A PORTABLE RELATIONAL ALGEBRA LIBRARY FOR HIGH PERFORMANCE DATA-INTENSIVE QUERY PROCESSING

A Thesis
Presented to
The Academic Faculty

By

Ifrah Saeed

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2014

Copyright © 2014 by Ifrah Saeed

A PORTABLE RELATIONAL ALGEBRA LIBRARY FOR HIGH PERFORMANCE DATA-INTENSIVE QUERY PROCESSING

Approved by:

Dr. Sudhakar Yalamanchili, Advisor
Professor, School of ECE
Georgia Institute of Technology

Dr. George F. Riley
Professor, School of ECE
Georgia Institute of Technology

Dr. Hyesoon Kim
Associate Professor, School of Computer Science
Georgia Institute of Technology

Date Approved: April 4, 2014

To my family and my adviser

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Dr. Sudhakar Yalamanchili, for his valuable suggestions and unwavering guidance. The completion of my thesis would not have been possible without his constant support and encouragement. I am deeply indebted to my parents, Ishfaq Ahmed and Khalida Yasmeen, for sending me to the US to pursue the master's degree and believing in my abilities throughout the course of my studies. I would also like to extend my gratitude to my siblings, especially my brother Shayan, who continuously encouraged me. Furthermore, I am very grateful to all of my CASL lab colleagues, especially Jin Wang, Haicheng Wu, and Syed Minhaj Hassan, who took their precious time to help me whenever I needed it. I want to thank Prashant Nair, who encouraged me to work on my thesis until late at night close to the thesis submission deadline. I would also like to acknowledge the help of Jude-Thaddeus Ojiaku, with whom I discussed the basics of OpenCL online. Many thanks to my roommate, Sadia Shakil, and other friends who patiently tolerated my mood swings during trying times. In the end, I very much appreciate the large Pakistani and Muslim community in Atlanta and at Georgia Tech who made me feel at home in the US. My life in the graduate school would have been barren without all of these friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A growing number of industries are turning to data warehousing applications such as forecasting and risk assessment to process large volumes of data. These data warehousing applications, which utilize queries comprised of a mix of arithmetic and relational algebra (RA) operators, currently run on systems that utilize commodity multi-core CPUs. If we acknowledge the data-intensive nature of these applications, general purpose graphics processing units (GPUs) with high throughput and memory bandwidth seem to be natural candidates to host these applications. However, since such relational queries exhibit irregular parallelism and data accesses, their efficient implementation on GPUs remains challenging. Thus, although tailored solutions for individual processors using their native programming environments have evolved, these solutions are not accessible to other processors. This thesis addresses this problem by providing a portable implementation of RA, mathematical, and related primitives required to implement and accelerate relational queries over large data sets in the form of the library. These primitives can run on any modern multi- and many-core architecture that supports OpenCL, thereby enhancing the performance potential of such architectures for warehousing applications. In essence, this thesis describes the implementation of primitives and the results of their performance evaluation on a range of platforms and concludes with insights, the identification of opportunities, and lessons learned. One of the major insights from our analysis is that for complex relational queries, the time taken to transfer data between host CPUs and discrete GPUs can render the performance of discrete and integrated GPUs comparable in spite of the higher computing power and memory bandwidth of discrete GPUs. Therefore, data movement optimization is the key to effectively harnessing the high performance of discrete GPUs; otherwise, cost effectiveness would encourage the use of integrated GPUs. Furthermore, portability also enables the complete utilization of all GPUs and CPUs in the system at run time by opportunistically using any type of available processor when a kernel is ready for execution.

CHAPTER 1

INTRODUCTION

As traditional operational databases typically handle short queries and small amounts of data, they are useful for simple transactions, not for processing complex queries in data-intensive applications [2]. 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 [3]. 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) 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.

Over the past few years, the implementation of RA operators on GPUs has led to better performance than that on CPUs [4]. In their implementation, a substantial portion of execution time is spent on explicit data transfer between the host/CPU and the GPU. While this data transfer from the host to the GPU and back to the host after processing on the GPU incurs overhead [5], it may be avoided if the host and the device share the same memory space. Keeping this in mind, the industry introduced integrated CPU-GPU architectures.

Recent processors with this architecture include AMD Kaveri and Richland [6], Intel′s Haswell and Ivy Bridge [7], and Nvidia Echelon [8]. The industry is also focusing on making discrete GPUs, such as AMD Southern Island devices [9] and NVIDIA Kepler [10], with separate host and device memory more powerful through the addition of more computing units and an increase in memory bandwidth. Researchers are concurrently developing methods of increasing the performance of RA operators on modern CPUs by exploiting their multiple general-purpose cores. Many recent performance studies comparing GPUs to CPUs, most of which were carried out using CUDA on Nvidia GPUs, have reported tremendous speedup [1] [4]. However, one of the major drawbacks of using this approach is that CUDA restricts portability and runs only on Nvidia processors [11]. Therefore, this motivates us to pursue the portable implementation of RA, arithmetic, and other primitives required to execute relational queries using OpenCL and evaluate their performance on a range of available CPU and GPU platforms [12].

The objective of this thesis is to construct a portable library that comprises the optimized OpenCL implementation of a set of RA, arithmetic, and other related primitives. These primitives are also used to implement micro-benchmarks derived from the TPC-H industry standard benchmark suite [13]. This study will evaluate the library across a range of CPUs, discrete GPUs, and fused CPU-GPU architectures. The research goals are 1) to understand the relative strengths and weaknesses of both discrete and fused CPU-GPU configurations from the perspective of the relational processing of large data volumes and 2) to enable the cross-platform portable realization of support for relational queries. The long-term goal of this project is to merge this library with the ongoing Red Fox project [1] in which our OpenCL library will complement the existing RA primitive CUDA library. The overview of this project is given in Chapter 2. From our experiments, we found out that if only the time taken to execute kernels is measured, discrete GPUs having a large number of compute units, high memory bandwidth, and many hardware resources available on-chip are the fastest. When the data transfer time is added to the kernel execution time,

the performance of discrete GPUs is slightly better than that of integrated GPUs in case of complex micro-benchmarks and worse in case of simple micro-benchmarks. Therefore, we concluded that when multiple processors are available in the system then one should schedule fine-grained kernels on the integrated GPU, complex kernels on the discrete GPU, and kernels requiring a large amount of memory on the CPU.

CHAPTER 2

BACKGROUND

2.1 Relational Algebra Operators

Relational database management system (RDBMS) and relational database query languages such as SQL [14] and Datalog [15] are theoretically based on RA (relational algebra), which defines a formal set of rules of how relational databases operate. RA consists of several basic and derived operators that work on relations. A relation is a set of tuples, each of which is composed of one or more attributes. In other words, an n-ary tuple in a relation consists of n values for each attribute in a relation. The main RA operators include SELECT, PROJECT, CROSS PRODUCT, JOIN, and SET OPERATIONS (SET UNION, SET DIFFERENCE, SET INTERSECTION). A single data warehouse query contains several RA and other operators that provide a means of retrieving and manipulating data in existing relations and transforming data into new relations.

In our implementation, we have stored relations as densely packed arrays of key-value pairs or tuples. A tuple has two attributes: a key attribute, and a value attribute. Each tuple can have up to three value attributes, shown in Figure 1. Operators usually function on key attributes, but in some cases, they work on both key and value attributes. For optimization across operators, we implemented all operators using an algorithmic structure with a sequence of identical stages: partition, compute, and gather. These multistage algorithms are explained in Chapter 3.

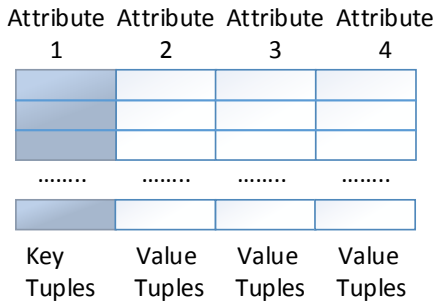


Figure 1: Key-value pairs representation

2.2 OpenCL vs CUDA

Recent frameworks commonly used for developing structured and irregular applications on parallel architectures are Open Computing Language (OpenCL) [16] and Compute Unified Device Architecture (CUDA) [11]. These frameworks include OpenCL and CUDA C programming languages, respectively, which follow a parallel programming model known as bulk-synchronous parallel (BSP) model [17]. With this model, the OpenCL or CUDA application has one or more data-parallel kernels executed across a number of work-items (OpenCL) or threads (CUDA). Work-items/threads are grouped into local work-groups/ cooperative thread arrays (CTAs) that share local/shared memory and can be synchronized using the OpenCL API or CUDA function, or barriers. Work-items/threads within the work groups are also grouped on the hardware level, and these groups are referred to as ”wavefronts” in OpenCL and ”warps” in CUDA. From now onwards, we will use OpenCL terminology. Wavefronts execute the same instruction, work in lock-step fashion, and a large number of them executing in parallel can hide pipeline and memory latency. Figure 2 illustrates a work-item, a work-group, and an NDRange, which represents the total number of work-items and can be either one-, two-, or three- dimensional. A work-item operates on a processing element, a work-group on a compute unit, and NDRange on the device. Processing elements and compute units are explained in Chapter 4 when platforms are described in detail.

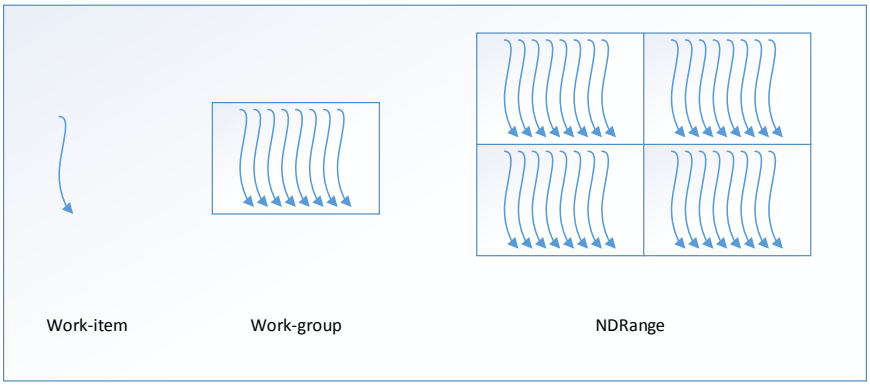


Figure 2: Basic blocks of OpenCL

Table 1: Comparison between OpenCL and CUDA

	OpenCL	**CUDA**
Vendor Support	Multiple vendors: AMD, Intel, Nvidia and others.	One vendor: Nvidia.
Code Portability	Code portable on multiple architectures but not optimized for speed.	Not portable. Run only on Nvidia processors.
C++ Constructs	Based on C99 (AMD supports static C++ kernels).	Allows C++ constructs.
Out-of-order Queues	OpenCL supports out-of-order queues.	CUDA still has no equivalent to OpenCL out-of-order queues.

