Performance Portability with OpenMP on NVIDIA GPUs

Carlo Bertolli
Advanced Compiler Technology Team
IBM T.J. Watson Research Center
Technical Work by Gheorghe-Teodor “Doru” Bercea

Team: Kevin O’Brien, Samuel Antao, Alexandre Eichenberger, Arpith Jacob, Zehra Sura, Tong Chen, Hyojin Sung, Georgios Rokos

DOE Centers of Excellence Performance Portability Meeting
April 20 2016
Research Goals

• Obtain same GPU performance when writing CUDA and OpenMP 4
  - What is the performance of a simple porting?
  - Can I tune my application to match CUDA?

• Proxy application analysis: LULESH
  - One of five DARPA challenge problems
  - Represents code that accounts for 30% of the runtime on DoE/DoD supercomputers
  - Already ported to CUDA

• Broad Strategy
  • Look at many different kinds of applications
  • Develop optimization schemes and mechanisms for each class
  • Merge together in an optimizing compiler

• Not 1-to-1 mapping between CUDA and OpenMP 3.1 versions
  • CUDA hand-transformations:
    - Loop interchange
    - Loop fusion
    - etc..
LULESH OpeMP 4.0

• Based on OpenMP 3.1 version

```c
#pragma omp parallel for
for (Index_t i = 0 ; i < numElem ; ++i)
{
}
```

Data mapping

```c
#pragma omp target data \ 
map(to: p[:numElem], q[:numElem]) \ 
map(from: sigxx[:numElem], sigyy[:numElem]) \ 
map(from: sigzz[:numElem])
{
#pragma omp target teams distribute parallel for
for (Index_t i = 0 ; i < numElem ; ++i)
{
}
```

• **target**: offload to GPUs
• **teams**: use many CUDA blocks
• **parallel**: use many CUDA threads
• **distribute** and **for**: block loop and schedule to blocks and threads
Basic OpenMP Implementation on NVIDIA GPUs

• Challenge: any OpenMP construct may be used within a target region.

• This includes arbitrary sequences of sequential and parallel regions, tasks, locks, etc.

• General implementation scheme: **control loop**

```c
#pragma omp target
{
  if(a[0]++ > K || b[1]++ < L) {
    #pragma omp parallel for
    for(int i = 0 ; i < K ; i++) {
      if(omp_get_thread_num() > 2) {
        #pragma omp simd
        for(int j = 0 ; j < L ; L++) { <S1> }
      } else {
        #pragma omp simd
        for(int j = 0 ; j < L ; L++) { <S2> }
      }
    } else {
      #pragma omp parallel for
      for(int i = 0 ; i < K ; i++) { <S3> }
    }
  }
}
```
Basic OpenMP Implementation on NVIDIA GPUs

- Challenge: any OpenMP construct may be used within a target region
- This includes arbitrary sequences of sequential and parallel regions, tasks, locks, etc.
- General implementation scheme: control loop
- Ease of integration into clang without rewriting entire C/C++ implementation is also a constraint

```c
nextState = SQ1;
while(!finished) {
    switch(nextState) {
        case SQ1:
            if(tid > 0) break;
            // sequential reg. 1
            nextState = PR1;
            break;
        case PR1:
            if(tid > 4) break;
            // parallel reg. 1
            if (tid == 0) nextState = SQ2;
            break;
        case SQ2:
            if(tid > 0) break;
            // sequential reg. 2
            finished = true;
            break;
    }
    __syncthreads();
}
```
# First Naive Runs

<table>
<thead>
<tr>
<th>Cuda Kernel Region</th>
<th>CUDA Runtime (usec)</th>
<th>Control Loop Runtime (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration Calculation</td>
<td>3.2</td>
<td>712</td>
</tr>
<tr>
<td>Apply Boundary Acceleration</td>
<td>5.1</td>
<td>279</td>
</tr>
<tr>
<td>Position and Velocity Calculation</td>
<td>3.2</td>
<td>775</td>
</tr>
<tr>
<td>Kinematics and Monotonic Gradient Calculation</td>
<td>17</td>
<td>608</td>
</tr>
<tr>
<td>Monotonic Region Calculation</td>
<td>11</td>
<td>2546</td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>1913</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3760</td>
</tr>
<tr>
<td></td>
<td></td>
<td>509</td>
</tr>
<tr>
<td></td>
<td></td>
<td>619</td>
</tr>
<tr>
<td></td>
<td></td>
<td>544</td>
</tr>
</tbody>
</table>
Two Missing Important Bits

