An Experimental Study on Performance Portability of OpenCL Kernels

☑️ Sean Rul
☐ Hans Vandierendonck
☐ Joris D’Haene
☐ Koen De Bosschere

SAAHPC’10
Knoxville, Tennessee
July 13-15, 2010
Programming Accelerators: a Tower of Babel?

- Cell SDK
- SSE
- CUDA
- Brook+
- OpenCL
- AMD
- NVIDIA
- KHRONOS GROUP
- Core i7
- Phenom II
- TESLA
OpenCL: One ring to bind them all?

‘The open standard for parallel programming of heterogeneous systems’

Functional portability

What about performance portability?

What is the specificity of code optimizations to the underlying architecture?

What is the performance impact?
Method

Parboil Benchmark suite

Architectures

Code optimizations
## Method: Hardware Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Clock(MHz)</th>
<th>Cores (Threads)</th>
<th>GFLOPs</th>
<th>Bandwidth(GB/s)</th>
<th>TDP (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7-720QM</td>
<td>1600</td>
<td>4 (8)</td>
<td>25</td>
<td>21</td>
<td>45</td>
</tr>
<tr>
<td>FirePro V8700</td>
<td>750</td>
<td>10(160)</td>
<td>816</td>
<td>108</td>
<td>114</td>
</tr>
<tr>
<td>Tesla c1060</td>
<td>1300</td>
<td>30 (240)</td>
<td>933</td>
<td>102</td>
<td>187</td>
</tr>
<tr>
<td>Cell SPE</td>
<td>3200</td>
<td>6 (6)</td>
<td>150</td>
<td>25</td>
<td>45</td>
</tr>
</tbody>
</table>
## Method: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CUDA→OpenCL</th>
<th>Parameterized</th>
<th>nVidia GPU</th>
<th>x86 CPU</th>
<th>Ati GPU</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cp</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Mri-fhd</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Mri-q</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Rpes</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Tpacf</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Pns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sad</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Too small
Method: Code Optimizations

Loop Unrolling

Vectorization

Single Vector instruction
ai R1, R2, 1

\[ R_1 \begin{bmatrix} A & B & C & D \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ R_2 \begin{bmatrix} A' & B' & C' & D' \end{bmatrix} \]

Thread Block Size
Translating the Parboil benchmark suite to OpenCL

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mri-fhd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mri-q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rpes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tpacef</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Bar chart showing execution time for different benchmarks and platforms (CUDA, OpenCL). The chart includes bars for Other Time, Compile Time, Copy Time, and Device time.](image-url)
Translating the Parboil benchmark suite to OpenCL

![Graph showing execution times for different benchmarks in CUDA and OpenCL]

- OpenCL kernel compiled at execution time
- Other Time
- Compile Time
- Copy Time
- Device time

Benchmark:
- cp
- mri-fhd
- mri-q
- rpes
- tpacf

Execution times in seconds for each benchmark under CUDA and OpenCL are depicted in the graph.
Translating the Parboil benchmark suite to OpenCL

Specific CUDA functions allow faster gonio exec

Execution time (sec)

Other Time
Compile Time
Copy Time
Device time

CUDA
OpenCL
CUDA
OpenCL
CUDA
OpenCL
CUDA
OpenCL
CUDA
OpenCL

cp
mri-fhd
mri-q
rpes
tpacf
Performance Impact of Loop Unrolling and Vectorization

Benchmark: cp
Block size Cell: 1
Block size others: 128
Performance Impact of Loop Unrolling and Vectorization

CPU benefits from vectorization
CPU is indifferent to loop unrolling

Benchmark: cp
Block size Cell: 1
Block size others: 128
Performance Impact of Loop Unrolling and Vectorization

Vectorization is critical to Cell
Cell benefits from loop unrolling

Benchmark: cp
Block size Cell: 1
Block size others: 128
Performance Impact of Loop Unrolling and Vectorization

Optimizations interact with each other

FirePro more sensitive

Too much unrolling degrades performance

Benchmark: cp
Block size Cell: 1
Block size others: 128
Performance Impact of Loop Unrolling and Vectorization

Vectorization is critical to Cell

Cell benefits from loop unrolling

Benchmark: cp

Benchmark: mri-fhd
Performance Impact of Thread Block Size

Benchmark: mri-fhd
Loop unrolling: 2
Vectorization: yes
Performance Impact of Thread Block Size

Thread block size has little to no effect on CPU

Benchmark: mri-fhd
Loop unrolling: 2
Vectorization: yes
Performance Impact of Thread Block Size

Execution time (secs) vs. Block size for CPU, Tesla, FirePro, and Cell. Thread block size is not adjustable for the Cell.