In the OpenCL or CUDA memory hierarchy, global memory has the highest latency. It is used for the transfer of data between the host and the device, and by kernels to read and write. Work-groups share the local memory, and private memory or registers are dedicated to each work-item. OpenCL and CUDA do not provide any global barriers, but the programmer can do his own global synchronization while writing code. Both frameworks offer some advantages and disadvantages, but the common purpose is to try to increase computing performance. The main differences between OpenCL and CUDA are listed in Table 1.

Studying irregular applications such as graph and relational query processing applications on parallel processors is now mainstream research [18] [1]. A significant amount of research has been devoted to the study of the potential of GPUs to speed up irregular applications. This thesis targets the implementation of relational algebra and related primitives used in relational database queries that are data-intensive but highly unstructured and irregular. Because of the support of multiple vendors and the convenience of porting code across multiple different architectures, we used OpenCL for this thesis [19], thereby gaining access to a wide range of CPUs and GPUs. This thesis provides the performance evaluation of primitives on the GPUs and CPUs of two different vendors. Other accelerators and

processors such as Intel Xeon Phi will be evaluated in the future.

2.3 Red Fox

Red Fox [1] is a framework that accelerates Datalog [15] queries on GPUs, thereby providing an execution environment for enterprise applications. The organization of Red Fox is shown in Figure 3, which illustrates its capability of compiling LogiQL source, a superset of Datalog, and converting it into GPU binary code. It is the only open source system known to date that can run the full TPC-H queries suite on GPUs. The purpose of Red Fox is to improve energy efficiency, increase the amount of parallelism, and reduce data movement between the host and the GPU. It provides the optimized implementation of RA operators on GPUs. It uses hybrid multi-stage algorithms for the primitives designed to utilize all cores and saturate memory bandwidth [20]. It also optimizes data movement between the host and the GPU by increasing the granularity of the kernel computation using the principle of temporal locality [13]. Red Fox also applies cluster-wide memory aggregation techniques for in-core techniques [21].

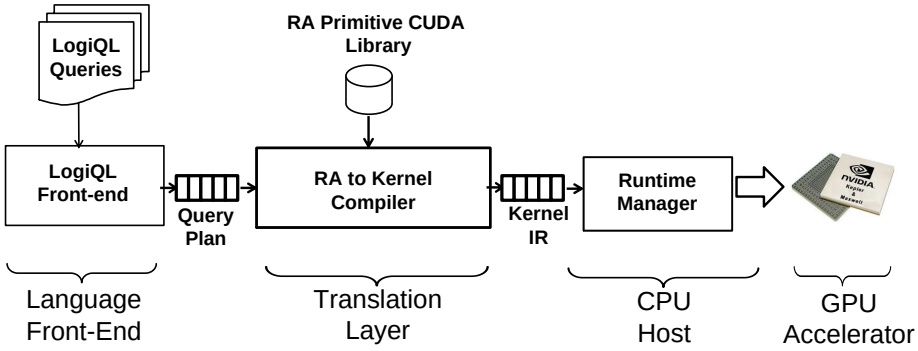


Figure 3: Red Fox Platform Diagram [1]

The Red Fox compilation and run-time infrastructure, illustrated in Figure 3, is comprised of the following components: 1) *LogiQL front-end*, which takes LogicQL program as input, parses and analyzes it to produce an intermediate representation (IR) of the query plan for primitives; 2) *RA primitives CUDA library*, which contains the optimized implementation of RA and related primitives in CUDA. Because of the implementation of these

primitives in CUDA, Red Fox is supported only on NVIDIA GPUs. This thesis provides the implementation of these primitives in OpenCL so that this framework can take advantage of many platforms; 3) *RA to kernel compiler*, which takes in the query plan and the executable CUDA implementation of the primitives and converts it into the kernel IR; and 4) *run-time manager*, which executes the binary of the kernel IR. As already listed, the entire framework is divided into separate components that facilitate the easy integration of other language front ends and GPU backends.

2.4 TPC-H Benchmark

The TPC-H [22] 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. The Red Fox infrastructure supports this full set of queries and achieves high speedup compared to a commercial LogiQL [15] system implementation optimized for CPUs [1]. In this thesis, in addition to the individual primitives themselves, we will analyze the performance of micro-benchmarks constructed from an analysis of the TPC-H benchmark. These micro-benchmarks [13] represent commonly occurring patterns of dependent kernels that were found to occur in the 22 queries. The micro-benchmarks are discussed in detail in Chapter 4.

CHAPTER 3

RED FOX OPENCL PRIMITIVES LIBRARY

The Red Fox primitives library contains relational algebra, arithmetic, and other related primitives required to efficiently execute relational queries. We are merging our OpenCL primitives library with the RA primitives CUDA library in Red Fox so that it can take advantage of modern multi-core, many-core, and other recent processors and accelerators. This chapter lists and describes the primitives present in the library and the algorithmic structure of each.

3.1 Primitives

The library is comprised of relational algebra operators (i.e., PROJECT, SELECT, CROSS PRODUCT, INNER JOIN, SET INTERSECTION, SET UNION, SET DIFFERENCE), math primitives (i.e., REDUCE, REDUCE BY KEY), and other primitives (i.e., SORT, UNIQUE) as described in Table 2. The table provides a brief description of all of the primitives implemented as a part of this thesis. These primitives are sufficient to execute all of the TPC-H queries. In examples provided with the primitives in Table 2, the first attribute is ”key,” the second attribute is ”value,” and both are unsigned integers. However, in the case of the output of INNER JOIN, the first attribute is key and the other two are value attributes.

Table 2: Set of Implemented Primitives

Operators	Description	Examples
PROJECT	A unary operator that selects one or more attributes from an input relation to produce a new output relation.	x={$(4,a),(1,b)$} Project(x.key) $\rightarrow$ {$4,1$}
SELECT	A unary operator that selects a set of tuples from an input relation that satisfies certain filter criteria, applied on one or more tuple attributes, to produce a new output relation.	x={$(4,a),(1,b)$} Select(x.key >2) $\rightarrow$ {$(4,a)$}
CROSS PRODUCT	A binary operator that combines every tuple of one input relation to all of the tuples of the second input relation to produce a new relation.	x={$(4,a),(1,b)$}, y={$(1,c)$} Product(x,y) $\rightarrow$ {$(4,a,1,c),(1,b,1,c)$}
INNER JOIN	A binary operator that combines the tuples with the same keys in both input relations to produce the output relation.	x={$(4,a),(1,b)$}, y={$(1,c)$} InnerJoin(x,y) $\rightarrow$ {$(1,b,c)$}
SET INTERSECTION	A binary operator that puts tuples present in the input relations with the same keys and values in the output relation.	x={$(4,a),(1,b)$}, y=$(1,b)$ Intersection(x,y) $\rightarrow$ {$(1,b)$}
SET UNION	A binary operator that combines all of the tuples of the input relations after removing duplicates to produce the output relation.	x={$(4,a),(1,b)$}, y={$(1,c)$} Union(x,y) $\rightarrow$ {$(1,c),(1,b),(4,a)$}
SET DIFFERENCE	A binary operator that puts the tuples of one input relation that do not exist in the second input relation in the output relation.	x={$(4,a),(1,b)$}, y={$(1,b),(4,c)$} Difference(x,y) $\rightarrow$ {$(4,a)$}
SORT	A unary operator that sorts the tuples of an attribute (either a key or a value attribute) in ascending order. For key-value pairs, it sorts the tuples of the key attribute in ascending order and sorts the value attribute in ascending order relative to the key attribute.	x={$(4,a),(1,b),(1,a)$} Sort(x) $\rightarrow$ {$(1,a),(1,b),(4,a)$}
UNIQUE	A unary operator that removes consecutive duplicates in an attribute.	x={$4,1,1,1,5,7,7,2$} Unique(x) $\rightarrow$ {$4,1,5,7,2$}
REDUCE	A unary operator that sums the tuples of an attribute.	x={$4,1,1,1,5,7,7,2$} Reduce(x) $\rightarrow$ {28}
REDUCE BY KEY	A unary operator that, for each group of consecutive keys, sums the respective tuples of another attribute.	x={$(4,a),(1,b),(1,a)$} ReduceByKey(x) $\rightarrow$ {$(4,a),(1,a+b)$}

3.2 Algorithm Design

This section describes the high-level algorithmic structure of the primitives in the library. These algorithms have a structure very similar to those described in [20]. 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 [20], 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 [20] 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:

3.2.1 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**
    $register \leftarrow input[w].key$;
    $output[w] \leftarrow register$;
end

Algorithm 1: PROJECT

3.2.2 SELECT

In SELECT, the key of each input tuple is evaluated for a given predicate function. 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 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

3.2.3 PRODUCT

Another relatively simple operator that uses two input relations, PRODUCT, combines each tuple of one input relation with all the tuples of the other. Therefore, the number of tuples

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

Algorithm 4: Gather

in the output relation is the product of the number of tuples in both relations. To implement PRODUCT, no special partitioning is required, so we divide the input tuples into the same number of equal-sized partitions as the number of generated work-groups. Since each tuple is accessed multiple times, each work-group places its assigned tuples in the shared local memory for faster access. After combining the tuples, the work-groups transfer them to the global memory. Since we know the output size a priori, we do not need to determine it separately, as we did for SELECT. Algorithm 5 gives the precise description of PRODUCT.

```
foreach work-item w (with local id lw) in parallel do
    leftReg ← left[w];
    for i ← 0 to elements in right array do
        rightLocal[lw] ← right[lw + i];
        for j ← 0 to localSize do
            output ← leftReg and rightLocal[j];
        end
        i ← i + localSize;
    end
end
```

Algorithm 5: PRODUCT

3.2.4 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 6, 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.

