A Portable Benchmark Suite for Highly Parallel Data Intensive Query Processing

Ifrah Saeed
Georgia Institute of Technology, USA
IfrahSaeed@gatech.edu

Jeffrey Young
Georgia Institute of Technology, USA
jyoung9@gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology, USA
sudha@gatech.edu

Abstract

Traditionally, data warehousing workloads have been processed using CPU-focused clusters, such as those that make up the bulk of available machines in Amazon's EC2, and the focus on improving analytics performance has been to utilize a homogenous, multithreaded CPU environment with optimized algorithms for this infrastructure. The increasing availability of highly parallel accelerators, like the GPU and Xeon Phi discrete accelerators, in these types of clusters has provided an opportunity to further accelerate analytics operations but at a high programming cost due to optimizations required to fully utilize each of these new pieces of hardware.

This work describes and analyzes highly parallel relational algebra primitives that are developed to focus on data warehousing queries through the use of a common OpenCL framework that can be executed both on standard multi-threaded processors and on emerging accelerator architectures. As part of this work, we propose a set of data-intensive benchmarks to help compare and differentiate the performance of accelerator hardware and to determine the key characteristics for efficiently running data warehousing queries on accelerators.

Categories and Subject Descriptors C.1.3 [*Re*]: Other Architecture Styles— Heterogeneous (hybrid) systems; D.2.8 [*Software Engineering*]: Metrics—performance measures; H.2.4 [*Database Management*]: Systems—relational databases

General Terms Algorithms, Measurement, Performance

Keywords Heterogeneous Computing, Relational Databases, TPC-H, Benchmarking, SHOC, OpenCL

1. Introduction

As traditional relational databases typically handle short queries and small amounts of data, they are useful for simple transactions but are not always best suited for processing complex queries in data-intensive applications [8]. The need to handle big data in such applications prompted the creation of data warehouses. Such mass storage facilities are playing an increasingly important role in current industrial applications. Modern enterprises use various data warehousing applications for collecting, managing, analyzing, and disseminating big data [6]. The emergence of big data in sensor technology, retail and inventory transactions, social media, computer vision, and many other fields has led to the establishment of warehouses comprised of on-line analytical processing (OLAP) systems that handle large transactions and complex queries. A significant portion of data warehousing applications comprises relational queries that, in turn, are based on a mix of arithmetic and relational algebra (RA) operators. Since they process a substantial amount of data, these operators should be capable of exhibiting massive degrees of parallelism. To accelerate these operators and applications, modern graphics processing units (GPUs), coprocessors, and other accelerators with a highly parallel structure and high memory bandwidth are attractive. However, these operators exhibit highly unstructured and irregular parallelism and perform very few operations per byte. Therefore, unlike other traditional scientific operations and computations that effectively utilize parallelism, the efficient parallel implementation of these operators is a challenge.

To address these challenges, we have built on previous relational algebra implementations that were targeted solely towards one type of accelerator, NVIDIA GPUs. By using a common accelerator API, OpenCL, we can map efficient implementations of relational algebra operators to CPUs, GPUs, and possibly eventually even to devices like FPGAs that support some OpenCL functionality [4].

This paper makes the following contributions:

- Provides efficient database primitives using OpenCL. These primitives include relational algebra (RA), arithmetic, and other operators required to execute relational queries in data warehousing applications. These primitives are also used to implement micro-benchmarks derived from the TPC-H industry standard benchmark suite [3].
- Evaluates these database primitives and micro-benchmarks across multiple accelerators.
- Provides a new open-source benchmark for evaluating accelerators for database operations as part of the SHOC benchmark suite.

2. Related Work

Traditional enterprise-focused benchmark suites like BigDataBench [14] and TPC-H [3] have traditionally focused on non-accelerated systems. However, rapid hardware development has resulted in diverse accelerators and parallel CPU/GPU architectures giving rise to the need of having a benchmark suite that can evaluate portable database operators written in OpenCL across different architectures. [17] proposes a portable query processor using kernel-adapter based design. This design makes the kernel aware of the underlying architecture for optimization of database primitives across different architectures. The paper only presents basic results on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the author/owner(s). Publication rights licensed to ACM.

PPAA'15, February 8, 2015, San Francisco, CA, USA
ACM 978-1-4503-3405-1/15/02
http://dx.doi.org/10.1145/2726935.2726943

portability and efficiency. [1] also implements portable database primitives but focuses on using software engineering for efficiency. The paper claims that software engineering is a better solution than [17] because of its low implementation and maintenance cost. Both of these projects are in their preliminary stages and have proposed visions to improve the efficiency of the portable code for the respective architectures, but they have not yet addressed concrete ideas as to how to implement relational database primitives for an application similar to the style of analytics addressed by the TPC-H benchmark. However, these general principles could be incorporated for optimization purposes in future work. As opposed to previous general implementations of underlying kernels like scan like in [1], the goal of this paper is to provide the algorithm design of the database primitives and to demonstrate the importance of having a benchmark suite for databases that evaluate different accelerators.

