HASTE: A HETEROGENEOUSLY ACCELERATED SQL TRANSACTION ENGINE

by

Brian M. Romoser, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2014

DEDICATION

This thesis, my education, and all the experiences I've gained, wouldn't have happened the way they did without the people who have shared my life. Whether they were my classmate, my friend, my teacher, my partner, or simply someone I once met, they all cast ripples across a pond, setting me on my course.

I dedicate this thesis, the culmination of my educational career to date, to the people who most directly affected my life:

My parents, for providing the foundation of my life and shaping my personality.

My professors, for guiding, supporting, motivating, and driving my passion for learning.

My friends, for being a constant reminder that I'm never alone.

My loving partner Kim, who has taught me more than I could ever express with words. She has taught and supported me more than I could believe any person could. She has, quite simply, shown me the meaning of the words "I love you."

Ultimately, this thesis is dedicated to myself. As I walk from one path in life to another, gaining experiences and overcoming struggles, I endeavor to never lose sight of the only true constant in life and the most important thing to me. I have always, in some ways, tried to follow its meaning, but it wasn't until my early adult years that I truly began to understand the implications.

gnōthi seauton

ACKNOWLEDGEMENTS

Many countless thanks to…

My thesis advisor, Dr. Ziliang Zong, who taught me the fundamentals of research as an undergraduate and instructed me daily throughout my graduate education. In the many publications we authored together, he showed me how to be a diligent student and an eager researcher.

Dr. Martin Burtscher, for his amazing depth of knowledge in the fields of computer architecture and parallel programming. His courses always excited and engrossed me and gave me a reason to show up every day.

Dr. Anne Ngu, for her valuable insight into the workings of databases and her detailed explanation of the fine details of the science they embody.

Dr. Lee Koh, my first Computer Science professor, who taught me to always ask questions.

My esteemed classmates Ivan Sutanto, Richard Hay, Ribel Fares, and Scott Julian

John Bartko, for always being a good friend and keeping my spirits up.

Gabriel Flores, who came with me to college from high school and showed me the enjoyment to be had from making friends.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Databases are the backbone of the digital age and empower the storage and processing of massive amounts of data. As private and public data grows at an astonishing rate, the technologies that drive data processing must adapt or be discarded. Conventional database engines struggle to provide responsiveness at the level required of them when faced with ever-expanding datasets and more demanding use cases. With the recent surge in public adoption of hardware parallelism and co-processor offloading, we have explored the concept of employing new parallel processing techniques and technologies to a database management system (DBMS) to achieve higher query processing performance.

In this paper, we demonstrate a custom-designed, modular DBMS targeted at parallel platforms including the Intel Many Integrated Core architecture and an Nvidia CUDA platform. Our Heterogeneously Accelerated SQL Transaction Engine (HASTE) uses a novel query parsing methodology to create a hardware agnostic query definition which can be processed by adaptable modules written for new and existing hardware and software platforms and executed on one more such modules simultaneously. This paper demonstrates these modules designed for a modern CPU, Xeon Phi 5110 co-processor from Intel, and Tesla K20 GPGPU from Nvidia, but can also be extended to run on virtually any technology that interfaces with the HASTE host kernel. Through experimenting on both synthetic and real-world data, we achieve a speedup of up to 2000 percent with the Xeon Phi and 6700 percent with the Tesla hardware.

I. INTRODUCTION

a. Need for Faster Database Processing

Databases are undisputedly one of the most critical facets of the Information Age for businesses, governments, and individuals alike, with the number of database administrators growing by over 15% per year (Bureau of Labor Statistics, US Department of Labor, 2014). Databases can be utilized to store financial transactions, asset inventories, personal records, and serve purposes big and small in all corners of modern life (Gartner, 2013). Today, the amount of information stored by the largest of organizations easily exceeds many petabytes in size, and in 2009 the estimated information storage capacity of the world was estimated at 297 exabytes (297,000,000,000 gigabytes) (Hilbert & Lopez, 2011). Utilizing data stores of such massive sizes requires much advancement in large-scale processing of databases. Efficiency in data retrieval is of paramount importance for large-scale organizations to support continued information growth – Facebook's HADOOP database, for example, grew by 10 petabytes from 2010 to 2011 (Thusoo, et al., 2010).