3.2.5 SET Family

SET INTERSECTION, SET UNION, and SET DIFFERENCE operate on two sorted input relations. INTERSECTION finds the common key-value pairs in both. After eliminating the duplicates, UNION combines all tuples in both relations. DIFFERENCE selects tuples that exist only in one relation. The implementation of the SET family follows the same series of kernels and steps as that of INNER JOIN (Algorithm 6), but it differs in the partitioning step and the actual computation. To identify duplicate key-value pairs, every tuple is assigned a rank. Every unique tuple has a 0 rank, and with every occurrence of the same tuple, the rank is increased by 1. Although SET family also uses a binary search to perform partitioning, in this case, partitioning is not just based on keys but on the combination of keys, values, and ranks. The implementation of the second kernel, parallel prefix sum, given by Algorithm 3, and that of the third kernel is same as that in JOIN. The actual computation kernels differ in INTERSECTION, UNION, and DIFFERENCE.

```
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] ← right[w];
            leftLocal[lw] ← 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 7;
                end
                while right partition end do
                    rightLocal[lw] ← right[w];
                    if the leftLocal last tuple key < the rightLocal first tuple key then
                        break the loop;
                    end
                    Algorithm 7;
                end
            end
        end
    end
    if lw is 0 then
        array[p] ← total;
    end
end
```

Algorithm 6: INNER JOIN

```
rightReg ← right[lw];
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;
if totalReg < size of outputLocal then
    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 7: JOIN Block

In the case of INTERSECTION, we take the output size of each partition as the minimum size of two partitions. The work-group brings the tuples of two partitions into the local shared memory, and each work-item takes charge of one tuple in the second input partition. Then it performs the binary search to find its lower and upper bounds. If the same tuple with the same rank is found in the first input partition, the count register is incremented. In UNION, the count register increases even if a tuple is not found in the second partition because we have to place all of the tuples in the output after removing the duplicates. Each work-item saves its lower and upper bounds in local memory and finds the leftover tuples in the first partition by subtracting its lower bound from the upper bound, calculated by the previous work-item. This count is also added to the count register to determine the number of resulting tuples. Here, we take the output size of each partition as the sum of the sizes of two partitions. In DIFFERENCE, each work-item operates on one tuple and checks to see if it exists in the second partition. If it does not, the count register is increased by 1. For DIFFERENCE, we take the output size of each partition as the size of the first partition. In case of all three operators of the SET family, the implementation of last three kernels is the same as described for the SELECT operator.

3.2.6 SORT

SORT is taken from VexCL [23], a vector expression template library that facilitates the writing of code in OpenCL and CUDA by reducing the boilerplate code and providing simple APIs for commonly used functions while programming GPGPUs. These frequently occurring functions include sort, reduce, and scan. VexCL is open-source and provided under an MIT license. It provides both CUDA and OpenCL backends and works on multiple devices and platforms. Details about this library can be found in [23]. From this library, we chose SORT because the algorithm is based on Sean Baxter's SORT algorithm [24]. Modern GPU is considered to have very fast and efficient algorithms and codes. Other open-source OpenCL SORT codes or algorithms available online work with inputs of sizes of a power of two, including codes in AMD and Nvidia SDK, or they are restricted to a

particular platform such as SORT in the Bolt library [25].

3.2.7 UNIQUE

The UNIQUE operator removes consecutive duplicates within the provided attribute. The input sequence does not need to be sorted. 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 8. 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.

```
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 8: UNIQUE

3.2.8 REDUCE

A simple aggregation operation, REDUCE, processes a single attribute as in UNIQUE (Algorithm 8). REDUCE sums all the tuples or elements in the attribute and stores the result in the output variable. In our implementation, Algorithm 9, the input attribute is first brought in the local shared memory in a coalesced manner. Then, each work-group operates on its own partition. Half of the work-group starts the reduction process and each work-item from one half adds the value from the other half to its value. This process is performed in a loop in which the number of work-items and the number of partitions decrease by half in every iteration until the number becomes one. By that time, the partial sum of this work-group is in the first position of the local cache. Every local work-item 0 (any work-item can do this) places this partial sum in global memory and is returned to the host side. The final sum is calculated on the host for efficiency reasons.

```
foreach work-item w (with local id lw) in parallel do
    local[lw] ← input[w];
end
foreach partition p in parallel do
    while partitionSize > 0 do
        partitionSize ← partitionSize/2;
        foreach local work-item lw in parallel do
            local[lw] ← local[lw] + local[lw + partitionSize];
        end
    end
    if lw is 0 then
        output[p] ← local[0];
    end
end
```

Algorithm 9: REDUCE

3.2.9 REDUCE BY KEY

The implementation of REDUCE BY KEY is a little more involved than that of UNIQUE (Algorithm 8), but differs significantly from that of REDUCE (Algorithm 9). REDUCE BY KEY works on key-value pairs and reduces or accumulates the elements in the value

attribute that correspond to consecutive duplicates in the key attribute. This operator also consists of four OpenCL kernels as described in SELECT, and the last three are exactly the same. In the first kernel, Algorithm 11, simple partitioning is done as before, and each work-item in the work-group puts the element on the index in the same way as the the work-item ID and its next adjacent element in registers. All the steps performed in Algorithm 8 are repeated in this kernel for the key attribute and are not shown in REDUCE BY KEY (Algorithm 11). For the value attribute, we initialize another register to determine the actual position of tuples of the value attribute. To calculate the value of the flag, local work-item 0 saves a value of 1 in the flag, and the remaining work-items in the work-group compare the values of the elements of the value attribute in two registers to check for equality. If they are same, the work-item sets the flag value at 0; otherwise, it is set at 1. Then, to determine the position of the resulting tuples for the value attribute, the work-group computes the inclusive scan (Algorithm 10) on flags. The work-items that evaluated unique keys place corresponding values in the local memory at the respective index determined by the scan and then waits on the barrier. After synchronization, the work-items with flag value 0 perform the atomic addition of the values in the register and the values stored at positions corresponding to duplicate keys to avoid the race condition. Finally, the resulting tuples in the local memory are transferred to the global memory in bulk. The reduced number of tuples in each partition is saved in an intermediate buffer. The corner case is handled in the same way as it is in UNIQUE. In this implementation, in the case of duplicates at the edges of the partition, the value in the former partition is added to the value of the later partition, and the size of the former partition is decreased by one.

```
foreach work-item w (with local id lw) in parallel do
    localBuffer ← input[w];
    for i ← 0 to numOfPartitions do
        if numOfPartitions > 2^i then
            if lw ≤ 2^i then
                localBuffer[lw] ← localBuffer[lw − 2^i] + localBuffer[lw];
            end
        end
    end
    output[w] ← local[lw];
end
```

Algorithm 10: Inclusive scan

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

Algorithm 11: REDUCE BY KEY

CHAPTER 4

EXPERIMENTATION

This chapter describes the target platforms used in this study and the results of the evaluation of our library on these platforms. The goal is to cover almost all the main architectural variants of CPU and GPU integration.

4.1 Target Platforms

In our experiments, we used CPU- and GPU-only homogeneous architectures and discrete and integrated heterogeneous architectures. In the discrete heterogeneous architectures (discrete GPUs), the PCI-e bus connects the CPU to the discrete GPU whereas in integrated heterogeneous architectures, the CPU and the GPU reside on the same die (fused CPU-GPU). Our CPU-only homogeneous architectures include the CPUs of the Intel Ivy Bridge i5-3470 machine and the AMD A10-5800K APU (Trinity) machine, and our integrated heterogeneous architectures consist of fused GPUs, Intel HD2500 graphics, and ATI Radeon HD7660D of the Ivy Bridge and Trinity. For our example of heterogeneous discrete architecture, we connected the ATI Radeon HD 5870 GPU to the CPU of the AMD APU via the PCI-e bus. We also used the HD 5870 GPU as an example of the GPU-only homogeneous architecture. The specifications of machines we used in our experiments are provided in Table 3. A brief description of each architecture is provided in the following sections:

4.1.1 Intel Ivy Bridge CPU Architecture

Figure 4 taken from [26] represents the main features of the Intel ivy bridge architecture. As specified in Table 3, our experimental machine, i5-3470, has a quad-core CPU with a three-level cache hierarchy. Each core, an x86-64 core, has a dedicated level 1 (L1) and level 2 (L2) cache. The third level L3 cache, which is also the last level cache, is shared by four CPU cores, the GPU, and the system agent via a ring bus. The system agent holds

Table 3: Specifications of Machines Used

	CPUs		GPUs		
			Fused GPU		Discrete GPU
Processor Core Names	Intel CPU i5-3470	AMD A10-5800K APU	Intel HD Graphics 2500	AMD Radeon HD 7660D	AMD Radeon HD 5870
Heterogeneous vs Homogeneous	Homo	Homo	Hetero	Both	Hetero
Compute Units	4	4	6	6	20
Shaders	N/A	N/A	768 (16-way SIMD * 6EUs * 8-way threaded)	384 (16PEs * 6CUs * 4 (VLIW4))	1600 (16PEs * 20CUs * 5 (VLIW5))
Max TDP	77W	100W	77W	100W	50W
Clock Speed	3.2GHz (Turbo boost 3.6GHz)	3.8GHz (Turbo boost 4.2GHz)	650 MHz (maximum dynamic 1.1GHz)	800MHz	850MHz
Cache Size	L1=128KB (32KB*4), L2=1MB (256KB*4), L3=6MB	L1=64KB (16KB*4), L2=4MB (2MB*2), L3=0	GRF (Graphics Register File) = 512KB, Local Memory = 64KB, L3 = 256KB	VGPRs = 256*6 (128-bit), Local Memory = 192KB, L1=48KB (8KB per compute unit), L2=128KB (64KB per memory channel), L3 = 0	VGPRs = 256KB, Local Memory = 640KB, L1=160KB (8KB per compute unit), L2=512KB (64KB per memory channel), L3=0
Memory Size	16GB	16GB	2GB	512MB	1GB
Maximum Memory Bandwidth	25.6 GB/s	29.9 GB/s	25.6 GB/s	29.9 GB/s	153.6 GB/s

a couple of interfaces and links such as a PCI-e interface, a direct media interface (DMI) link, a dual-channel DDR3 memory controller, and a display controller, not shown in the architecture diagram for simplicity. The GPU is on the same die in the ivy bridge, and its architecture is explained in Section 4.1.3.

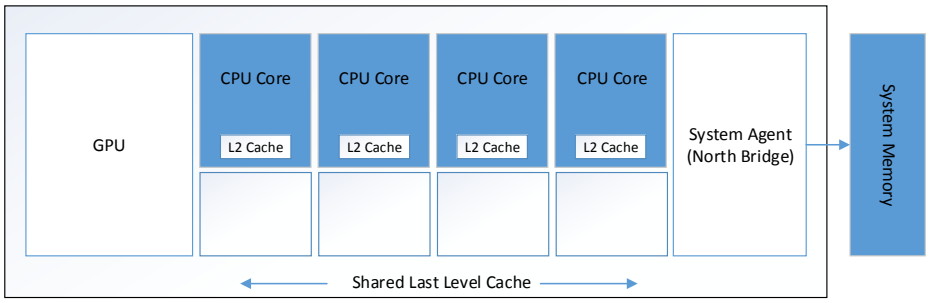


Figure 4: Intel Ivy Bridge Architecture

4.1.2 AMD Trinity CPU Architecture

Figure 5 taken from [27] shows the prominent features of the CPU architecture of AMD A10 Trinity APUs. We used the AMD A10-5800K machine containing the trinity APU, which has quad-core Piledriver CPU cores. Unlike the Intel ivy bridge, it has a two- instead of a three-level cache hierarchy. The architecture also contains a DDR3 memory controller, a PCIe interface, and other links and interfaces not included in the figure so that relevant features are emphasized. As evident from Figures 4 and 5, Intel has dedicated more space to the CPU while AMD has dedicated more area to the GPU.