3. Design of the Primitives

Table 1. Relational Algebra Primitives

Primitive Name	Input Tuple Size	Primitive Name	Input Tuple Size
Project	1	Add	2
Reduce	1	Subtract	2
Reduce by Key	1	Multiply	2
Select	1	Difference	2
Unique	1	Product	2
Inner Join	2		

This section describes the high-level algorithmic structure of several of the primitives in the library, all of which are shown in Table 1. For space we focus on a few of the most interesting primitives in this paper (bolded in Table 1), but algorithms for the other primitives are described in further detail in [12]. The relational algebra algorithms have a structure very similar to those described in [7], and all of the primitives are either unary or binary (single or two inputs) operators. The main differences are changes in the algorithms that optimize them for both CPUs and GPUs, and the challenges faced in translating these algorithms into a different language. In [7], Diamos et al. presented the implementation of primitives in CUDA optimized for Nvidia GPUs. Accounting for multiple underlying architectures, we wrote our algorithms in OpenCL. Moreover, our library contains not only the basic relational database operators described in [7] but also the additional primitives such as aggregation, required to execute basic relational queries. Most of the primitives in the library are implemented using the same sequence of stages: *partition, compute, gather*. Note that our relations are stored as a densely packed array of tuples, each of which comprises two attributes, a key and a value. Each tuple supports one key and up to three value attributes (up to 256 bits). Many of our algorithms that require complex partitioning store the input relations and maintain results in sorted form because operations such as array/vector partitioning and tuple lookup are efficient with the sorted array/vector. Because of this sorted property, algorithms are executed in an efficient manner on multi-core and many-core processors. The overview of primitives algorithms is given as follows:

3.1 PROJECT

A data-parallel operation that removes one or more attributes from the input relation and returns only selected attribute(s) in the output relation is PROJECT, which takes advantage of thread-level parallelism. As no complex partitioning is required for this operator, its implementation is relatively simple. Only one pass is required by our algorithm in which each work-item operates on one input tuple. Each work-item picks the specified attribute from the tuple, "key" in our case, places it in the register, and then transfers it to the output memory as shown in Algorithm 1. Every work-item operates in parallel, but its operation is serialized if available hardware resources such as registers are not sufficient for all work-items in action.

```
foreach work-item w in parallel do
    register ← input[w].key;
    output[w] ← register;
end
```

Algorithm 1: PROJECT

```
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
```

Algorithm 2: SELECT

3.2 SELECT

In SELECT, the key of each input tuple is evaluated for a given predicate function (e.g., a threshold like all dates for a certain event of interest). If the comparison is successful, the entire tuple is copied to the output relation; otherwise, the tuple is ignored. To implement this algorithm following the efficient three-stage procedure (i.e., partitioning, computing, and gathering), four OpenCL kernels are sequentially executed. Since complex partitioning is not required by SELECT, both partitioning and computational stages are performed by the first kernel, the gathering stage is performed by the fourth kernel, and other simple operations required to complete the SELECT operation are performed by second and third kernels, described later in the section. In the first kernel given by Algorithm 2, the input tuples are divided into the same number of equal-sized partitions as the number of work-groups. Each work-item is devoted to one tuple. The work-item reads the tuple in a register, determines the result of the predicate comparison, and stores the resulting Boolean, *countReg*, in another register. Work-items in each partition, or work-group, compute the parallel prefix-sum of countReg following Algorithm 3 to determine the positions of resulting tuples in the intermediate output memory set aside for each partition. This intermediate memory chunk can contain any number of tuples in the respective partition, depending on the predicate function; thus, we do not know the resulting size returned by each partition beforehand. To improve memory efficiency, a work-group transfers selected tuples first to the shared memory and then in bulk to the global memory dedicated to each partition. Therefore, gaps are left between the consecutive chunks in the global memory. An intermediate array stores the number of matched tuples found by each partition. To combine tuples selected by each partition, the parallel prefix sum is determined by second and third kernels to compute the indices of all the matched tuples in the output relation. The parallel prefix sum performed by the second kernel is given in Algorithm 3 and partitions in the second kernel is combined by the third kernel (not shown here). Finally, gaps between the tuples are removed by the fourth kernel, Algorithm 4, by placing them in contiguous positions, computed by second and third kernels. To improve memory efficiency, this memory-to-memory transfer in the fourth kernel, or the gather stage, is also done in bulk. SET operations and INNER JOIN use the gather kernel in a similar fashion.

```
foreach work-item w (with local id lw) in parallel do
    if lw is equal to 0 then
        $local[0] \leftarrow 0$;
    end
    localBuffer = local+1;
    $localBuffer \leftarrow input[w]$;
    for $i \leftarrow 0$ to numOfPartitions do
        if $numOfPartitions > 2^i$ then
            if $lw \leq 2^i$ then
                $localBuffer[lw] \leftarrow$
                $localBuffer[lw - 2^i] + localBuffer[lw]$;
            end
        end
    end
    $output[w] \leftarrow local[lw]$;
end
```

