A PORTABLE BENCHMARK SUITE FOR HIGHLY PARALLEL DATA INTENSIVE QUERY PROCESSING

Ifrah Saeed, Sudhakar Yalamanchili, School of ECE
Jeff Young, School of Computer Science

February 8, 2015
The Need for Accelerated Data Warehousing

Data Warehousing has become a large part of supply chain operations

Analytics of weekly and monthly trends helps to predict future supply needs and ordering patterns

- How many people will buy grills around July 4th?

The explosion of Big Data makes this analytics tougher

New hardware like GPU and Phi accelerators can be used to accelerate queries for data warehousing applications with large amounts of data

- Co-processing with GPUs can provide 2-27x speedup [1]

Our work focuses on mapping a data warehousing benchmark, TPC-H, to a portable accelerator language, OpenCL

Related Work

Currently, there is little work in the area of data analytics on accelerators and no accelerator-based analytics benchmarks

- OmniDB: Kernel-adapter design that uses OpenCL operators as part of larger framework; unclear as to current project status [2]
- Work has also focused on portable database primitives from a software engineering standpoint [3]
- Companies like Map-D are focusing on CUDA-based analytics using SQL queries [4]

Related Work: Red Fox [5]

Our OpenCL primitives grew out of this GPU-focused project

Red Fox is a collaborative project with LogicBlox that has focused on CUDA implementations of the TPC-H queries using relational algebra (RA).

OpenCL primitives build off the CUDA primitives

- Existing primitives have “GPU slant” – vectorization and testing geared towards Fermi-class GPUs
- Red Fox work demonstrates a path forward for full OpenCL implementation of TPC-H

Contributions of this work

• Portable database relational algebra primitives using OpenCL for cross-platform compatibility

• A new open-source benchmark that allows these primitives to be run on a variety of systems (extensions for SHOC)

• Evaluation of these primitives and related microbenchmarks on multiple hardware platforms – Intel and AMD CPUs, integrated and discrete GPUs, and Xeon Phi

• An eventual path towards a fully portable, accelerated implementation of the standard data warehousing benchmark, TPC-H [6]

TPC-H Benchmark Suite

Consists of 21 queries meant to represent common data warehousing operations

Benchmark results typically report on the capabilities of a particular hardware system and database setup.

Accelerated versions of TPC-H are complex

Previous Red Fox implementations of queries required many CUDA kernels – the simplest query requires ~15 CUDA kernels and an accompanying scheduler

- For this reason, our work focuses on OpenCL primitives first

Q1: Pricing Summary Report Query: returns a price summary of all items shipped within a certain date range
Scalable HeterOgeneous Computing (SHOC) Suite

Accelerator-based benchmark suite that provides benchmarks written in multiple languages [8]

• Designed as a tool to compare algorithms across software platforms but also to compare hardware systems

• OpenCL, CUDA, Phi (OpenMP), and OpenACC variants include “speeds and feeds” benchmarks as well as parallel benchmarks

Currently there is a focus to add more “Big Data” benchmarks to represent non-scientific workloads

• TPC-H primitives and queries are a good candidate along with ML and graph algorithms

This talk focuses on project, select, and join primitives; see [7] for others.

Microbenchmarks A (Chained Select), B (Chained Join), C (Select, Join, Project) represent patterns common in TPC-H queries.

Basic Design of Primitives

Partition, compute, gather
Values are stored as an array of tuples with key-value pairs

Project:
• Partition, compute, and gather are all combined into one kernel

Select:
• Partition and compute are combined into “Selection” kernel; separate gather phase

Join:
• Find Bounds kernel is part of partition phase, separate compute and gather stages implemented by different kernels
Select Primitive

<table>
<thead>
<tr>
<th>Key</th>
<th>0x1235</th>
<th>0x1462</th>
<th>0x1172</th>
<th>0x1903</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0xAB23</td>
<td>0xD3F2</td>
<td>0x7213</td>
<td>0x8931</td>
</tr>
</tbody>
</table>