Current research and industry advancements in the area of database acceleration have employed hardware parallelization techniques to achieve higher throughput of data processing utilizing a range of computing devices. The HASTE project (a Heterogeneously Accelerated SQL Transaction Engine) focuses on employing two emerging hardware platforms: general purpose graphics processing units (GPGPUs) and many integrated core (MIC) processors, along with traditional CPU parallelization techniques. GPUs have been used for high performance computing for many years and have been shown to provide an improvement of over 5,000% compared to single-threaded code on a CPU (Bakkum & Skadron, 2010). Unlike GPUs, MIC coprocessors are a far more recent entry into the hardware market, with the first commercial implementation – Intel's Xeon Phi – released in 2012. Both GPU and MIC devices have the potential for highly scalable performance, thanks to the presence of several thousand low-powered thread processors within GPUs and up to 60 traditional processing cores on the Xeon Phi.

b. Overview of HASTE

HASTE is a modular database engine supporting hardware-agnostic execution modules, based upon the GPU-centric design of the Virginian project published by the University of Virginia (Bakkum & Skadron, 2010). HASTE consists of a host thread acting as a front-end interface to the user and a back-end controller for data transfer and query preparation. Our database engine behaves in a manner consistent with standard popular databases, receiving a SQL statement following generally accepted syntax and returning to the user a table consisting of selected rows and columns from the source database which match the provided query. We implement a query translation system that enables our use of interchangeable execution modules by creating an intermediary state for the SQL transaction. Specifically, the user‘s query is transformed into a set of opcodes which can be interpreted on all HASTE execution modules, which is similar to the intermediary bytecode used by the Java virtual machine. Database contents are stored in a proprietary format which compartmentalizes data into small segments which we refer to as 'tablets'. The practice of subdividing data is seen in large-scale scientific and commercial use of

Google's MapReduce programming model and allows data transfer to take place in small chunks so that processing may take place simultaneously and reduce the delay associated with the data transfer.

c. Experimental Environment and Results

We design multiple experiments to evaluate HASTE's performance in varied situations. Both a synthetic dataset and a real-world dataset are created for querying. Our synthetic data set consists of a one million row and an eight million row database with three integer columns and three floating point columns, filled with randomly generated values. We have supplemented the synthetic database with a real-world dataset of almost five million rows of values representing user queries of satellite imagery provided by the United States Geologic Survey (USGS) Earth Resources and Observation System (EROS). A total of four queries are created for the synthetic datasets, two for the one million row dataset and two for the eight million row dataset which measure performance of simple and complex queries, along with two experiments for the EROS datasets which results in a total of six experiments. These experiments are evaluated on six combination of execution modules that we have designed: CPU, GPU, MIC, CPU+GPU, CPU+MIC, and GPU+MIC. We observe the largest speedup over single-thread CPU code from the GPU module, with a difference in response time of up to nearly 7x when using optimal data transfer methods. The Xeon Phi exhibits its weakness in data transfer with a total execution time 7,500% slower than the 32 thread CPU module. However, we discover that the Xeon Phi's processing capabilities begin to rival those of the Tesla K20 GPU when examining time spent preforming computation and excluding memory transfer timings.

d. Major Contributions of this Thesis

During our course of research, we have created the first ever DBMS designed for parallelized computing that utilizes CPU, GPU, and MIC architectures within a single software package. HASTE is designed from the ground up to be a modular, interchangeable environment to support easily implementable parallel functionality for database processing on current and future platforms. We explore the capabilities of Intel's first entry into the small-scale hardware coprocessor market, the Xeon Phi, observing the performance of first-generation hardware as a competitor to GPUs' long standing monopoly on the market. With the DBMS framework provided by HASTE and the low-cost hardware from Nvidia and Intel, the processing of big data and large scale databases becomes available to individuals, researchers, and corporations, without the need for large capital investments or access to supercomputers.