Algorithm 3: Parallel Prefix Sum

```
foreach partition p in parallel do
    foreach local work-item lw in parallel do
        output[number of elements in previous partition + lw] $\leftarrow$
        $globalPartition[lw]$;
    end
end
```

Algorithm 4: Gather

3.3 INNER JOIN

The most complex operators are INNER JOIN and set family. INNER JOIN takes in two sorted input relations and combines tuples containing the same keys. The implementation of INNER JOIN consists of the same three stages (i.e., partition, compute, and gather) and use five OpenCL kernels. In INNER JOIN, we use a separate kernel for the complex partitioning of input relations. One input relation (left) is divided into the same number of equal-sized partitions as the number of generated work-groups. Based on the pivot elements of each partition on the left, we use a binary search to look up tuples in the other input (right). In this way, we create partitions on the right that may contain tuples with the same keys. Along with finding bounds, we predict the output size of each partition in this partitioning kernel. After the partitioning stage, the parallel prefix sum is computed by the second kernel given by Algorithm 3. These partitions are combined by the third kernel to determine the actual starting position of the output of each partition. The actual join operation, depicted in Algorithm 5, is performed by the fourth kernel. In this kernel, the tuples in each partition are first placed in the local shared memory. Then each work-item takes one tuple of the partition on the right, finds the number of matching tuples in the partition on the left using the same binary search, and stores this number in a register, *foundCountReg*. If the partition on the right does not fit in the local memory, it is brought to the local memory in multiple iterations and the same work is performed. Subsequently, the prefix sum is computed on foundCountReg to find the index of each resulting tuple in the output memory. The joined tuples are first placed in the local memory at their respective indices and then transferred to the intermediate output memory. The resulting count of the number of joined tuples is also stored in an array. After the computation in the fourth kernel, the implementation of last three kernels is the same as described for the SELECT operator.

```
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 lw) in parallel do
            $rightLocal[lw] \leftarrow right[w]$;
            $leftLocal[lw] \leftarrow left[w]$;
            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
                while right partition end do
                    $rightLocal[lw] \leftarrow right[w]$;
                    if the leftLocal last tuple key < the
                    rightLocal first tuple key then
                        break the loop;
                    end
                    Algorithm 6;
                end
            end
        end
    end
    if lw is 0 then
        $array[p] \leftarrow total$;
    end
end
```

Algorithm 5: INNER JOIN

```
$rightReg \leftarrow right[lw]$;
$lowerReg \leftarrow$ lowerBound for rightReg.key in leftLocal;
$upperReg \leftarrow$ upperBound for rightReg.key in leftLocal;
$foundCountReg \leftarrow upperReg - lowerReg$;
indexReg = positions of selected tuples obtained from Algorithm 3;
totalReg = number of selected tuples obtained from Algorithm 3;
if totalReg < size of outputLocal then
    for $i \leftarrow 0$ to foundCountReg do
        $outputLocal[indexReg + i] \leftarrow$
        rightReg and matching left tuple;
    end
    $output \leftarrow outputLocal$;
else
    put in multiple iterations from outputLocal to output;
end
```

Algorithm 6: JOIN Block

3.4 UNIQUE

The UNIQUE operator removes consecutive duplicates within the provided attribute. The input sequence does not need to be sorted.

```
foreach partition p in parallel do
    foreach work-item w (with local id lw) in parallel do
        countReg ← 1;
        if w is not the last work-item of the partition then
            reg1 ← input[w];
            reg2 ← input[w];
            if reg1 is equal to reg2 then
                countReg ← 0;
            end
        end
        indexReg ←
        positions of selected tuples obtained from Algorithm 3;
        totalReg ←
        number of selected tuples obtained from Algorithm 3;
        if countReg is equal to 1 then
            local[indexReg] ← reg1;
        end
        foreach lw < totalReg do
            output[lw] ← local[lw];
        end
        if lw is 0 then
            array[p] ← total;
        end
    end
end
```

Algorithm 7: UNIQUE

Our implementation of UNIQUE is similar to that of SELECT (Algorithm 2), in which partitioning and computation stages are performed in the first kernel and output tuples are gathered by the last kernel. Instead of comparing input tuples with a predicate function, each tuple, or an element in this context, is compared with its adjacent element in the sequence. After simple partitioning, depending on the generated number of work-groups, each work-item of a work-group takes one element of the same index as its work-item ID from the input sequence, places it in a register, takes the next adjacent element, places it in another register, and then determines if they are equal or not. It is similar to creating two input sequences, one starting from the first element of the actual input and ending at the next to the last element and the other starting from the second element and ending at the last element, and then comparing corresponding elements. If they are not equal, another register or flag is set at 1; otherwise, it is set at 0. Note that work-items in the partition are equal to the size of the partition, but active work-items are one less than the size. The flag is initially set at 1 so that the last idle work-item has a flag value of 1. The prefix sum is calculated on this register, and the positions of the resulting unique elements or tuples in the intermediate output memory are determined. The unique elements are first placed in local shared memory and then transferred to global memory in bulk. The number of resulting tuples are stored in an intermediate OpenCL buffer. This algorithm is given in precise form in Algorithm 7. To eliminate duplicates on the edges of consecutive partitions, a separate kernel is called. If the same tuple is present on the edges of partitions, the tuple in the former partition is eliminated by reducing the partition size by one in the intermediate buffer. The implementation of last three kernels is same as that described for the SELECT operator.