• Uncoalesced Accesses:
  - By default, OpenMP schedules loops by contiguous chunks
  - Change default to schedule(static,1) assigns successive iterations to successive threads within same blocks

• Tuning of number of blocks and block size per kernel
### After First Tuning

<table>
<thead>
<tr>
<th>Cuda Kernel Region</th>
<th>CUDA Runtime (usec)</th>
<th>Control Loop Runtime (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration Calculation</td>
<td>3.2</td>
<td>55</td>
</tr>
<tr>
<td>Apply Boundary Acceleration</td>
<td>5.1</td>
<td>43</td>
</tr>
<tr>
<td>Position and Velocity Calculation</td>
<td>3.2</td>
<td>54</td>
</tr>
<tr>
<td>Kinematics and Monotonic Gradient Calculation</td>
<td>17</td>
<td>511</td>
</tr>
<tr>
<td>Monotonic Region Calculation</td>
<td>11</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td></td>
<td>140</td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td>529</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>
Occupancy / Register Allocation

• Many reasons:
  - A while loop with a switch inside may hit hard register allocation
  - In OpenMP 4.0 kernel parameters are passed as pointer to pointer
    ‣ The kernel is allowed to do pointer arithmetic
    ‣ This results in an additional register allocated for each parameter
    ‣ Fixed by OpenMP 4.5 firstprivate-related rules
  - NVCC and LLVM backends for NVPTX are different:
    ‣ nvcc uses libnvvm, which is shipped as a library
    ‣ LLVM uses the open source code in the trunk
    ‣ Different optimization strategies
#pragma omp target teams distribute parallel for schedule(static,1) \
for( Index_t gnode=0 ; gnode<numNode ; ++gnode ) 
{
}

for (int i = threadIdx.x + blockIdx.x * blockDim.x; 
     i < n; i += blockDim.x * gridDim.x) {
    g_node = i;

    // codegen loop body
}

1-to-1 mapping of CUDA grid to iteration space

CUDA-style notation

Compiler:
• Detect pragma combination
• Prove absence of nested pragmas
• Prove absence of function calls
### Performance of combined Construct

<table>
<thead>
<tr>
<th>Cuda Kernel Region</th>
<th>CUDA Runtime (usec)</th>
<th>Control Loop Runtime (usec)</th>
<th>% diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration Calculation</td>
<td>3.2</td>
<td>4.3</td>
<td>35%</td>
</tr>
<tr>
<td>Apply Boundary Acceleration</td>
<td>5.1</td>
<td>4.8</td>
<td>-6%</td>
</tr>
<tr>
<td>Position and Velocity Calculation</td>
<td>3.2</td>
<td>4.8</td>
<td>178%</td>
</tr>
<tr>
<td>Kinematics and Monotonic Gradient Calculation</td>
<td>17</td>
<td>6.5</td>
<td>514%</td>
</tr>
<tr>
<td>Monotonic Region Calculation</td>
<td>11</td>
<td>15</td>
<td>36%</td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>3</td>
<td>247%</td>
</tr>
</tbody>
</table>
Small Kernels: Acceleration Calculation

![Acceleration Calculation Runtime](image)

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>CUDA Blocks</th>
<th>Time (µs)</th>
<th>OpenMP 4.0 (Control) Teams</th>
<th>Time (µs)</th>
<th>OpenMP 4.0 (Combined) Teams</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$12^3$</td>
<td>(*)</td>
<td>3.264</td>
<td>32</td>
<td>32.512</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>$30^3$</td>
<td>(*)</td>
<td>8.224</td>
<td>64</td>
<td>49.087</td>
<td>128</td>
<td>8.32</td>
</tr>
<tr>
<td>$100^3$</td>
<td>(*)</td>
<td>304.45</td>
<td>128</td>
<td>567.29</td>
<td>1024</td>
<td>318.4</td>
</tr>
</tbody>
</table>
Large Kernels

- Reduce loop count in OpenMP 3.0 (45) to about 18 in OpenMP 4.0 (simple fusion)
- Direct correspondences with CUDA still hard to come by for complex loops.
- Ideal candidate: loops applying material properties