Benchmark: mri-fhd
Loop unrolling: 2
Vectorization: yes
Performance Impact of Thread Block Size

Thread block size is the most important parameter for Tesla

Benchmark: mri-fhd
Loop unrolling: 2
Vectorization: yes
# Differences in Execution Time

<table>
<thead>
<tr>
<th>Bench-mark</th>
<th>Platform</th>
<th>CPU i7</th>
<th>Tesla</th>
<th>FirePro</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU i7</strong></td>
<td></td>
<td><em>12.62</em></td>
<td><em>50.50</em></td>
<td><em>12.62</em></td>
<td><em>3,11</em></td>
</tr>
<tr>
<td><strong>Tesla</strong></td>
<td></td>
<td><em>2.49</em></td>
<td><em>0.48</em></td>
<td><em>0.49</em></td>
<td><em>2.49</em></td>
</tr>
<tr>
<td><strong>FirePro</strong></td>
<td></td>
<td><em>164.60</em></td>
<td><em>1.83</em></td>
<td><em>1.71</em></td>
<td><em>164.60</em></td>
</tr>
<tr>
<td><strong>Cell SPE</strong></td>
<td></td>
<td><em>8.80</em></td>
<td><em>39.75</em></td>
<td><em>49.49</em></td>
<td><em>8.80</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bench-mark</th>
<th>Platform</th>
<th>CPU i7</th>
<th>Tesla</th>
<th>FirePro</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mri-fhd</strong></td>
<td></td>
<td><em>2.90</em></td>
<td><em>2.93</em></td>
<td><em>4.66</em></td>
<td><em>3.11</em></td>
</tr>
<tr>
<td><strong>Tesla</strong></td>
<td></td>
<td><em>0.38</em></td>
<td><em>0.31</em></td>
<td><em>0.49</em></td>
<td><em>0.81</em></td>
</tr>
<tr>
<td><strong>FirePro</strong></td>
<td></td>
<td><em>1.84</em></td>
<td><em>1.30</em></td>
<td><em>1.10</em></td>
<td>n/a</td>
</tr>
<tr>
<td><strong>Cell SPE</strong></td>
<td></td>
<td><em>1.20</em></td>
<td><em>1.26</em></td>
<td><em>10.22</em></td>
<td><em>1.14</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bench-mark</th>
<th>Platform</th>
<th>CPU i7</th>
<th>Tesla</th>
<th>FirePro</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mri-q</strong></td>
<td></td>
<td><em>5.78</em></td>
<td><em>5.99</em></td>
<td><em>8.74</em></td>
<td><em>6.22</em></td>
</tr>
<tr>
<td><strong>Tesla</strong></td>
<td></td>
<td><em>0.80</em></td>
<td><em>0.77</em></td>
<td><em>1.15</em></td>
<td><em>1.87</em></td>
</tr>
<tr>
<td><strong>FirePro</strong></td>
<td></td>
<td><em>1.49</em></td>
<td><em>1.98</em></td>
<td><em>1.25</em></td>
<td>n/a</td>
</tr>
<tr>
<td><strong>Cell SPE</strong></td>
<td></td>
<td><em>2.28</em></td>
<td><em>2.08</em></td>
<td><em>24.00</em></td>
<td><em>1.91</em></td>
</tr>
</tbody>
</table>

* x4 slower
* x96 slower
How much performance is exploited?

Percentage of Peak Performance

- CPU i7
- Tesla
- FirePro
- Cell

- cp
- mri-fhd
- mri-q
What have we learned from this?

**Functional portability: almost a free lunch**
- Thread block size compatibility
- Use of `__constant` pointer
- Endianness of data

**Performance portability: no free lunch**
- Architecture specific optimizations
- Program specific sensitivity
- Interaction between optimizations
Conclusion

• OpenCL can deliver functional portability, but no performance portability
• Developer should consider optimizations relevant to all target architectures
• Auto-tuning is faced with a large search space
Likewise, the heterogeneity of computing architectures might mean that a particular loop construct might execute at an acceptable speed on the CPU but very poorly on a GPU, for example. CPUs are designed in general to work well on latency sensitive algorithms on single threaded tasks, whereas common GPUs may encounter extremely long latencies, potentially orders of magnitude worse. A developer interested in writing portable code may find that it is necessary to test his design on a diversity of hardware designs to make sure that key algorithms are structured in a way that works well on a diversity of hardware. We suggest favoring more work-items over fewer. It is anticipated that over the coming months and years experience will produce a set of best practices that will help foster a uniformly favorable experience on a diversity of computing devices.