3.5 TPC-H Micro-benchmarks

The TPC-H [3] benchmark is a decision support benchmark suite that provides a set of ad-hoc business queries. Using multiple data types and operators on large amounts of data sets, this suite, which consists of 22 queries with a high degree of complexity, addresses real-world business problems. After a detailed analysis of these TPC-H queries [15], Wu et al. identified a set of commonly occurring patterns of relational operators. The frequently occurring combinations of kernels, called "micro-kernels", drawn from the TPC-H benchmark suite, are shown in Figure 1 from [15]. From now on, we will call them (A), (B), and (C). The micro-benchmark (A) contains a series of SELECT operators that obtain the required tuples from the input relation; (B) consists of multiple dependent JOIN operators, which result in a large output relation that combines three input relations. (C) applies the JOIN operator on three input relations, following the SELECT operation on the relations and therefore, JOIN works on input relations that are smaller in (C) than in (B); then after SELECT and JOIN, required attributes are obtained using the PROJECT operator.

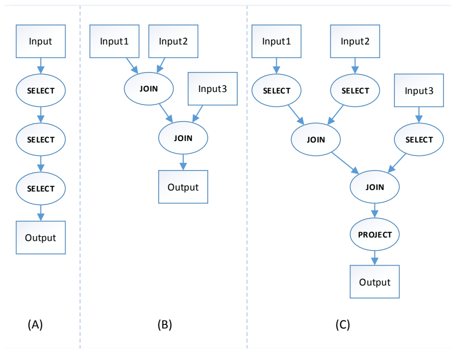

Figure 1. Micro-kernels from TPC-H queries

3.6 SHOC Benchmark Suite

The Scalable HeterOgeneous Computing benchmark suite (SHOC) [5] was designed to compare a set of common algorithms across multiple accelerated architectures and across multiple programming models. Compared to other accelerator-based benchmark suites like Parboil [13], Rodinia [2], and LoneStarGPU, SHOC stands out due to its focus on scalable, multi-node application benchmarks using MPI combined with its inclusion of CUDA, OpenCL, and Xeon Phi-focused versions of the same benchmarks. This cross-language development allows for performance comparisons of different accelerator architectures as well as the comparison of different programming models for the same type of accelerator [10].

At the same time, SHOC and other related benchmark suites have mainly focused on applications that are relevant to high-performance scientific computing while excluding enterprise-focused benchmarks like TPC and MapReduce-based benchmarks like TeraSort [11]. To fill this gap for evaluating accelerators, we propose the creation of a set of TPC-H microbenchmarks and eventually the implementation of a fully accelerated version of all the queries in the TPC-H benchmark. These queries already have been developed for CUDA outside of the SHOC framework, so the main engineering contribution for a full TPC-H OpenCL benchmark would be porting the proposed OpenCL primitives to operate in a similar manner as the Red Fox compiler framework [16], which takes TPC-H queries and breaks them down into the correct CUDA primitives for executions on GPUs. The continued development of this OpenCL-based TPC-H benchmark will also further the development of a stable platform for performing high-throughput queries on CPUs and accelerators for emerging Big Data problems.

This work provides a large first step towards this goal by demonstrating the implementation of the OpenCL primitives and evaluating several of these primitives and TPC-H microkernels on as many hardware platforms as possible. The tested benchmarks in this paper are currently available as a public fork of the main SHOC de-

Table 2. Experimental Setup

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

velopment branch. Currently, the contributed code focuses on the execution of the individual primitives and microbenchmarks and not on total execution runtime. Total runtime will become a more useful metric as these benchmarks are implemented as part of a larger implementation of TPC-H queries.

4. Experiments and Evaluation

Table 2 lists the experimental hardware used to evaluate the OpenCL RA primitives and microkernels. A selection of desktop-grade and enterprise-grade hardware was selected to include both high-end accelerators like the Xeon Phi "Knight's Corner" card and the NVIDIA K40 Kepler card as well as low-end CPUs like the Ivy Bridge i7 and the AMD Trinity APU with a fused CPU-GPU architecture. The existing benchmarks currently do not optimize based on the OpenCL workgroup size for each device, so some of the experiments may skew slightly towards the default settings which were used (a constant workgroup size of 256). This optimization would allow us to make more certain judgments on the differences in hardware used, but selecting a constant workgroup size also helps to control for one factor in systems that have many different software and hardware features.