Select < 0xA000

- Selection
- Prefix Sum
- Prefix Sum
- Sum
- Gather

Find position of intermediate output
Find position of final output
Sum number of outputs in histogram
Migrate local results from shared to global memory

```c
foreach partition p in parallel do
    foreach work-item w (with local id lw) in parallel do
        keyReg ← input[w].key;
        valueReg ← input[w].value;
        countReg ← 0;
        if keyReg < THRESHOLD then
            countReg ← 1;
        end
        indexReg ← positions of selected tuples obtained from Algorithm 3;
        totalReg ← number of selected tuples obtained from Algorithm 3;
        if count is equal to 1 then
            local[indexReg].key ← keyReg;
            local[indexReg].value ← valueReg;
        end
        if lw < totalReg then
            globalPartition[lw] ← local[lw];
        end
        if lw is 0 then
            array[p] ← total;
        end
    end
end
```
Join Primitive

<table>
<thead>
<tr>
<th>Key</th>
<th>0x1235</th>
<th>0x1462</th>
<th>0x1172</th>
<th>0x1903</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0xAB23</td>
<td>0xD3F2</td>
<td>0x7213</td>
<td>0x8931</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>0x1172</th>
<th>0x1235</th>
<th>0x1820</th>
<th>0x1903</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0xB723</td>
<td>0x0342</td>
<td>0x6418</td>
<td>0xC298</td>
</tr>
</tbody>
</table>

- **Join (L0:L3, R0:R4)**
- Find sizes of input arrays and estimate output array size
- Find position of intermediate output
- Find position of final output
- Sum number of outputs in histogram
- Migrate local results from shared to global memory

```
foreach partition p (left and corresponding right) in parallel do
  while one of both left and right partitions not exhausted do
    foreach work-item w (with local id bw) in parallel do
      leftLocal[bw] ← left[w];
      rightLocal[bw] ← right[w];
      if the last tuple key of leftLocal < the first tuple key of rightLocal then
        go to the end of leftLocal;
      else
        if the last tuple key of rightLocal < the first tuple key of leftLocal then
          go to the end of the rightLocal;
        else
          Algorithm 6;
        end
      end
    end
  end
end
```

Algorithm 5: INNER JOIN

```
rightReg ← right[bw];
lowerReg ← lowerBound for rightReg.key in leftLocal;
upperReg ← upperBound for rightReg.key in leftLocal;
foundCountReg ← upperReg − lowerReg;
indexReg = positions of selected tuples obtained from Algorithm 3;
totalReg = number of selected tuples obtained from Algorithm 3;
for i ← 0 to foundCountReg do
  outputLocal[indexReg[i]] ← rightReg and matching left tuple;
end
output ← outputLocal;
else
  put in multiple iterations from outputLocal to output;
end
```

Algorithm 6: JOIN Block
Experimental Test bed

<table>
<thead>
<tr>
<th>Platform</th>
<th>CPU</th>
<th>Accelerator</th>
<th>Device Memory</th>
<th>OS and Software</th>
<th>OpenCL Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Trinity APU</td>
<td>A10-5800K</td>
<td>HD 7660D</td>
<td>16 GB DDR3</td>
<td>CentOS 7.0</td>
<td>AMD APP 2.9</td>
</tr>
<tr>
<td>Intel Ivy Bridge</td>
<td>i5-3470</td>
<td>HD 2500</td>
<td>16 GB DDR3</td>
<td>Ubuntu 14.04</td>
<td>Intel OpenCL 14.2</td>
</tr>
<tr>
<td>Intel Sandy Bridge</td>
<td>2xE5-2670</td>
<td>Phi 5110</td>
<td>24 GB DDR3, 8 GB DDR</td>
<td>CentOS 6.2, gcc 4.8.2</td>
<td>SDK for Applications and Beignet 1.0</td>
</tr>
<tr>
<td>Intel Haswell</td>
<td>i7-4770</td>
<td>GT2</td>
<td>16 GB DDR3</td>
<td>Ubuntu 14.04, gcc 4.8.2</td>
<td></td>
</tr>
<tr>
<td>Nvidia Fermi</td>
<td>Xeon X5660</td>
<td>M2090</td>
<td>6 GB</td>
<td>CentOS 6.2, gcc 4.8.2</td>
<td></td>
</tr>
<tr>
<td>Nvidia Kepler</td>
<td>Xeon E5520</td>
<td>K40</td>
<td>6 GB</td>
<td>CentOS 6.4, gcc 4.8.2</td>
<td>CUDA 6.0</td>
</tr>
</tbody>
</table>