e. Description of Remaining Chapters

The remainder of this thesis is organized as follows. Chapter 2 provides details on other related work in the field of database acceleration, focusing on generalized database acceleration, use of GPUs, and other parallelization and optimization techniques. Chapter 3 details the design of the HASTE engine, the execution modules, and the interconnections between the varied HASTE components. Chapter 4 contains the results of our experiments on varied combinations of execution modules, data sets, and SQL queries. Finally, Chapter 5 concludes this thesis with a summary of our work and discusses the interpretation of our findings from chapter 4, as well as briefly outlining our ideas for future improvements to the HASTE project, areas of further interest, and a summary of the benefits provided by our current research.

II. RELATED WORK

a. Database Acceleration

Database acceleration is a field that has been pursued for many years by researchers targeting a great number of possible solutions. Amongst the most publicized, topics focus upon: improving the underlying structure of database queries and processing techniques, utilizing CPU-based parallelism (including cluster and shared-memory machines), and recently, using GPGPUs for co-processor based parallelism (DeWitt & Gray, 1990). Early research, such as that by Schneider and DeWitt, suggest that efficiency may be achieved by decomposing complex queries and then utilizing varied techniques to combine results to form true results, however this would not benefit simple queries performed on large datasets (Schneider & DeWitt, 1990). Chiu et al. proposes a novel concept of utilizing clustering algorithms as used in machine learning which would enable faster searching in a large database environment (Chiu, Fang, Chen, Wang, & Jeris, 2001). Graefe suggests that growing database sizes will have an adverse impact on the underlying query-processing algorithms in DBMS and that proper query evaluation techniques and more prudent data models will support more efficient growth (Graefe, Query Evaluation Techniques for Large Databases, 1993). In the course of our research, we find that data transfer proves to be a sizable difficulty for certain environments, to which Graefe proposes the solution of data compression not only for storage and transit, but manipulation and querying as well (Graefe, Data Compression and Database Performance, 1991). Despite being an edge case, Li et al. demonstrates that caching yield substantial gains for very large scale database-driven web applications with repetitive access of high-use datasets (Li, et al., 2003).

b. Parallelized DBMS

Parallelism via CPU-centric techniques has a long publication history compared to GPGPU and other technologies thanks to their relative age. Rognes and Deeberg take advantage of the MMX and SSE extensions provided by consumer x86 architectures to enhance targeted algorithms for bioinformatics searches (Rognes, 2000). Bellatreche improves querying heterogeneous databases utilizing a scatter-gather methodology tailored for data warehouses (Bellatreche, Benkrid, Crolotte, Cuzzocrea, & Ghazal, 2012). A more holistic approach is suggested by Rahman, who designs a custom DBMS in order to tune resource usage and form a stronger architectural foundation for parallel systems (Rahman, 2013). Chaiken et al. employs an approach similar to the structure we utilize for HASTE query parsing by introducing a new scripting language which can be decomposed by the execution engine for faster, more efficient processing of user queries on massive data sets (Chaiken, et al., 2008). Exploitation of superscalar processor design to enable simultaneous multithreading of queries, as demonstrated by Lo et al., exemplifies the value to be gained from knowledge of hardware strengths and the developing of code to suit the execution environment (Lo, et al., 1998).