The OpenCL implementations used included Intel's official OpenCL SDK, AMD's APP SDK, NVIDIA's OpenCL library in CUDA 6.0, and the beta software to support OpenCL on Intel integrated GPUs, Beignet [9]. Each of these packages aims to be interoperable with each other and support OpenCL 1.2, but there are certain differences due to continuing development of new OpenCL features and bugs that are still under development. Most importantly, the Intel OpenCL SDK currently has a bug that does not allow for OpenCL programs to be compiled with all the possible optimizations due to a regression in the underlying code that vectorizes OpenCL kernels for CPUs and the Phi. This temporary issue likely has impacted our CPU and Phi results slightly, but it is difficult to quantify the effect at this moment. For this reason, we emphasize that the "best" hardware for a specific operation is continually in flux and these results may change with the next OpenCL 1.2 release by vendors.

4.1 Benchmark Setup

Each of the discussed benchmarks, Select, Project, Unique, Join and microkernels, A, B, and C were built using the platform-local version of OpenCL as part of the overall SHOC framework. OpenCL timers were used to measure the amount of execution time for data transfer, computation for the overall primitive and for the component kernels used to implement a primitive. Each primitive was tested over a range of input sizes from 32 KB up to 1 GB, where the listed size represents the size of one of the inputs as opposed to the total input size. While not every device tested can run a specific benchmark with 1 GB sizes, those that can are listed to emphasize that higher-end devices allow for larger memory buffer allocations. Inputs for all the input tuples were generated randomly, with special focus on join inputs to try and prevent joins between empty sets of tuples.

4.2 Results

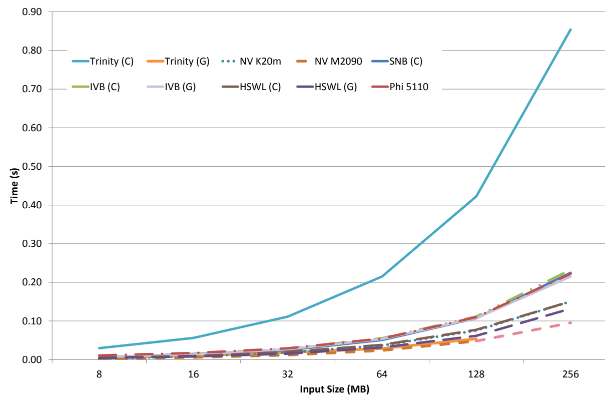

Figure 2. Select Total Execution Time

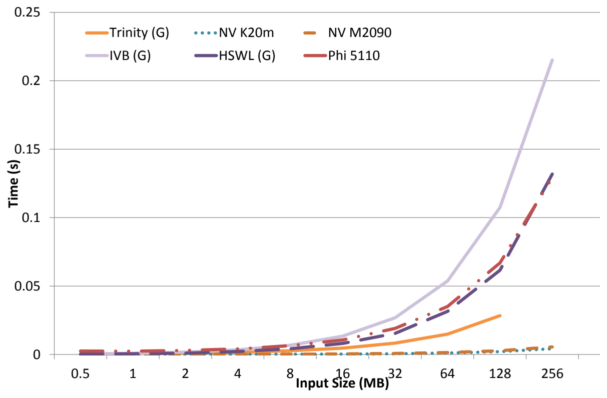

Figure 3. Select Kernel Execution Time - Accelerators

The total timing results for the unary select benchmark are shown in Figure 2, where the primitive takes from 143 ns (M2090) up to 85 ms (Trinity CPU) to run for sizes ranging from .5 MB up to 256 MB, including all data transfers. Right away this figure demonstrates that the low-end CPU, the AMD APU performs much slower due to its lower thread count and processor speed, but this figure doesn't provide a detailed picture of how each device fairs for this primitive. Figures 3 and 4 break out these results into similar classes of CPUs and accelerators for the kernel execution time excluding all data transfers. Figure 3 shows that the Trinity integrated GPU actually outperforms both the Intel and Haswell integrated GPUs as well as the Phi 5110, while the Kepler GPU card has the lowest kernel execution time (4.2 ms) of all devices at the largest input size of 256 MB. While this is surprising that the Trinity GPU performs so well on select, the lack of Intel-specific optimizations (see Section 4) is likely to penalize Phi

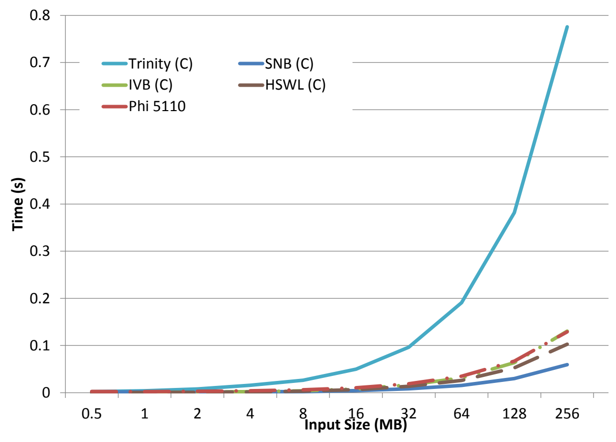

Figure 4. Select Kernel Execution Time - CPUs and Phi

results slightly as well as the fact that the specific kernels may not be able to make use of all of the available Phi cores without specific code targeting. Figure 4 demonstrates that the Trinity CPU lags behind the consumer-grade Intel CPUs as well as the high-end Sandy Bridge platform, which has 2 Xeon processors, each with twelve CPU threads.