4.1.3 Intel HD 2500 GPU Architecture

The architecture of the GPU in the Intel i5-3470 machine in our experiments are illustrated in Figure 6 taken from [28]. The architecture contains six execution units, labeled EUs in the figure. Each EU is a 16-way SIMD and 8-way threaded core. The 8-way threading refers to the number of SMT threads, or hyperthreads, that each of the six execution units (EUs) of the HD 2500 GPU have. Since there is less data access locality than on CPUs, GPUs make use of multiple hyperthreads to minimize or even avoid the high cost of memory misses. This Intel GPU does not have an L1 or L2 cache, but it contains register files

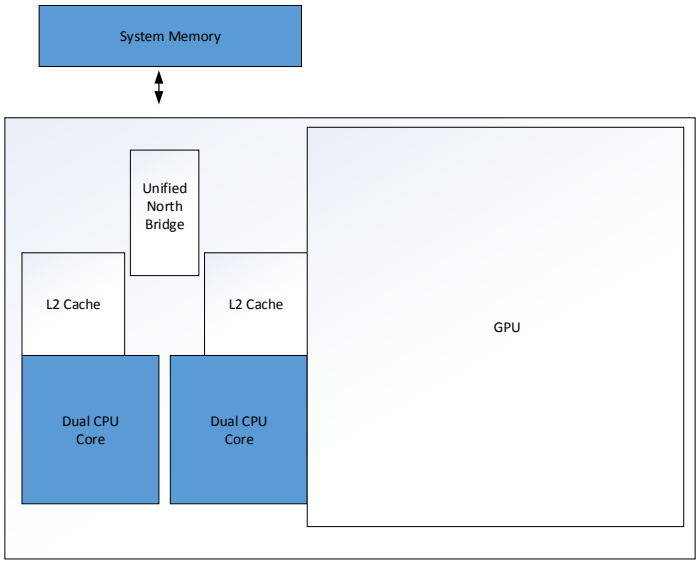


Figure 5: AMD Trinity Architecture

dedicated to work-items and local memory shared by all work-items in a work-group. The global memory is mapped to the L3 cache, shown in the figure.

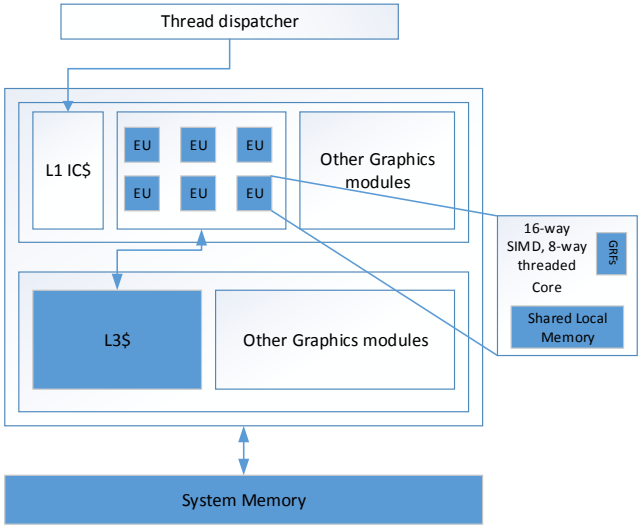


Figure 6: Intel HD2500 Architecture

4.1.4 AMD Northern Island and Evergreen GPUs Architecture

Figure 7 taken from [29] shows the architecture of both AMD GPUs used in our experiments. Both of these devices contain several compute units with unique structures and numbers depending on the GPU family they belong to. The discrete AMD GPU, HD 5870,

is a part of the AMD Evergreen GPU family and the AMD integrated GPU, HD 7660D, is a part of the AMD Northern Islands GPU series. A compute unit is comprised of numerous processing elements, each of which has multiple ALUs. In the Evergreen GPU, the processing element contains ALUs in a 4-way VLIW (VLIW4) configuration and in the Northern Islands GPU, they are a 5-way VLIW (VLIW5) configuration. One work-item works on one processing element, that is, 4 or 5 ALUs at a time. Both AMD GPUs contain L1 and L2 caches along with general purpose registers dedicated to one work-item and the local memory dedicated to work-items in a work-group.

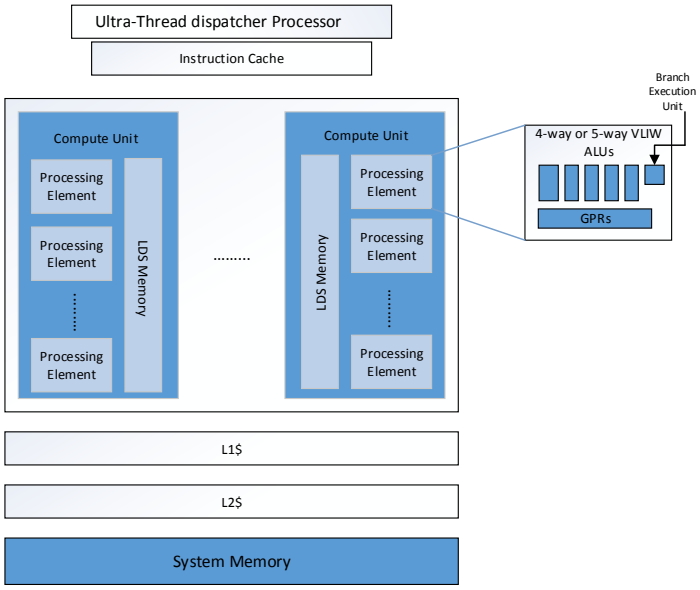


Figure 7: AMD GPU Architecture

4.2 Performance Evaluation of Primitives

We did our performance evaluation using Visual Studio 2012 on Windows 7 and used The OpenCL 1.2 Specification [16]. The reported execution time is obtained after averaging the time obtained after each primitive is run 10,000 times. The y-axis is on a logarithmic scale and each value is obtained after taking log to the base 10 of the execution time in microseconds, and the x-axis represents the size of each input relation (i.e., the number of tuples * size of each tuple). To evaluate the library, we assign a single attribute as an input

to UNIQUE and REDUCE and for other operators, we provided each of them with the input tuples that comprise of one key and one value attribute. The legend accompanying the following figures is provided in Figure 8.

Figure 8: Common legend for graphs representing the execution time

We did not use any special memory flags to obtain the results which means that input data is transferred to the GPU memory and resides there during computation. The input data size is bound by the maximum buffer size limit that is one-fourth of the global memory size of the GPU, imposed by all OpenCL vendors. The minimum ”maximum memory allocation limit” across the devices we are using is on the AMD fused GPU and that is 200,540,160 bytes. Therefore, we used 128MB size buffer as the the maximum input buffer size. To illustrate the kernel execution time, the time spent by the primitive on the OpenCL device, we did not take the data transfer time and the kernel invocation time into account. Figure 9 depicts the performance achieved by a simple primitive PROJECT that is a highly parallelizable operator. As evident from the graph, the discrete GPU is outperforming other devices. This is because of the fact that the discrete GPU has more compute units and higher memory bandwidth, thus, taking the least time for execution. Figure 11 shows the execution time of an irregular operator INNER JOIN. The discrete GPU is again executing it faster for the same reason. Only in the case of PRODUCT (Figure 10), CPUs perform better because PRODUCT is not only memory bound, but also requires a large amount of memory. CPUs have larger caches than GPUs and hence they perform better in this case. Figures that represent the kernel execution time of all primitives on all devices can be found in Appendix A.

Even though the discrete GPU is executing all kernels faster, a significant amount of time is spent in transferring inputs and outputs between the host and the device. In case

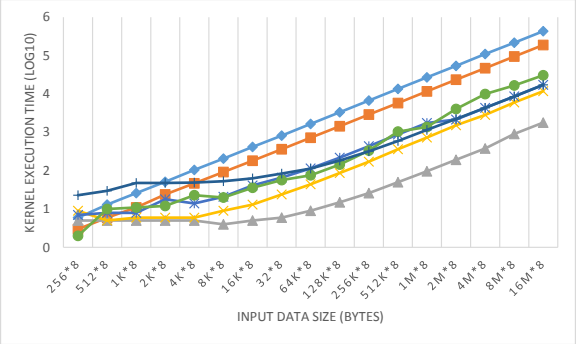


Figure 9: The integrated GPU outperforms CPUs and even integrated GPUs in the case of the very simple operator PROJECT

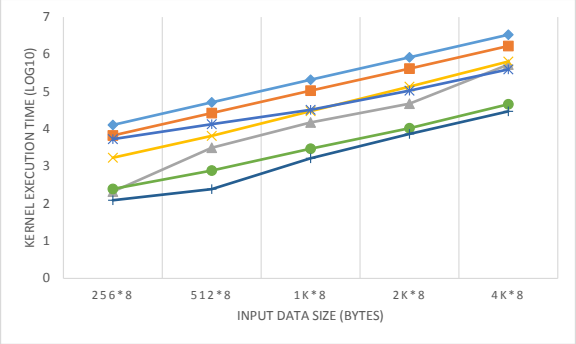


Figure 10: CPUs perform significantly better than GPUs for the primitive PRODUCT, which requires a large amount of memory

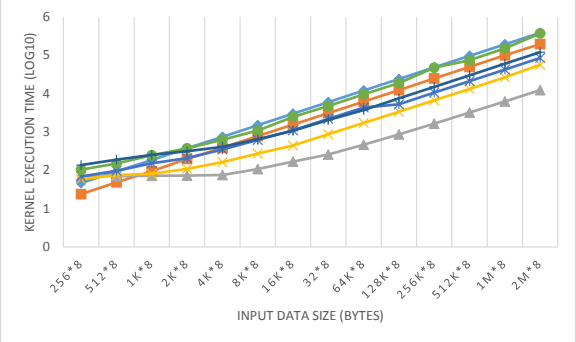


Figure 11: The discrete GPU executes the compute-intensive operator INNER JOIN faster than other devices

of simple primitives such as PROJECT, the kernel execution time is a very small fraction of the total execution time because the rest of the time is spent on transferring data as can be seen in Figure 12. Transferring data from the host buffer to the device buffer and vice versa is relatively faster in integrated GPUs and CPUs than the discrete GPU. This slow transfer in the discrete GPU is due to the slow PCI-e bus. Moreover, integrated GPUs and CPUs execute kernels slowly than the discrete GPU. Because of these two reasons, devices other than the discrete GPU spend more time in execution compared to the data transfer, especially in the case of complex primitives such as INNER JOIN (Figure 14) and PRODUCT (Figure 13). These graphs are giving a pictorial view of the normalized execution time spent on transferring data and executing kernels. In these graphs, each bar from five bars corresponding to one input size represents one device. The lower parts that are in the shades of green represent the percentage of the total time spent on executing kernels and upper portions in the shades of blue show the percentage of the total time taken to transfer data between the host and the OpenCL device. In some graphs, a two is multiplied with the input size on x-axis, which shows that the primitive takes two relations as input.