c. Hardware-Accelerated DBMS

Once GPU-based computing was shown to have the potential for sizable performance gains, there is continued research and publication with the aim to employ this technology for the improvement of databases on big data. Govindaraju et al. focuses on the techniques possible to enhance memory bandwidth when using GPUs as coprocessors in managing large databases (Govindaraju, Gray, Kumar, & Manocha, 2006). Deriving primitive operations from SQL statements and forming GPU kernels for computing database queries is explored by Wu et al. and results in performance jumps in both computation and memory transfer (Wu, Diamos, Cadambi, & Yalamanchii, 2012). The basis for our work in HASTE was provided by Bakkum and Skadron as they implemented SQLite-derived opcodes for use on CUDA platforms to obtain substantial performance gains (Bakkum & Skadron, 2010). Comingling of GPU and CPU platforms have received attention from the research community as well, with Breb et al. proposing context-switching to form a hybrid model wherein a query would be evaluated to determine which environment would provide greater performance (Breß, Schallehn, & Geist, 2013). Similarly, Milloy et al. design a methodology in which result matching takes place on CPU and indexing takes place on GPU, making use of the strengths of each (Milloy, Fanerty, & Gerber, 2012). Query length is used to determine which hardware should execute a query in a publication by Zidan et al (Zidan, Bonny, & Salama, 2011).

d. Intel MIC

To the best of our knowledge, no research has yet been published which utilized MIC architecture to recreate an entire DBMS, owing to the recency of the MIC's availability for research. However, several features of DBMS have been presented independently to lay the groundwork for future work in this field. Full-table scans of column-store databases have been implemented on Xeon Phi hardware to achieve performance gains (Willhalm, Oukid, & Miller, 2013). Sorting operations were studied on an early implementation of MIC hardware to achieve 2.2x the performance of CPU-based sorts

(Satish, et al., 2010). Additionally, the mapping stage of MapReduce has been designed to utilize the vectorization gains through MIC hardware to obtain significant performance improvement (Dean & Ghemawat, 2008).

e. Additional Acceleration Techniques

Standing apart from techniques to utilize parallelism, distributed processing is a frequent subject of note in research surrounding database performance. MapReduce, an algorithm developed by Google and notably distributed by Apache with Hadoop, is a distributed framework designed for data-intensive applications. HASTE is inspired by the MapReduce algorithm's methodology of breaking a complex problem into simpler subproblems, distributing the workload, and them collecting result data to form a complete solution. HASTE draws on this concept by enabling queries of arbitrary size to be executed on any number of execution modules and seamlessly combined to create a result set covering the entire data source.

III. ENGINE AND MODULE DESIGN

a. Introduction and Purpose

We began the HASTE project with the purpose of designing a database engine built from the ground up for rapid processing of database queries by taking advantage of modern parallelism techniques. We have identified two primary areas of influence on performance that we need to optimize: data transfer of the target tables to the processing device and parallelism in the execution model of the SQL query. Each parallelized component requires different code design to utilize its unique abilities, but the result of a query and the processing of user input remains the same, so we have implemented a modular design of the execution engine alongside a single set of engine components to prepare query for execution. Rather than modify an existing database engine to operate on GPU and MIC hardware, a complete database engine is designed from scratch for maximum flexibility and compatibility. HASTE is compartmentalized into several modules that build upon one another to transform a user's raw SQL query into the appropriate result set.

b. Component Overview

HASTE consists of a set of support components responsible for query parsing and execution, unified by a main thread residing on the host device which holds the database files locally and has access to the parallel hardware (in our case, the CPU and MIC are attached to the host via the PCI Express bus). **Error! Not a valid bookmark self-reference.** provides an overview of this architecture. A single database file system and instruction set form the framework of the HASTE system. These components are designed to enable hardware agnosticism and support the addition of new processing modules for future work on other parallelized systems by creating an intermediary phase in the query execution flow which can be utilized by differing execution models. Behind the HASTE support components resides a modular set of processing components which have been designed to operate independently or cooperatively. We have developed modules that execute on the CPU, GPU, and Xeon Phi hardware, with the opportunity for additional modules to be developed for other parallel systems.