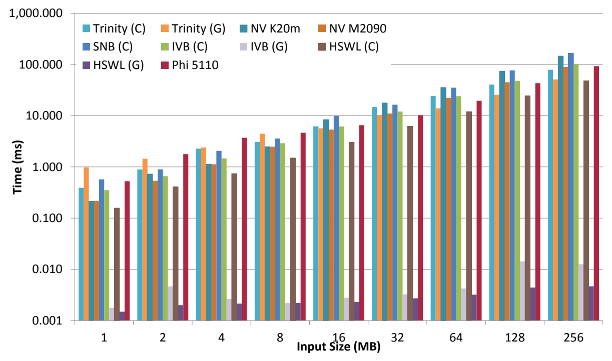

Figure 5. Select Data Transfer

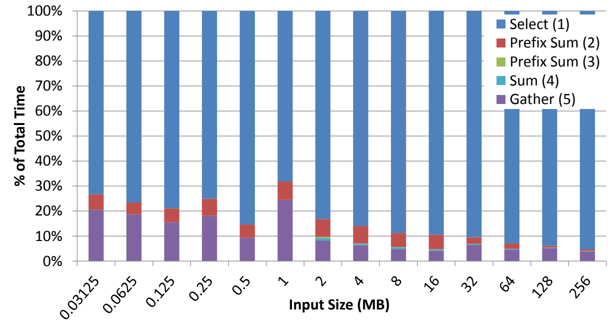

Figure 6. Select Kernel Breakdown - Phi 5110

The design of the benchmark suite for our relational algebra primitives also allows for the analysis of data transfer overheads and more precise measurements of the constituent components of a particular primitive, as shown in Figures 5 and 6. For most of the test cases with a single primitive, data transfer is a large source of overhead. For example data transfer consumes about 165 ms out of a total runtime of 221 ms for the Sandy Bridge processor running Select. At the same time, data transfer only takes 48 ms out of a total runtime of 151 ms for the Haswell CPU. These differences across different generations of processors can be mostly attributed to the implementation of zero-copy features with newer processors like Haswell and the AMD Trinity APU, which prevents operating system overhead to “transfer” data from the host memory space to the address space which is executing OpenCL kernels.

Figure 6 demonstrates the component subkernels for the Select primitive on the Xeon Phi. As mentioned in Section 3, each primitive is designed around a partition, compute, and gather phase. The “select” subkernel roughly corresponds to the partitioning and the first part of the compute step, while prefix sum and sum are used as intermediate computation steps, and gather performs the final phase of the selection operation. As the input size grows, the partitioning step becomes more time-consuming, indicating that optimized code for this step could be used to improve the overall performance of select operations.

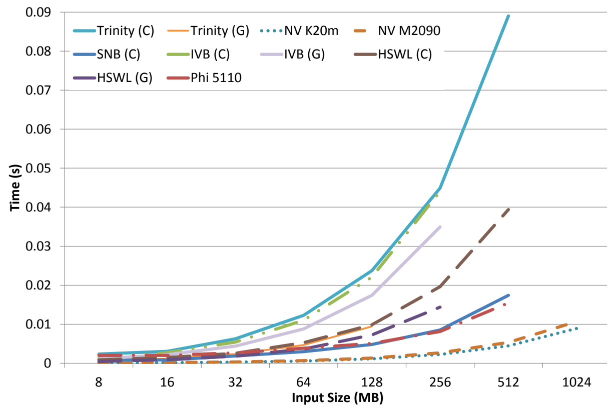

Figure 7. Project Kernel Execution Time

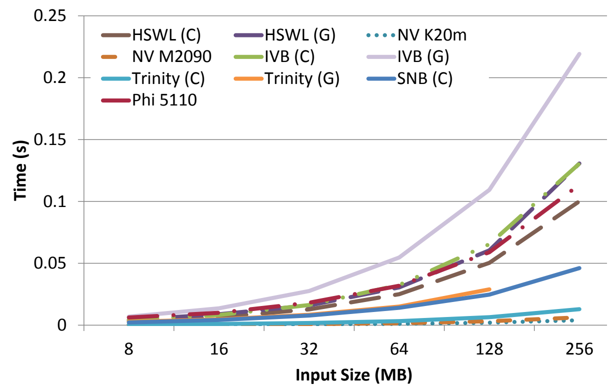

Figure 8. Unique Kernel Execution Time

Project (Fig. 7) and Unique (Fig. 8) are both interesting for different reasons, since they detail how different processors and accelerators map for simplistic unary operators (project) and more complex binary operators (unique, join, etc.). Project shows more separation in terms of how fast discrete accelerators (Kepler, Fermi, Phi) can perform the project operation when compared to CPU and integrated or fused accelerators. Here again the GPUs perform the best, likely due to the default OpenCL workgroup size skewing closely to their thread layout. Unique shows that the most optimized integrated CPU/GPU part, the AMD trinity performs much better for the unique kernel operation than its comparable CPUs, specifically the Ivy Bridge CPU and GPU parts.