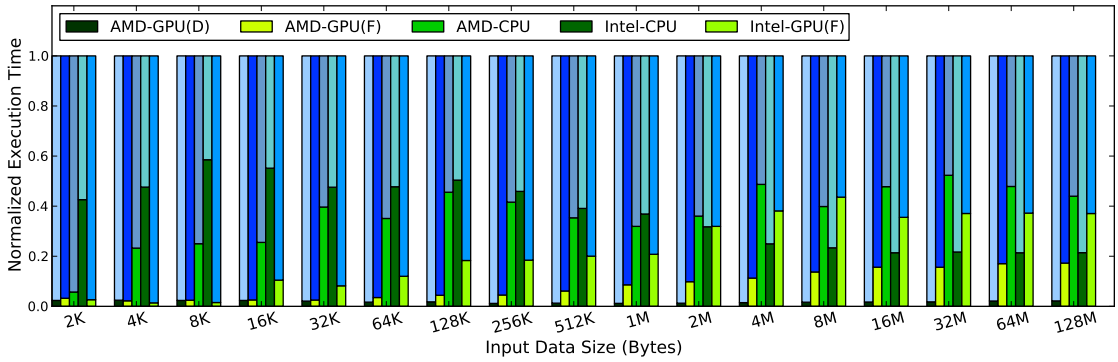


Figure 12: The normalized execution time of the simplest operator PROJECT shows that devices spend more time in transferring data than that in actual execution

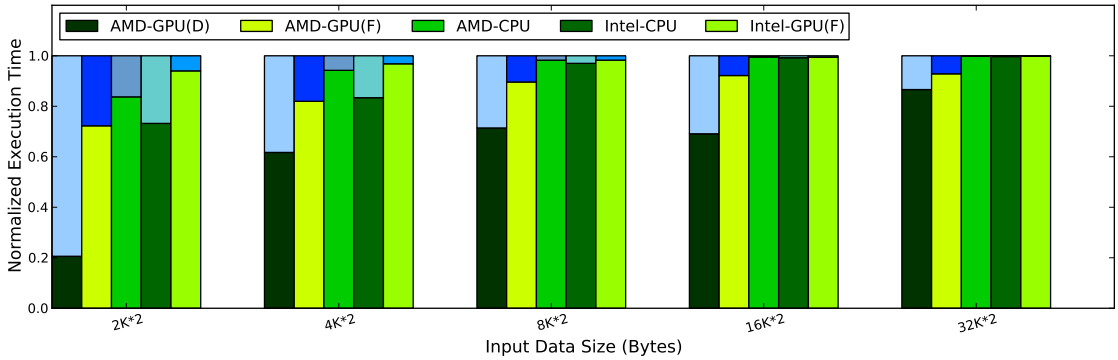


Figure 13: The execution of the PRODUCT operator is very slow on GPUs, therefore, the percentage of the time spent on transferring data is less than that on execution. The data transfer time is negligible in case of CPUs

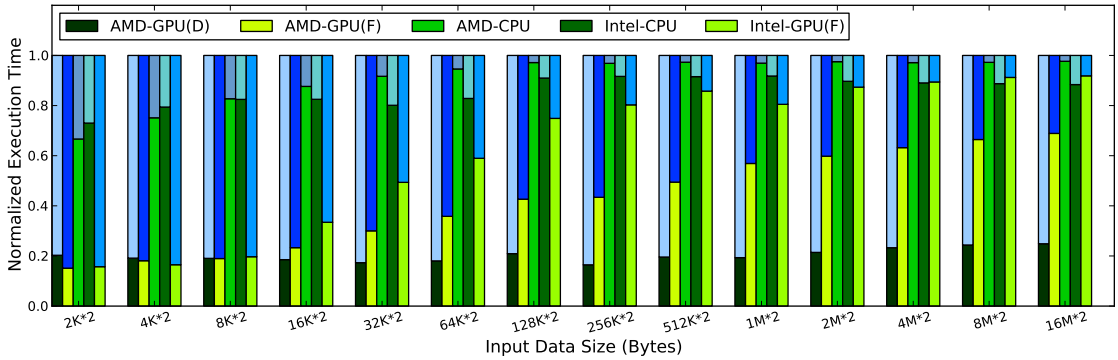


Figure 14: In the case of the complex operator INNER JOIN, the discrete GPU operates the fastest but the data transfer time is significantly large. Integrated GPUs and CPUs are very slow in executing the operator and the addition of their data transfer worsens their performance

4.3 Micro-benchmarks/RA Kernels

After a detailed analysis of TPC-H queries [13], 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 15 from [13]. From now on, we will call them (A), (B), (C), (D), and (E). 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. (D) applies different filters on the one input relation to obtain different results; and (E) performs arithmetic operations that are common in TPC-H queries.

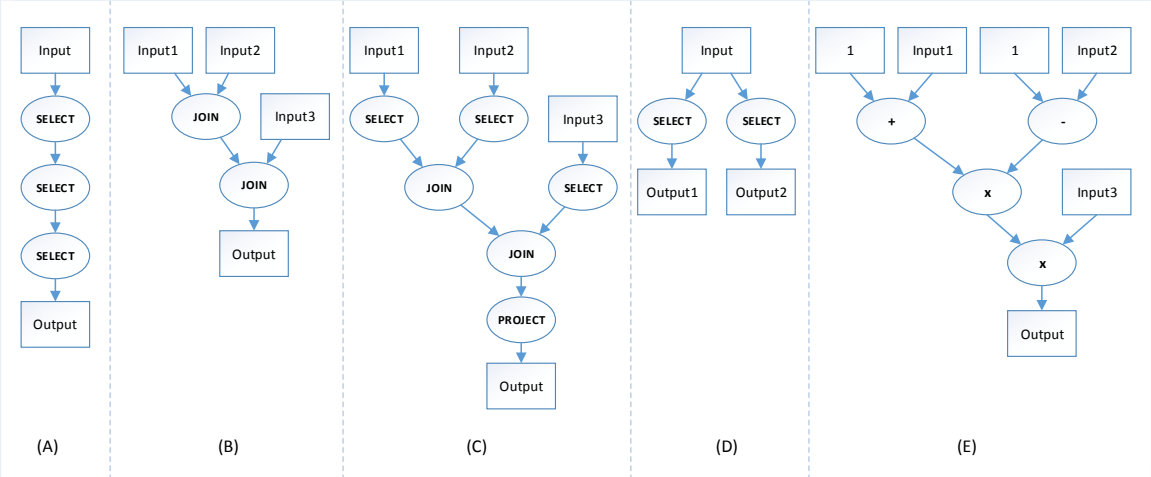
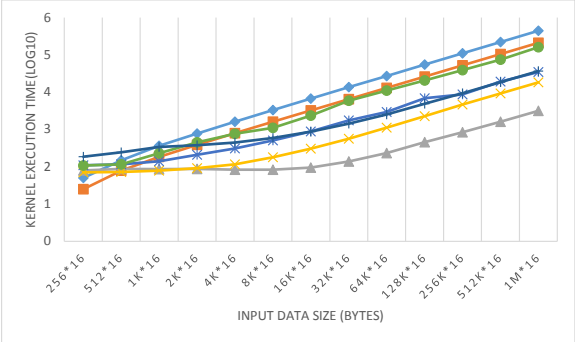


Figure 15: Micro-kernels from TPC-H queries

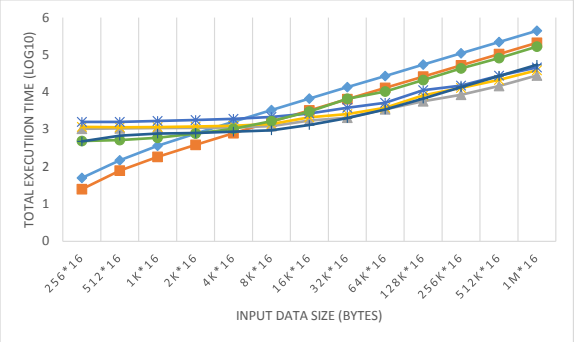
4.4 Performance Evaluation of Micro-benchmarks

Graphs/Figures in this section use the legend in Figure 8. The x-axis in graphs represents the size of the input relation if only one input is used and the size of each input relation in case of two inputs (number of tuples in the input relation * size of each tuple). The

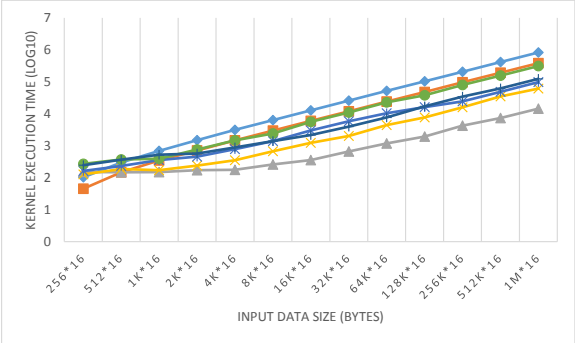
y-axis is the execution time in microseconds on a logarithmic scale (base 10). Figures 16a, 16c, 16e, 16g, and 17a illustrate the kernel execution time of these micro-benchmarks on the y-axis taken by each device. These graphs do not include the kernel launch time and the data transfer time. As evident from these graphs, the discrete GPU executes all the micro-benchmarks in the least amount of time. Integrated GPUs are next in line followed by CPUs. When the total execution time is plotted against the input size, which includes the kernel launch time, kernel execution time, and data-transfer time, results come out to be slightly different as can be seen in Figures 16b, 16d, 16f, 16h, and 17b. The performance of the discrete GPU is now comparable to that of integrated GPUs in case of all benchmarks except for micro-benchmark (E). In case of (E), the total time taken by the discrete GPU is more than some of the other devices. The reason is that kernels in (E) are very simple and take very little time to execute especially on the discrete GPU as shown in Figure 17a, but the time to transfer three input relations from the host to the device is more than the total execution time of kernels.



