complex c1[sizeh]; // components 2
    double complex c2[sizeh]; // components 3
} vec3_soa_t;

vec3_soa_t fermions;

SoA: corresponding populations of different sites are allocated at contiguous memory addresses, enabling coalescing of accesses, and making use of full memory bandwidth.
Fermions vectors data structure

```c
typedef struct {
    double complex c0[sizeh];
    double complex c1[sizeh];
    double complex c2[sizeh];
} vec3_soa_t;
```

Since C99 float/double standard complex data type:

<table>
<thead>
<tr>
<th>Real Part</th>
<th>Img Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>(8 bytes)</td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

```c
vec3_soa_t vec3_soa {
    double complex c0[sizeh];
    double complex c1[sizeh];
    double complex c2[sizeh];
}
```
**Gauge field matrices data structure**

```c
typedef struct {
    vec3_soa r0;
    vec3_soa r1;
    vec3_soa r2;
} su3_soa_t;
```

Diagram:

```
 su3_soa
 r0
  {  c0
     c1
     c2
  }  
 r1
  {  c0
     c1
     c2
  }  
 r2
  {  c0
     c1
     c2
  }  

< r0.c0[i], r0.c1[i], r0.c2[i] >
< r1.c0[i], r1.c1[i], r1.c2[i] >
< r2.c0[i], r2.c1[i], r2.c2[i] >
```
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
CUDA example for the Deo function

```c
__global__ void Deo(const __restrict su3_soa_d * const u,
                     __restrict vec3_soa_d * const out,
                     const __restrict vec3_soa_d * const in) {

    int x, y, z, t, xm, ym, zm, tm, xp, yp, zp, tp, idxh, eta;

    vec3 aux_tmp;
    vec3 aux;

    idxh = ((blockIdx.z * blockDim.z + threadIdx.z) * nxh * ny) 
           + ((blockIdx.y * blockDim.y + threadIdx.y) * nxh) 
           + (blockIdx.x * blockDim.x + threadIdx.x);

    t = (blockIdx.z * blockDim.z + threadIdx.z) / nz;
    z = (blockIdx.z * blockDim.z + threadIdx.z) % nz;
    y = (blockIdx.y * blockDim.y + threadIdx.y);
    x = 2*(blockIdx.x * blockDim.x + threadIdx.x) + ((y+z+t) & 0x1);

    ...
```
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
void Deo(const __restrict su3_soa * const u,
         __restrict vec3_soa * const out,
         const __restrict vec3_soa * const in) {

    int hx, y, z, t;

    #pragma acc kernels present(u) present(out) present(in)
    #pragma acc loop independent gang(nt)
    for(t=0; t<nt; t++) {
        #pragma acc loop independent gang(nz/DIM_BLK_Z) vector(DIM_BLK_Z)
        for(z=0; z<nz; z++) {
            #pragma acc loop independent gang(ny/DIM_BLK_Y) vector(DIM_BLK_Y)
            for(y=0; y<ny; y++) {
                #pragma acc loop independent vector(DIM_BLK_X)
                for(hx=0; hx < nhx; hx++) {

                    ...
                }
            }
        }
    }
}
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
Execution times for a $32^4$ lattice

<table>
<thead>
<tr>
<th>Block-size</th>
<th>CUDA</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,8,8</td>
<td>7.58</td>
<td>9.29</td>
</tr>
<tr>
<td>16,1,1</td>
<td>8.43</td>
<td>16.16</td>
</tr>
<tr>
<td>16,2,1</td>
<td>7.68</td>
<td>9.92</td>
</tr>
<tr>
<td>16,4,1</td>
<td>7.76</td>
<td>9.96</td>
</tr>
<tr>
<td>16,8,1</td>
<td>7.75</td>
<td>10.11</td>
</tr>
<tr>
<td>16,16,1</td>
<td>7.64</td>
<td>10.46</td>
</tr>
</tbody>
</table>

Time in [ns per site], run on an NVIDIA K20m GPU using double precision; OpenACC code compiled using PGI 14.6
## Execution times summary