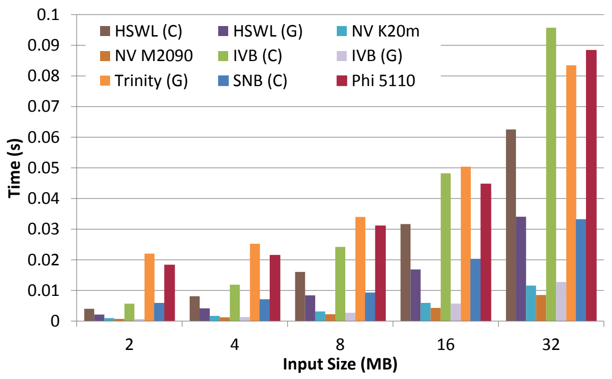

Figure 9. Join Kernel Execution Time

In Figure 9, the results for the Join kernel show that again the discrete GPUs are the best choice for this primitive, while the Haswell GPU and Sandy Bridge CPU also perform well on this complex operation. Somewhat surprisingly, the Xeon Phi struggles at the largest input size (2x 32 MB) with a kernel execution time of 88 ms while the Ivy Bridge CPU takes almost 95 ms. Both of these results are likely due to the lack of optimizations with the Intel SDK due to the aforementioned vectorization regression in the latest software release (Section 4). Currently our benchmark suite only supports this primitive with inputs of up to 64 MB due to the potential for large outputs with inner join operations.

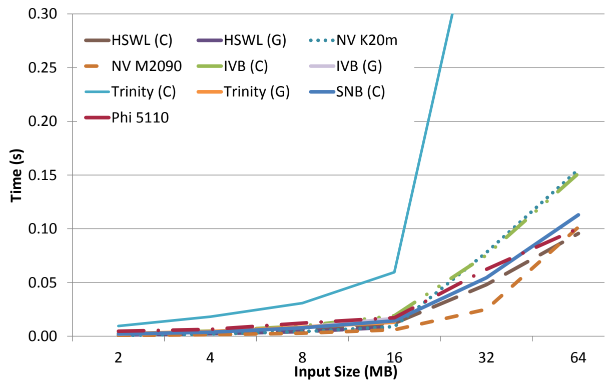

Figure 10. MicroKernel A Execution Time (with data transfer)

The microkernel benchmarks tend to exhibit the same characteristics as the primitives, although they do behave slightly differently due to the ordering of operations (and resultant data reductions for operations like select). Microkernel A (Fig. 10) demonstrates again that the AMD Trinity's CPU is the worst choice for chained select operations with a total runtime of around 900 ms, while the GPUs and newer CPUs provide a good degree of parallelism with most operations finishing in under 200 ms.

Like the join kernel, the results for microkernel B (two joins) in Figure 11 show that the Intel CPU parts and Trinity GPU take the longest to finish, due to lack of CPU-specific optimizations and low clock speeds for the Trinity GPU. In this particular test, the chained join does not take much longer than two single joins, indicating that the first join operation did not generate a large amount of output to be used as input for the second join.

Finally, the high-end accelerators results for microkernel C (Fig. 12) illustrate the contribution of the chained join operations in overall execution time. While the Phi can quickly perform select and project operations, its long execution time for join operations leads to it being much slower than the discrete GPUs at larger input sizes, with three 64 MB inputs taking 1.88 seconds to process versus 19 and 14 ms for the K20 and M2090 GPUs.

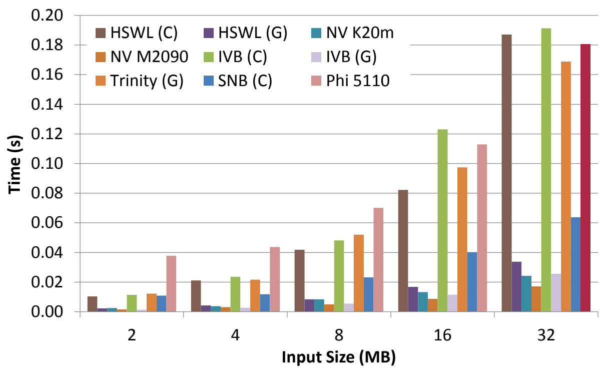

Figure 11. MicroKernel B Execution Time

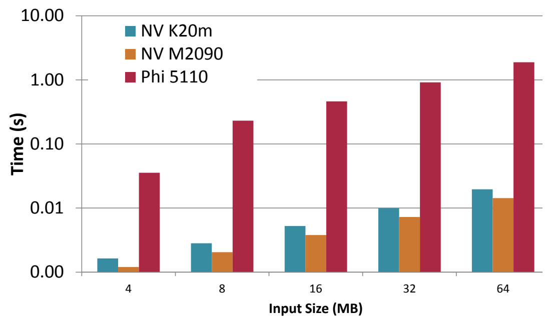

Figure 12. MicroKernel C Execution Time (High-end)

4.3 Further Discussion

These results demonstrate a cross-section of the full set of primitives discussed in more detail in [12], and they also illustrate several insights that may be familiar to GPU and Xeon Phi programmers but that bear repeating in the context of this new analytics application space.