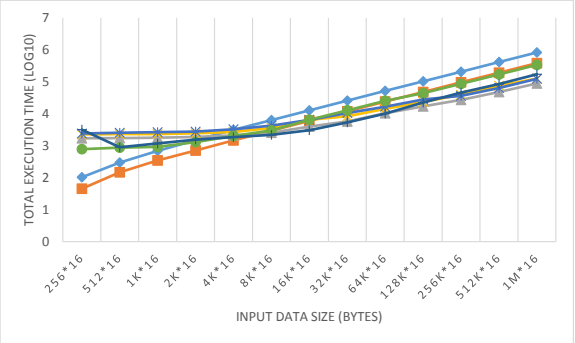
(a) The kernel execution time of (A)



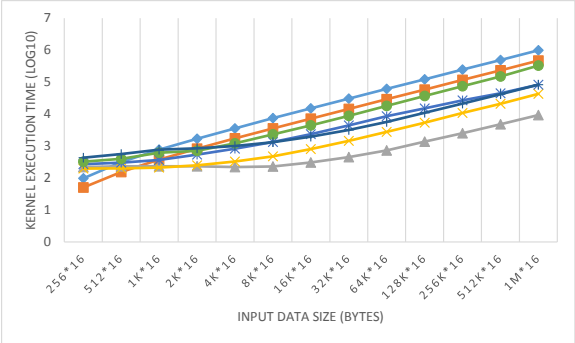
(b) The total execution time of (A)



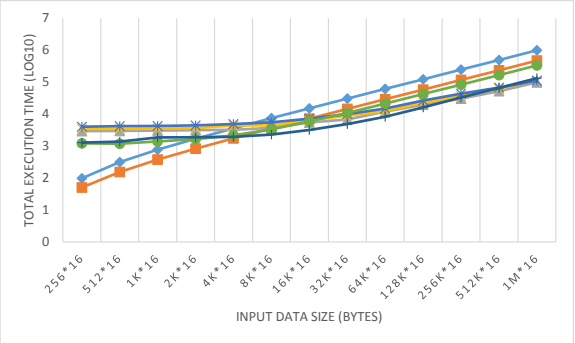
(c) The kernel execution time of (B)



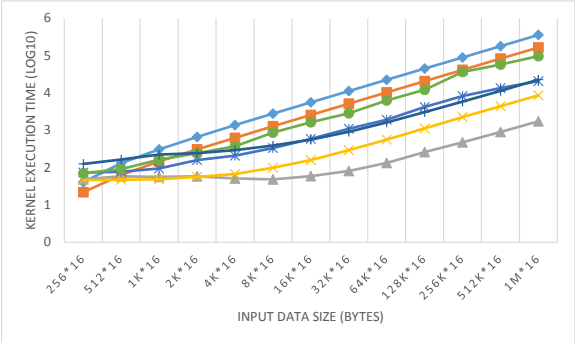
(d) The total execution time of (B)



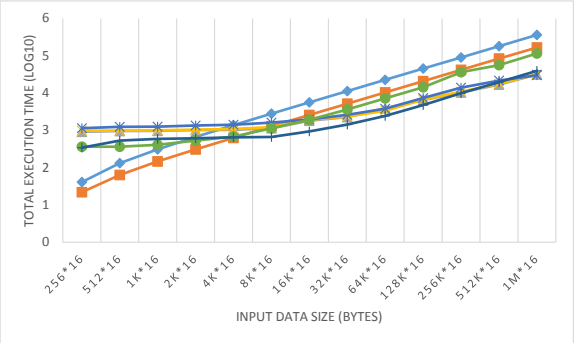
(e) The kernel execution time of (C)



(f) The total execution time of (C)

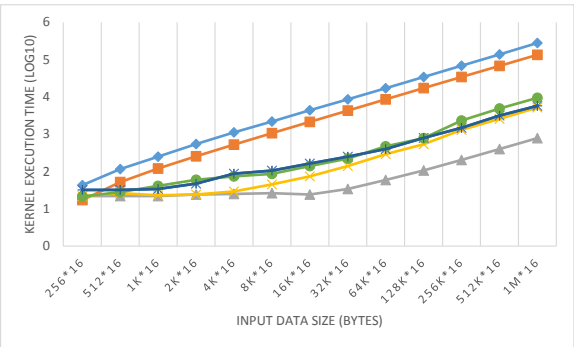


(g) The kernel execution time of (D)

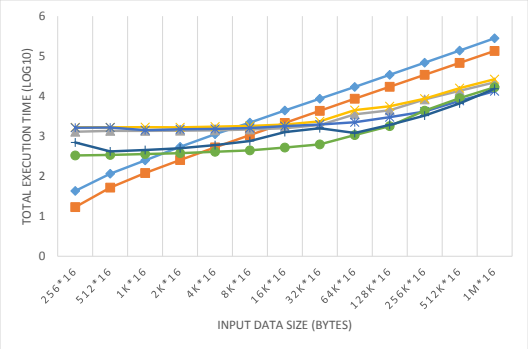


(h) The total execution time of (D)

Figure 16: For complex micro-benchmarks (A), (B), (C), and (D), the discrete GPU is taking the minimum execution time. The difference is quite significant from other devices when only the kernel execution time is considered. The discrete GPU performs fairly good even after adding the data transfer and the kernel launch time to the kernel execution time



(a) The kernel execution time of (E)



(b) The total execution time of (E)

Figure 17: For the simple micro-benchmark (E), the discrete GPU is outperforming other devices when only the execution time of kernels is measured. The inclusion of the data transfer and the kernel launch time degrades its performance

CHAPTER 5

LESSONS LEARNED

This thesis accomplishes the goal of providing the OpenCL implementation of primitives required for data-intensive relational query processing, which executes on a wide range of GPUs and CPUs. With this portability feature, all GPUs and CPUs in the system can be fully utilized at run time, which is currently not possible with CUDA implementation. The ability to fully utilize GPUs and CPUs in the system is a major contribution of this thesis. Several other observations from our experiments provide additional useful insights summarized as follows:

- Because of the large number of compute units, high memory bandwidth, and many on-chip resources, the discrete GPU takes the smallest amount of the execution time for all primitives when data fits in the device memory.

- The total execution time is the sum of the kernel execution time, the kernel launch time, and the data transfer time. In case of the discrete GPU, the data movement between the host/CPU and the device presents a performance bottleneck because the time to execute the primitives on the discrete GPU is much shorter than the data transfer time.

- 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 we are dealing with large data sets, the data transfer time cannot be reduced by using pinned memory, a scarce resource that cannot accommodate large 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.

- In our experiments, we limited the size of our input and output buffers to the minimum of the maximum allocation size on all devices. This limit differs across devices depending on the size of the global memory for GPUs and RAM in case of CPUs. The maximum allocation limit is a quarter of the available global memory or RAM. To run primitives for input sizes larger than the size of the maximum allocation size, we would have to perform multiple iterations or data transfers. Thus, in future work, to reduce the transfer time, we will pursue more innovative approaches such as transferring compressed data.

- The experimental results obtained from running the TPC-H micro-benchmarks also show that the execution time of kernels running on the more powerful discrete GPU is significantly less than that on other processors. After taking into account the data transfer time of multiple inputs and outputs between the host and the device, and buffers across kernels, the performance of the discrete GPU is slightly better than integrated GPUs in complex micro-benchmarks and worse in case of the simple arithmetic micro-benchmark.

- From our experimental results, we conclude 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.

Future steps will extend the Red Fox back-end so that it can transparently utilize OpenCL kernels, thereby extending the set of supported platforms.

CHAPTER 6

CONCLUSION

In this thesis, we have presented a portable library containing RA (relational algebra), arithmetic, and other related primitives required to run data-intensive relational queries. These primitives or operators are written in OpenCL and generally exhibit good performance across various platforms. In our experiments, we used multiple GPUs and CPUs to evaluate the library and concluded that since discrete GPUs are rich in resources and powerful, they outperform other integrated GPUs and CPUs. 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. When multiple primitives are executed in TPC-H micro-benchmarks, the performance of the discrete GPU and fused GPUs is comparable with the discrete GPU performing slightly better. 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.

CHAPTER 7

FUTURE WORK

The work presented in this thesis will be utilized and expanded in the following ways as future work:

- The OpenCL primitives library will be integrated with the ongoing Red Fox project at Georgia Tech. As already mentioned, Red Fox is currently supported only on Nvidia processors and the integration of our library with the project will enable it to utilize many other platforms.

- To achieve strong performance on a specific platform, primitives will be optimized based on the underlying architecture. This optimization will fully utilize the available features on the platform.

- Since the transfer of data from the host to the device and vice versa incurs the decline in device performance, the industry is coming up with unified memory address space between the host and the device to avoid the data transfer. This unified memory model will enable us to execute massive amounts of data sets removing the limitations imposed by the current slow transfer rates.

- The vendors of processors and accelerators are also adding a large number of powerful computation units and on-chip resources on their devices, which in turn will be beneficial in accelerating compute-intensive primitives.

- The big data market has been expanding at a very high rate. Massive amounts of data have been produced and consumed by industries, social media, sensor technology, and many other fields and the growth rate has been continuously increasing. This study will target all such big data applications to efficiently handle the queries used by these applications.

APPENDIX A

KERNEL EXECUTION TIME OF PRIMITIVES USING A BASE 10 LOGARITHMIC SCALE

The time taken by OpenCL devices to execute all primitives is presented in Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, and A.10. This time represents only the kernel execution time and does not show the data transfer time and the kernel invocation time. The y-axis is on a logarithmic scale and gives log to the base 10 of the execution time in microseconds, and the x-axis represents the size of each input relation (i.e., the number of tuples * size of each tuple). The legend is defined in Figure 8.