OpenCL 1.2 used because vendor implementations vary

- AMD, Intel support OpenCL 2.0 to a reasonable degree; NVIDIA supports 1.2; Intel discrete GPUs only supported on Linux by “Beignet”

Intel OCL latest version has an issue with vectorizing functionality – this resulted in disabled optimizations for Phi and CPU platforms

- Beignet is unaffected
Select Total (Compute and Data)

Total time for 256 MB select operation ranges from 95 ms (M2090) to 854 ms (Trinity CPU)
Select Kernel Accelerators (Compute)

Integrated GPUs complete 256 MB Select compute in less than 215 ms
- NVIDIA GPUs and AMD Trinity likely benefit from implicit 256 workgroup size
- Xeon Phi may be penalized by lack of vectorization optimizations
Sandy Bridge compute takes just 60 ms compared to total runtime (with data transfer) of 225 ms. 
• This Xeon CPU has higher clock rates, more threads (16), and more cache than other tested CPUs
As expected, data transfer consumes a large amount of execution time

- 165 ms out of 225 ms runtime on Sandy Bridge (74.7%); 48 ms out of 132 on Haswell (31.8%)
- Lower data transfer costs on Ivy Bridge and Haswell GPU are likely due to zero-copy schemes not used for CPU
Select kernel consumes an increasing portion of kernel runtime
• As described earlier, partitioning and compute were placed into one kernel – good place for future optimization
Project Kernel

Project kernel is highly parallel operation – just 1 kernel, no data dependencies

- Discrete GPUs and highly multithreaded architectures (SNB and Xeon Phi) perform best
- 10.6 ms for 1 GB project on K20m; 15.4 ms for 512 MB on Phi; 89 ms for 512 MB on Trinity
- However, total times for 512 MB project range from 139 ms (Haswell CPU) to 336 ms (SNB) with data transfer
Join Kernel

8.5 ms to 95.7 ms for 2x32 MB join operation
• Workgroup size of 256 (good for GPU, APU) unfairly penalizes Xeon Phi; Phi runs at lower clock speed than CPUs and depends heavily on vectorization for performance
A Microbenchmark (Compute + Data)

Results mirror select very closely – total runtime of 101 ms (M2090) to 891 ms (Trinity CPU – not shown)

- Subsequent selects operate on device-local data and each iteration, $i$, operates on $0.5i_{-1}$ input size
Chained join tracks single Join results linearly due to sequential operations
• 24 ms to 192 ms for 2x32 MB joins
3x64 MB input sets take from less than 20 ms to 1.88 seconds to perform select, join, and project

- Join is the most limiting kernel for Phi performance
Lessons Learned

Common language != optimized code for each platform

• Vendor differences, tuning of code for GPU test platform, bugs in implementations all contribute to widely varied performance across platforms

Architecture trends require further study

• Even in our limited tests, Sandy Bridge compute time was surprisingly low while Xeon Phi was surprisingly slow
• Our speculation is that lack of support for large numbers of work-items and limited vectorization opportunities limited the Phi

Data transfer costs still dominate, especially for small input sets

• In our tests, discrete GPU compute was fastest and data transfer was also relatively low
• However, improved zero-copy semantics make integrated GPUs more appealing for small queries or sub-queries
Future Work

Not just **device portability** but **performance portability**

- Needs more profiling!
- Support workgroup sizes specific to each device
- Results demonstrated that initial GPU-focused design limited performance on other platforms

Retest with latest vendor OpenCL stacks

Use primitives to implement full set of TPC-H queries

Investigate scheduling decisions for larger data sets – at what point is crossover from integrated to discrete accelerators worth it?
More Information

**Ifrah Saeed’s Masters Thesis [7]**
More detail on implementation of discussed primitives and all 11 primitives and operators

**Red Fox paper [5]**
CUDA implementation of TPC-H queries

**SHOC alpha release of these benchmarks**
www.github.com/jyoung3131/shoc
Still under development, so please feel free to email me if (when) you find bugs!

---

Questions?

Ifrahsaeed@gatech.edu, jyoung9@gatech.edu

Special thanks to NVIDIA and Dr. Jeff Vetter of ORNL for the use of GPUs and other accelerators used in this evaluation.