- Because of the large number of compute units, high memory bandwidth, and many on-chip resources, the discrete GPUs tend to be best suited for the evaluated primitives. While the Xeon Phi also performs well on primitives other than joins, it is penalized slightly by the lack of current Intel optimizations and limited workgroup size optimization on our part for the benchmarks.
- In all processors, particularly in the case of the discrete GPU, very simple primitives such as PROJECT, in which the kernel execution time is a very small fraction of the overall execution, the time spent on transferring data between the host and the device dominates the total execution time. Since the data transfer time is shorter for fused GPUs and CPUs, the total execution time of primitives ends up being shorter than that on the discrete GPU.
- Since data warehousing analytics typically deals with large data sets, the data transfer time cannot solely be reduced by using pinned memory, a scarce resource that cannot accommodate both the inputs and output relations used and produced by

the primitives. Therefore, we cannot fully take advantage of zero-copy memory accesses, but instead, we have to perform (relatively slow) transfers between the host and the device. This overhead was demonstrated in Figure 5 and it points to the need for improved zero-copy semantics for future "fused" CPU and GPU parts like the AMD APU or Haswell CPU.

- The excellent performance results for simple primitives with the high-end Sandy Bridge CPUs point to a clear caveat for optimizing the same application across different types of accelerators - in many cases, a fast, multi-threaded CPU can get good performance that compares well with accelerated code that is moderately optimized for a target architecture (as is the case with these primitives). However, our goal in this case is to ensure good performance portability across a variety of architectures, as opposed to an optimized solution for one piece of hardware.
- In our experiments, the memory buffer limit differs across devices depending on the size of the global memory for GPUs and RAM in case of CPUs. To run primitives for input sizes larger than the size of the maximum OpenCL allocation size, we would have to manage data transfers in concert with ongoing computation. In future work, we would like to investigate asynchronous data transfers for this benchmark suite and the full-fledged TPC-H implementation.
- From our experimental results, we see preliminary results that indicate that when scheduling kernels for execution on a system with both fused and discrete GPUs, one should schedule fine-grained kernels on the fused GPU, complex kernels on the discrete GPU, and kernels requiring a large amount of memory on the CPU. The OpenCL implementation of this library makes such scheduling decisions possible, thereby enabling one to opportunistically choose and use core types that are available at run time.

5. Conclusions

In this paper, we have presented a portable library containing RA (relational algebra), arithmetic, and other related primitives required to run data-intensive relational queries. In our experiments, we used multiple GPUs and CPUs to evaluate the library and concluded that discrete GPUs outperform other integrated GPUs and CPUs due to large numbers of threads and better available optimizations for the tested kernels. Although data transfer time contributes to a significant portion of the total execution time of discrete GPUs, the computation power of integrated GPUs and CPUs is still not sufficient to compete with the short execution time of discrete GPUs, especially for large data sets. However, the results showed that each successive generation of integrated GPU provides better performance that is comparable for smaller query sizes. This finding suggests that in complex queries involving multiple primitives that are common in data warehousing applications, less expensive integrated GPUs surpass expensive discrete GPUs in performance if data transfer is not optimized between devices. We look forward to further investigating these trade-offs with future work involving the larger set of TPC-H queries.

Acknowledgments

This research was supported in part by the National Science Foundation under grant CCF 1337177 and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. Also, special thanks to NVIDIA for the use of a Kepler GPU provided through Georgia Tech's NVIDIA CUDA Center of Excellence and Dr. Jeffrey Vetter of ORNL's FTG group for the use of several discrete GPUs.

References

[1] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward hardware-sensitive database operations.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[3] T. P. P. Council. TPC Benchmark H (Decision Support) Standard Specification, Revision 2.17.0 . `http://www.tpc.org/tpch/spec/tpch2.17.0.pdf`, 2013.

[4] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE, 2012.

[5] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[6] B. Devlin and L. D. Cote. *Data warehouse: from architecture to implementation*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[7] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. *CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-12-01*, 2012.

[8] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[9] Intel. Intel beignet opencl implementation. `http://www.freedesktop.org/wiki/Software/Beignet/`, 2014.

[10] A. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176–185, Sept 2011.

[11] A. Owen O'Malley. Terabyte sort on apache hadoop. `http://sortbenchmark.org.3s3s.org/YahooHadoop.pdf`, 2008.

[12] I. Saeed. A portable relational algebra library for high performance data-intensive query processing (MS thesis). `https://smartech.gatech.edu/handle/1853/51967`, 2014.

[13] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[14] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499, Feb 2014.

[15] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE Computer Society, 2012.

[16] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An execution environment for relational query processing on gpus. 2014.

[17] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013.