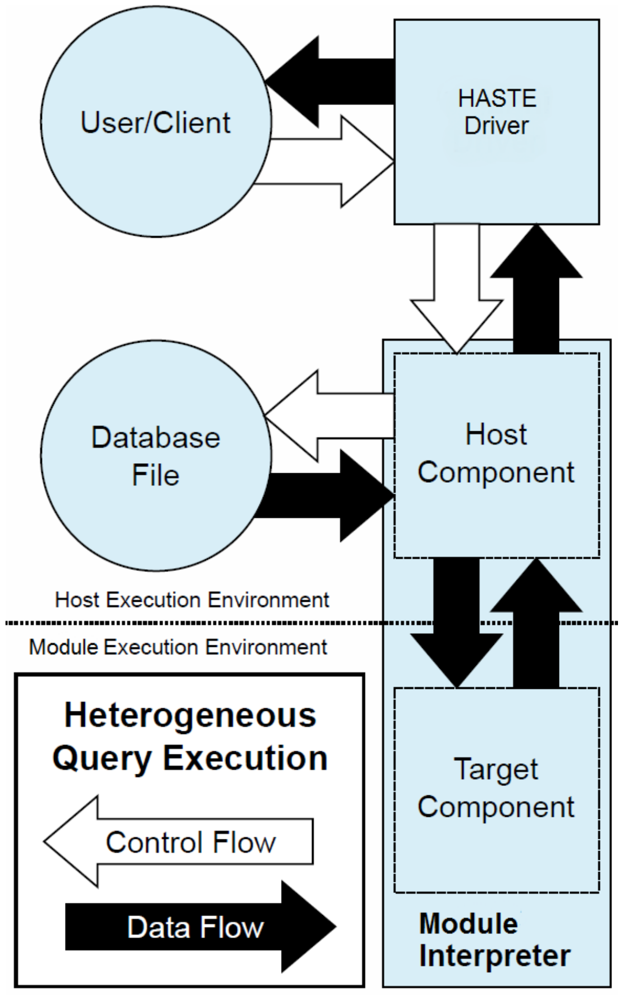


Figure 1 - Overview of the HASTE architecture

The main HASTE thread runs on a host machine and receives a SQL query from user input, parsing the query into a list of operations for use in one or more execution modules.

Once compiled, the list of SQL operations is passed to a processing module along with a segment of the database, and the module interprets the SQL operations to execute the user's query, returning valid results back to the main HASTE thread.

c. Database

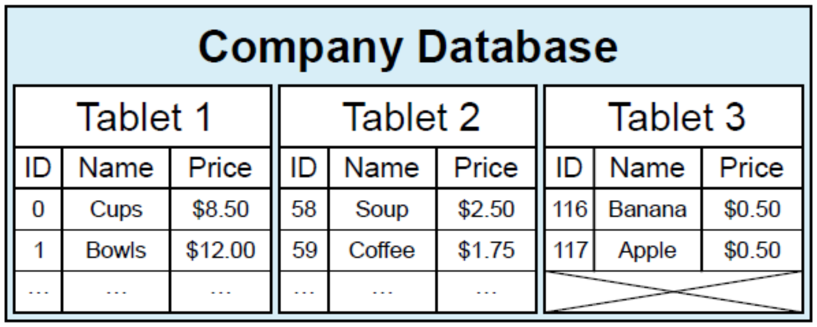
i. Purpose

We designed a simple flat file database system to serve as the storage mechanism for the HASTE system. Enhancements in storage efficiency and compression are not objectives for this project, so a flat file system was the most logical method for supporting offline database storage. Due to the modular nature of execution modules allowing for differing hardware possibilities, the simplest database is optimal over more complex options which would be found in modern database systems.

ii. File Structure

The HASTE database file is a single file residing on the hard disk of the host machine. Within the database are size-delimited blocks of data which we refer to as tablets, designed to facilitate transfer of data to execution modules. Each tablet is a fixed size, specified at the time that the database is created, with additional tablets automatically instantiated and appended to the end of the database file as new rows are added. Three distinct areas are contained within each tablet: metadata, primary keys, and the database contents themselves.