<table>
<thead>
<tr>
<th>Lattice size</th>
<th>Thread Block size</th>
<th>CUDA</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16^4$</td>
<td>8x8x8</td>
<td>7.27</td>
<td>9.86</td>
</tr>
<tr>
<td>$32^4$</td>
<td>8x8x8</td>
<td>7.58</td>
<td>9.23</td>
</tr>
<tr>
<td>$48^4$</td>
<td>8x8x8</td>
<td>7.86</td>
<td>9.11</td>
</tr>
<tr>
<td>$64x32x32x16$</td>
<td>16x8x4</td>
<td>7.59</td>
<td>9.16</td>
</tr>
<tr>
<td></td>
<td>32x8x2</td>
<td>7.62</td>
<td>9.12</td>
</tr>
<tr>
<td></td>
<td>32x4x4</td>
<td>7.54</td>
<td>9.06</td>
</tr>
<tr>
<td></td>
<td>32x4x2</td>
<td>7.61</td>
<td>10.56</td>
</tr>
<tr>
<td></td>
<td>32x2x2</td>
<td>7.71</td>
<td>10.18</td>
</tr>
<tr>
<td>$32x16x16x16$</td>
<td>16x8x4</td>
<td>7.45</td>
<td>9.78</td>
</tr>
</tbody>
</table>

Time in [ns per site], run on an NVIDIA K20m GPU using double precision; OpenACC code compiled using PGI 14.6
Outline

1. Introduction
   - Hardware trends
   - Software needs
   - OpenACC at a glance

2. Towards an OpenACC LQCD implementation
   - Data layout importance
   - CUDA implementation
   - OpenACC implementation

3. Preliminary results

4. Towards multi-GPU computations
Prospective multi-GPU Lattice: $48 \times 48 \times 48 \times 96$

<table>
<thead>
<tr>
<th>Local Lat $l_x, l_y, l_z, l_t$</th>
<th>No. of GPUs</th>
<th>Block Size</th>
<th>Mem [MB]</th>
<th>Data Trans. [48B]</th>
<th>$Tc$ [ms]</th>
<th>$Td$ [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$48 \times 48 \times 48 \times 6$</td>
<td>16</td>
<td>$8 \times 8 \times 8$</td>
<td>415</td>
<td>$2 \times (48 \times 48 \times 48)$</td>
<td>6.16</td>
<td>$\approx 1.6$</td>
</tr>
<tr>
<td>$48 \times 48 \times 24 \times 12$</td>
<td>16</td>
<td>$8 \times 8 \times 8$</td>
<td>415</td>
<td>$2 \times (48 \times 48 \times 24)$</td>
<td>6.21</td>
<td>$\approx 1.8$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$+2 \times (48 \times 48 \times 12)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$48 \times 48 \times 24 \times 6$</td>
<td>32</td>
<td>$8 \times 8 \times 8$</td>
<td>208</td>
<td>$2 \times (48 \times 48 \times 24)$</td>
<td>3.30</td>
<td>$\approx 1.4$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$+2 \times (48 \times 48 \times 6)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$48 \times 24 \times 24 \times 48$</td>
<td>8</td>
<td>$8 \times 8 \times 8$</td>
<td>830</td>
<td>$4 \times (48 \times 24 \times 48)$</td>
<td>12.26</td>
<td>$\approx 2.5$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$+2 \times (48 \times 24 \times 24)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$24 \times 24 \times 24 \times 96$</td>
<td>8</td>
<td>$8 \times 8 \times 8$</td>
<td>830</td>
<td>$6 \times (24 \times 24 \times 96)$</td>
<td>17.24</td>
<td>$\approx 3.0$</td>
</tr>
<tr>
<td>$16 \times 16 \times 16 \times 96$</td>
<td>27</td>
<td>$8 \times 8 \times 8$</td>
<td>369</td>
<td>$6 \times (16 \times 16 \times 96)$</td>
<td>5.38</td>
<td>$\approx 1.8$</td>
</tr>
</tbody>
</table>

Data transfers expressed in number of fermions (i.e. 48 bytes).
Thanks for Your attention