<table>
<thead>
<tr>
<th>Cuda Kernel Region</th>
<th>CUDA Runtime (usec)</th>
<th>Control Loop Runtime (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>314</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.1</td>
</tr>
<tr>
<td>1. Fuse Loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>525.6</td>
</tr>
<tr>
<td>2. Reorder values to have fewer divergent warps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>466.3</td>
</tr>
<tr>
<td>3. Loop over cells instead of regions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>102.8</td>
</tr>
</tbody>
</table>
## Performance of Combined Construct

<table>
<thead>
<tr>
<th>Cuda Kernel Region</th>
<th>CUDA Runtime (usec)</th>
<th>Control Loop Runtime (usec)</th>
<th>% diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration Calculation</td>
<td>3.2</td>
<td>4.3</td>
<td>35%</td>
</tr>
<tr>
<td>Apply Boundary Acceleration</td>
<td>5.1</td>
<td>4.8</td>
<td>-6%</td>
</tr>
<tr>
<td>Position and Velocity Calculation</td>
<td>3.2</td>
<td>4.1</td>
<td>178%</td>
</tr>
<tr>
<td>Kinematics and Monotonic Gradient Calculation</td>
<td>17</td>
<td>58</td>
<td>514%</td>
</tr>
<tr>
<td>Monotonic Region Calculation</td>
<td>11</td>
<td>15</td>
<td>36%</td>
</tr>
<tr>
<td>Apply Material Properties to Regions</td>
<td>92</td>
<td>102.8</td>
<td>11%</td>
</tr>
</tbody>
</table>
### PTXAS Report

<table>
<thead>
<tr>
<th>Kernel ID</th>
<th>Kernel Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>K13</td>
<td>CalcKinematicsForElems</td>
</tr>
<tr>
<td>K14</td>
<td>CalcMonotonicQGradientsForElems</td>
</tr>
</tbody>
</table>

#### Register usage, stack frame and spill sizes

<table>
<thead>
<tr>
<th></th>
<th>K8</th>
<th>K9</th>
<th>K10</th>
<th>K11</th>
<th>K12</th>
<th>K13</th>
<th>K14</th>
<th>K15</th>
<th>K16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>38</td>
<td>29</td>
<td>48</td>
<td>32</td>
<td>40</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Stack frame (B)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>840</td>
<td>592</td>
<td>168</td>
<td>232</td>
</tr>
<tr>
<td>Load Spills (B)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>320</td>
<td>956</td>
<td>164</td>
<td>404</td>
</tr>
<tr>
<td>Store Spills (B)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>304</td>
<td>644</td>
<td>172</td>
<td>328</td>
</tr>
</tbody>
</table>
Conclusion

• Good performance can be achieved for simpler kernels
  - Requires optimized compiler synthesis
  - How many patterns do we need?
• More complex kernels may require hand tuning over baseline
  - Register allocation figure is of paramount importance
  - Use libnvvm’s code synthesis to improve register allocation?
  - Other factors like coalescing may play a relevant role in a “bad register allocation” situation
Fallback
# Large Loop Performance by Problem Size

## Performance for Applying Material Properties to Regions

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>CUDA</th>
<th>OpenMP 4.0 (Control)</th>
<th>OpenMP 4.0 (Combined)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocks</td>
<td>Threads</td>
<td>Time(\mu s)</td>
</tr>
<tr>
<td>12^3</td>
<td>(*)</td>
<td>128</td>
<td>92.928</td>
</tr>
<tr>
<td>30^3</td>
<td>(*)</td>
<td>128</td>
<td>113.47</td>
</tr>
<tr>
<td>100^3</td>
<td>(*)</td>
<td>128</td>
<td>2015.8</td>
</tr>
</tbody>
</table>
Background

• Work by IBM’s Advanced Compiler Technology team

• OpenMP 4.0 implementation based on Clang/LLVM® compilation toolchain

• Targets node with IBM® Power® processors plus Nvidia® GPUs
  - All tests on IBM 8247-42L system: Power 8 + Kepler K40m

• All tools available as open source

• IBM Proprietary OpenMP optimized implementation through Lightweight OpenMP library (LOMP)
  - Lomp only available for OpenPower nodes and other IBM processors

• Ongoing implementation, transitioning to OpenMP 4.5
  - Beta OpenMP 4.5 will be available to DoE Labs around mid-April