Figure 2 below provides an example of how tablets constitute the database structure.



Company Database

Tablet 1			Tablet 2			Tablet 3		
ID	Name	Price	ID	Name	Price	ID	Name	Price
0	Cups	$8.50	58	Soup	$2.50	116	Banana	$0.50
1	Bowls	$12.00	59	Coffee	$1.75	117	Apple	$0.50
…	…	…	…	…	…			

Figure 2 - Example file structure of a database in HASTE

Tablet metadata houses a description of the overall database structure, information on the file structure on the disk, description of the database contents and datatypes, and pointers to the preceding and subsequent tablets in the file structure (when applicable). Metadata is a fixed-width portion of the tablet and comprises the first set of bytes within the tablet. The primary key area within the tablet contains the primary keys associated with the values contained within the tablet, with the size of the segment varying based upon the number of rows contained and the type of the primary key. The database contents are stored in column-major format, which is selected to improve search speed within a query by ensuring that values within a single column will be loaded into memory alongside one another. Typically, a column-major database will contain a primary key associated with each column, however our implementation contains the primary key in a separate area of each data segment, eliminating the need for this data overhead. The size of the data segment occupies the remainder of the tablet and contains a fixed-size and variable-size area. Fixed-size elements such as integers and floats are stored within the former segment, while the latter area is intended for strings, arrays, and blobs with varying sizes across rows. The number of rows contained within the data segment is a function of the overall tablet size, the number of columns in the database, and the number of fixed- and variable-sized elements stored within.

d. Instruction set

i. Purpose

In order to service the modular design ideals of HASTE, we needed to design a system of executing a SQL query on any of a number of hardware components. An opcode system is selected as the optimal way of achieving this functionality. Similar to the opcode system utilized in hardware design, we have implemented a set of operations that will act as an intermediary between SQL statements and the underlying code needed to execute them. These instructions will have varying implementation methodologies which can be defined within each of the execution modules.

ii. Design

We have designed an instruction set following the methodology used by the Virginian GPU accelerated database, which in turn is a derivation of the SQLite instruction set. The HASTE implementation of this instruction set consists of twenty-six operations which are required for basic database operations, arithmetic, control flow, and data manipulation. Currently, only the SELECT SQL operation is supported, but the modularity of the HASTE code allows for the addition of additional opcodes to expand the functionality of the SQL engine. Each opcode has room for up to four arguments, supporting passing values including locations, datatypes, operands, and other necessary parameters.

iii. Use

The HASTE bytecode is designed to achieve modularity across differing execution modules by parameterizing instructions and implementing operations as abstract functions which may be redefined as required at a deeper level. When transforming a SQL statement into HASTE code, a PARALLEL instruction and CONVERGE instruction surround the code block which contains the core operations for executing the query. Upon reaching the PARALLEL instruction, the host HASTE thread invokes the specified execution module, which is responsible for the series of opcodes until the CONVERGE instruction is reached. The execution module will contain definitions for each opcode which are designed in a manner that will leverage the hardware used in the respective module. By allowing each module to contain its own code for executing an opcode, the instructions can be optimized for multiple differing parallel environments while maintaining the same neutral intermediary step of translation from SQL statements to the HASTE instruction set. In addition, the process enables the use of multiple execution modules simultaneously for the same SQL query.

e. Query compiler

i. Purpose

HASTE’s query compiler was designed to enable the hardware-agnostic translation of SQL statements into the intermediary opcodes described in chapter 3. A SQL statement is received from the user upon invocation of the HASTE main thread. Before the query is executed it is first translated into opcodes by the query compiler. The translated opcodes are then passed on to the execution module to generate the results. Conversion of the

SQL statement to opcodes is a two-step process. First, an abstract syntax tree is generated by parsing keywords and parameters to create a logical path for the query to follow. Then, the AST is traversed and each node is converted into an instruction keyword and placed in a queue which represents the requisite steps to obtain the result of the SQL query from the execution modules.