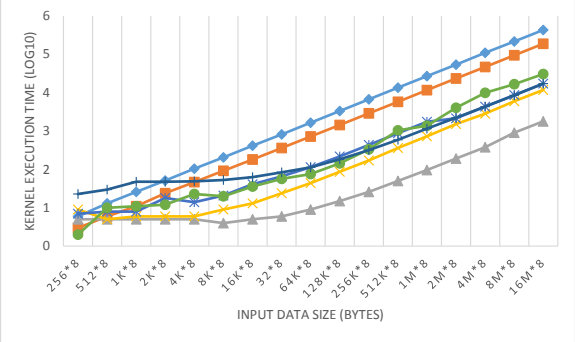


Figure A.1: PROJECT

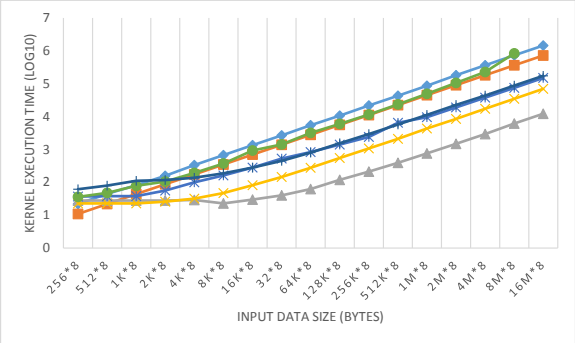


Figure A.2: SELECT

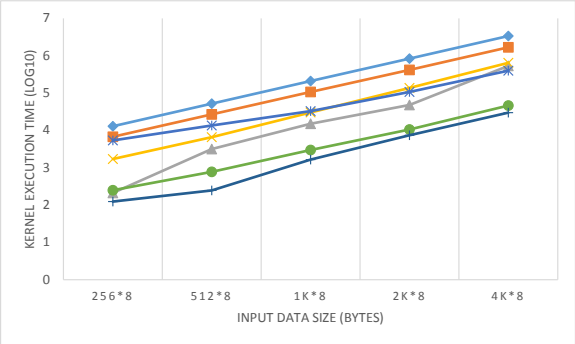


Figure A.3: PRODUCT

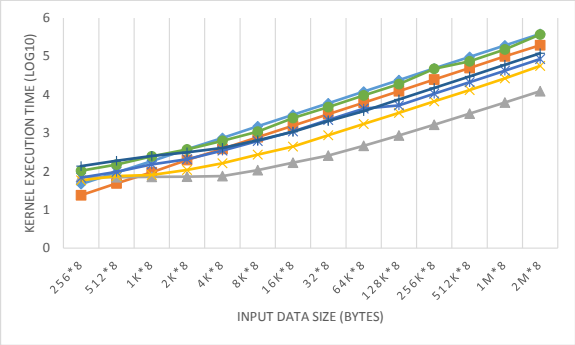


Figure A.4: INNER JOIN

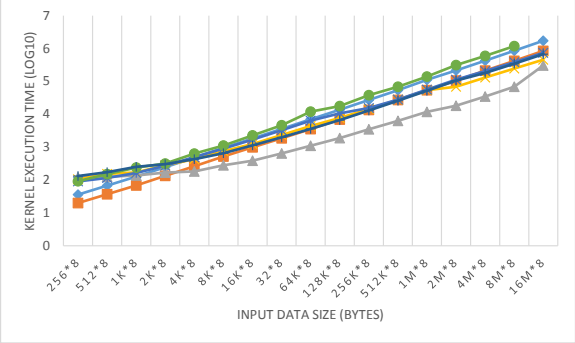


Figure A.5: SET INTERSECTION

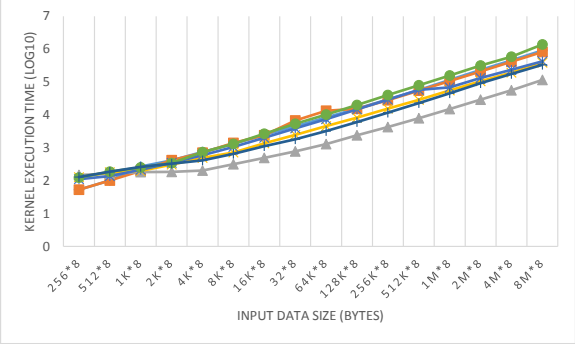


Figure A.6: SET UNION

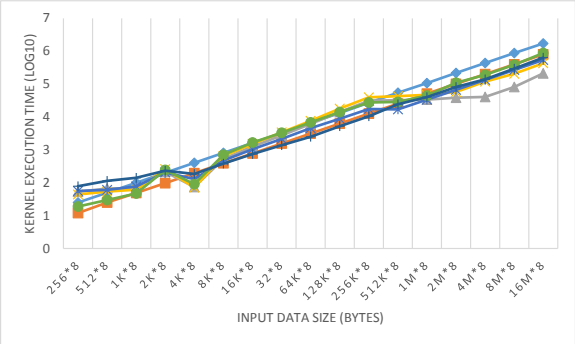


Figure A.7: SET DIFFERENCE

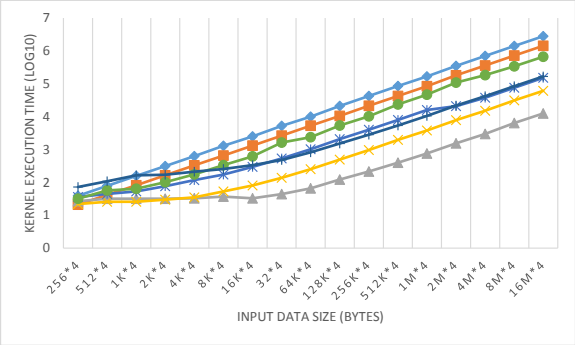


Figure A.8: UNIQUE

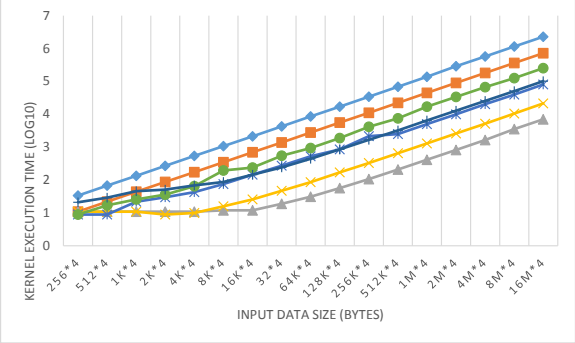


Figure A.9: REDUCE

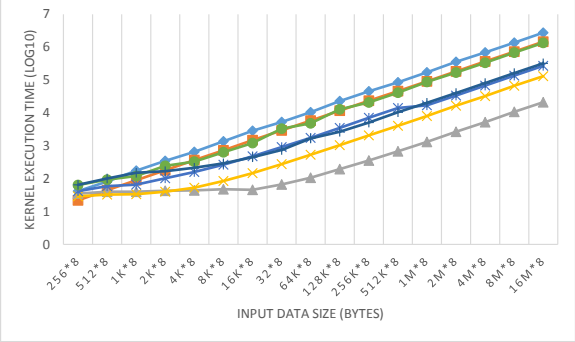


Figure A.10: REDUCE BY KEY

APPENDIX B

NORMALIZED KERNEL EXECUTION AND DATA TRANSFER TIME

Figures B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8, B.9, and B.10 illustrate the time spent on transferring data between the host and the device and executing kernels of primitives on all devices in the normalized form. The x-axis represents the input data size in bytes and the y-axis shows the normalized execution time. Corresponding to each input data size, each bar represents the normalized time of one of the five OpenCL devices used for our experiments. The bottom bars in the shades of green exhibit the kernel execution time and the bars in the shades of blue on the top of green bars depict the data transfer time. Note that the execution time is normalized, which implies that graphs show the percentages of the total time spent on the execution of kernels and transfer of data. In some graphs, a two is multiplied with the input data size, which indicates that these primitives take two input relations. In the legend, D and F written in brackets with GPUs represent discrete and fused respectively.

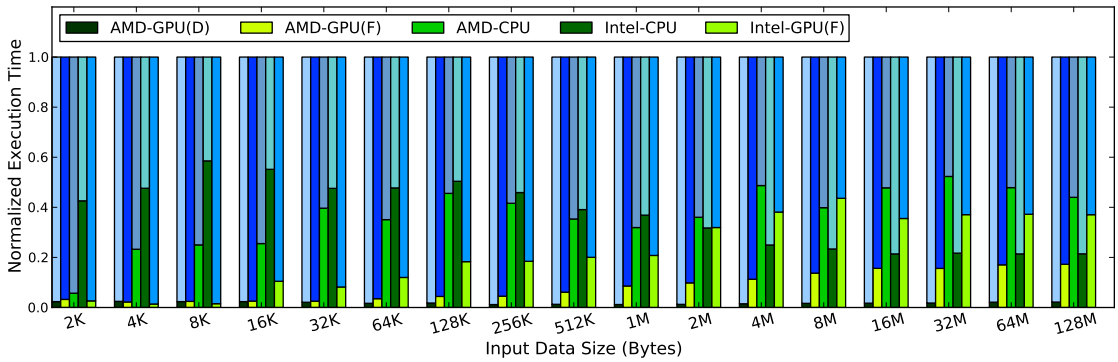


Figure B.1: PROJECT

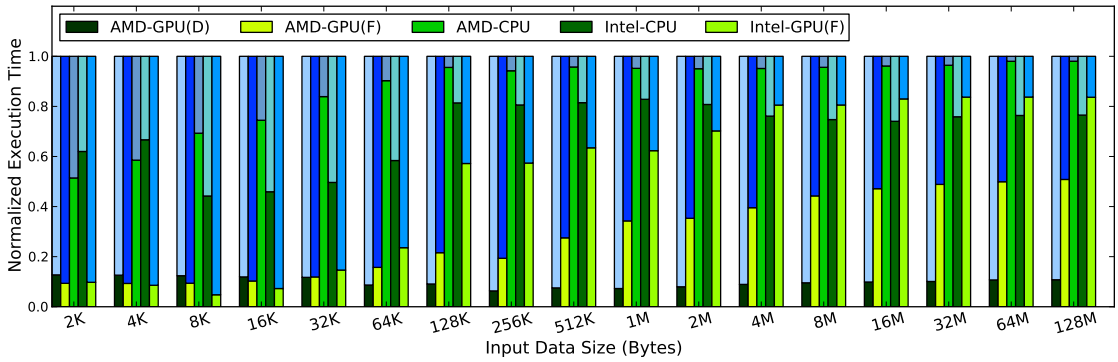


Figure B.2: SELECT

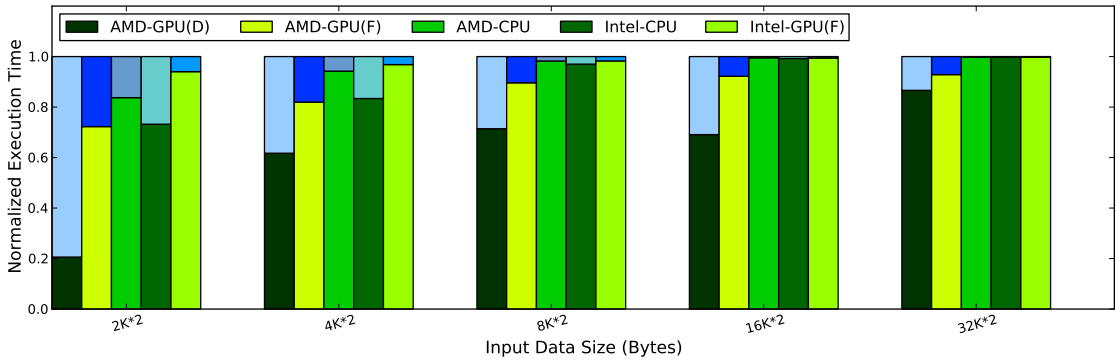


Figure B.3: PRODUCT

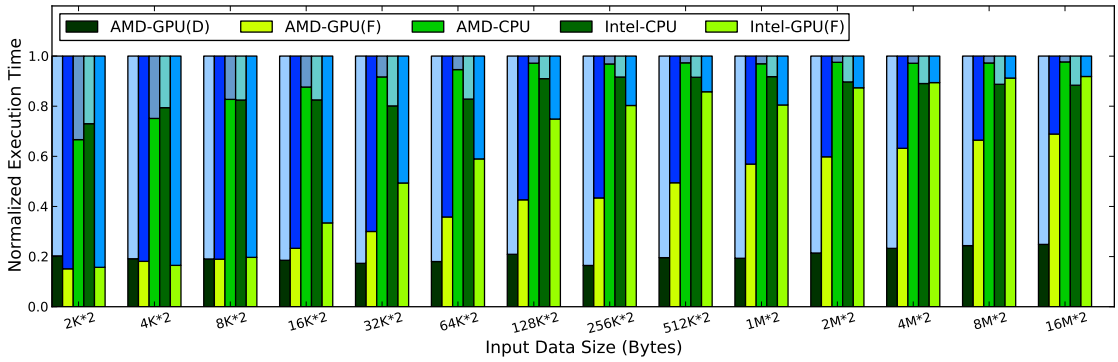


Figure B.4: INNER JOIN

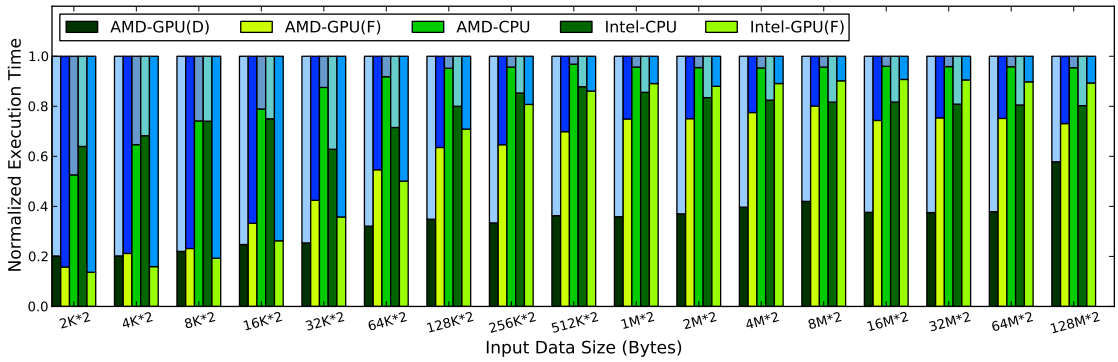


Figure B.5: SET INTERSECTION

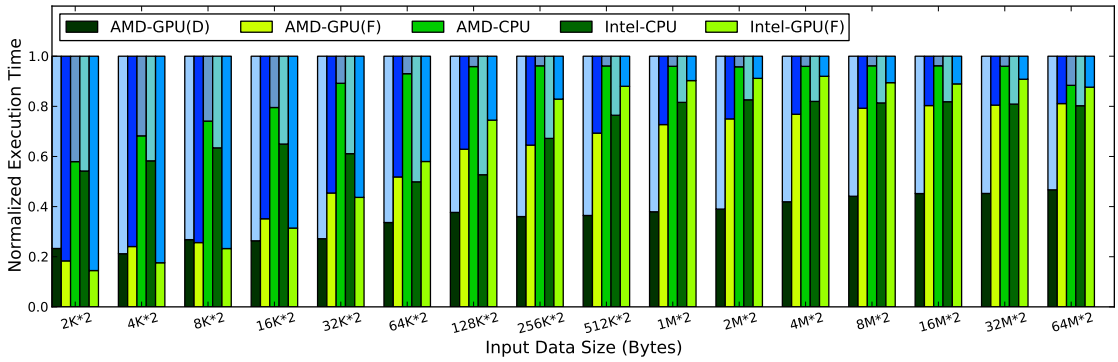


Figure B.6: SET UNION

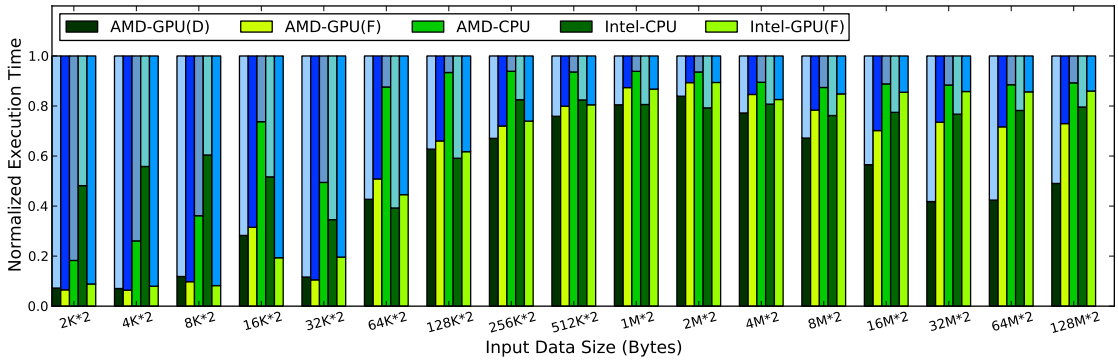


Figure B.7: SET DIFFERENCE

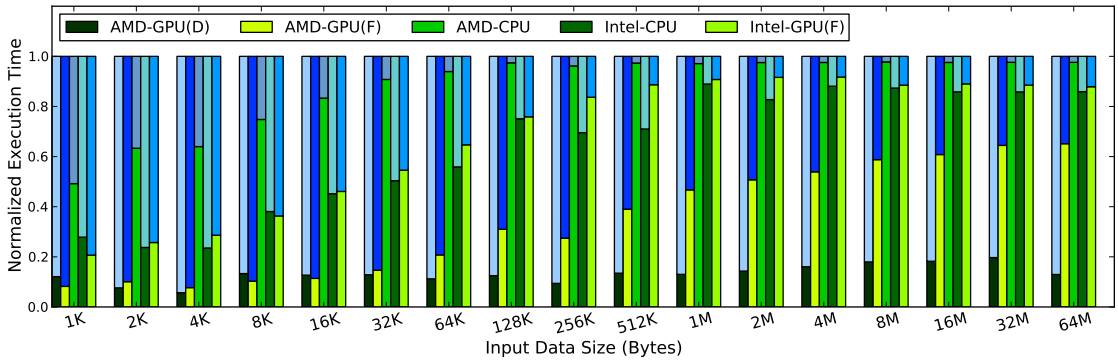


Figure B.8: UNIQUE

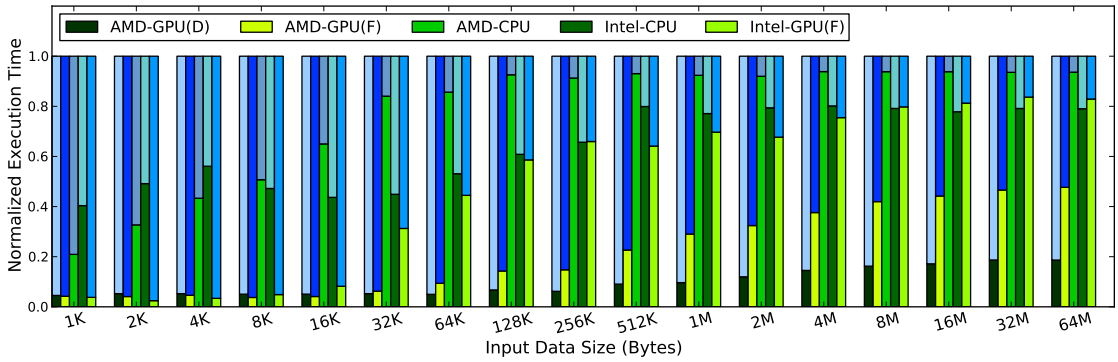


Figure B.9: REDUCE

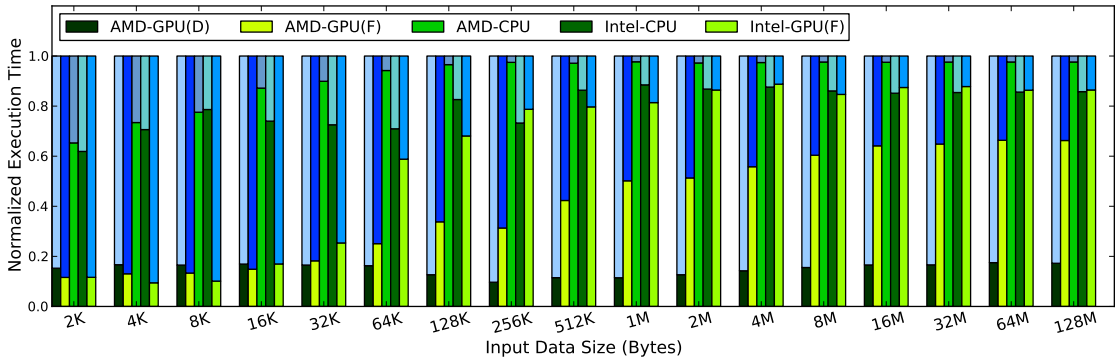


Figure B.10: REDUCE BY KEY

REFERENCES

[1] 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.

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

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

[4] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.

[5] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 134–144, IEEE, 2011.

[6] AMD, "AMD unveils innovative new APUs and SoCs." `http://www.amd.com/us/press-releases/Pages/amd_unveils_new_apus.aspx`, 2013.

[7] Intel, "Intel HD Graphics and Iris Brand Maximize Visual Computing." `http://www.intel.com/content/www/us/en/architecture-and-technology/hd-graphics/hd-graphics-iris-pro.html`.

[8] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, 2011.

[9] TechRadar, "AMD's Southern Islands graphics explained." www.techradar.com/us/news/computing-components/graphics-cards/amd-s-southern-islands-graphics-explained-1066145, 2012.

[10] Nvidia, "Kepler - the world's fastest, most efficient hpc architecture." http://www.nvidia.com/object/nvidia-kepler.html.

[11] C. NVIDIA, "C Programming Guide, April 2013."

[12] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: a parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[13] 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*, pp. 107–118, IEEE Computer Society, 2012.

[14] J. Melton, *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993.

[15] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: an interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1213–1216, ACM, 2011.

[16] K. O. W. Group *et al.*, "The OpenCL Specification." http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2011.

[17] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[18] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on GPUs," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 147–156, ACM, 2013.

[19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[20] 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.

[21] J. Young, S. Yalamanchili, B. Holden, and M. Cavalli, "HyperTransport over Ethernet-a scalable, commodity standard for resource sharing in the data center," 2011.

[22] T. P. P. Council, "TPC Benchmark H Standard Specification Revision 2.8. 0 (2008)." http://www.tpc.org/tpch/spec/tpch2.16.0.pdf.

[23] D. Demidov, "VexCL: Vector expression template library for OpenCL http://www. codeproject. com/articles/415058."

[24] S. Baxter, "Modern GPU." http://nvlabs.github.io/moderngpu/.

[25] P. Rogers and A. C. FELLOW, "Heterogeneous system architecture overview," in *Hot Chips*, 2013.

[26] "Intel's Ivy Bridge CPU Die Layout Estimated." http://www.tomshardware.com/news/Intel-Ivy-Bridge-LGA1155-Die-Layout,14782.html.

[27] "AMD A10-5800K Trinity APU Review." http://www.techspot.com/review/580-amd-a10-5800k/.

[28] "Intel's Ivy Bridge Architecture Exposed." http://www.anandtech.com/show/4830/intels-ivy-bridge-architecture-exposed/5.

[29] J. Wang, N. Rubin, H. Wu, and S. Yalamanchili, "Accelerating simulation of agent-based models on heterogeneous architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 108–119, ACM, 2013.

[30] D. Maier, *The theory of relational databases*, vol. 11. Computer science press Rockville, 1983.

[31] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, pp. 117–128, ACM, 2012.