ii. AST Creation

Upon submission of a SQL query, the constituent operators and operands are separated and run through a translation matrix. Internally, the pairing of Flex and Bison provide the background functions for building an abstract syntax tree containing the opcodes to execute the query. Flex utilizes a provided Lex file to generate a lexical analyzer which tokenizes the SQL query and provides an input for Bison. The tokens from Flex are matched with the grammar provided to Bison to generate AST nodes. Each AST node is representative of one of the twenty-six opcodes defined in the HASTE grammar. Once the AST is finalized, the final step of the pre-execution process is to traverse the AST and generate a linked list that contains the series of operations required to obtain a result from the database.

iii. Instruction Set Creation

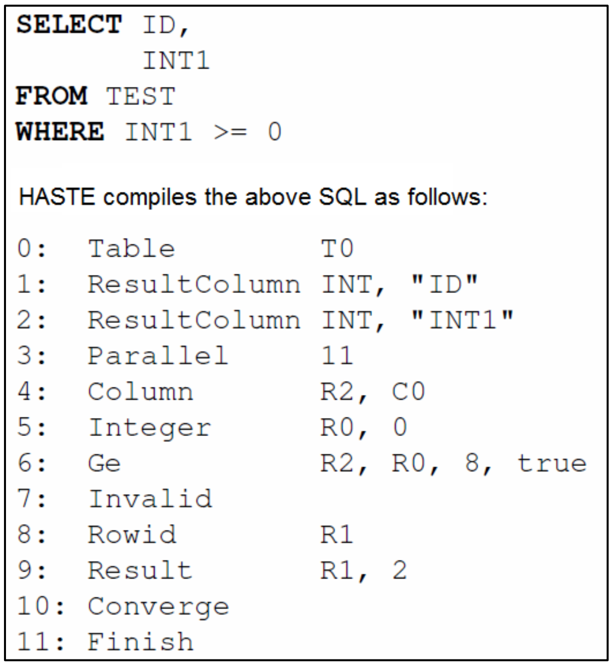
The HASTE instruction set's 26 instructions provide the minimum actions needed to support the basic SELECT operation. A translated query will contain a central chapter of arithmetic and logical steps to derive a result from input data and is surrounded by a generic set of wrapper instructions to prepare the query and data for execution. Wrapper instructions will define the table and columns to be operated upon and contain the trigger instructions to invoke and terminate an execution module. As an example, given a table TEST with one integer column INT1 and a primary key of ID, consider the following query:

```
SELECT ID, INT1

FROM TEST

WHERE INT1 >=0
```

this query will be translated as below in Figure 3.



```
SELECT ID,
       INT1
FROM TEST
WHERE INT1 >= 0
```

HASTE compiles the above SQL as follows:

```
0:  Table        T0
1:  ResultColumn INT, "ID"
2:  ResultColumn INT, "INT1"
3:  Parallel     11
4:  Column       R2, C0
5:  Integer      R0, 0
6:  Ge           R2, R0, 8, true
7:  Invalid
8:  Rowid        R1
9:  Result       R1, 2
10: Converge
11: Finish
```

Figure 3 - Translation of SQL query into Opcodes

In the example in Figure 3, the first instruction sets up the table TEST, residing in this particular database in table slot 0. The next two set up the result columns, giving them their appropriate names and assigning to each the type int. Between the Parallel/Converge instructions (the former of which specifies that the interpreter should jump to instruction 11 on completion of the latter), each instruction is performed on each row of input.

The column INT1 is processed by loading the value in column 0 (INT1) into register 2; loading a literal zero into register 0; and then comparing the values. If the comparison is successful (i.e., INT1 >= 0), then the primary key is processed. Otherwise, the row is invalidated, and execution falls through instructions 8 and 9, effectively treating them as no-ops.

To process the primary key, its value is loaded by the Rowid instruction into register 1. No further computation is necessary at this point, and so the results are emitted. This is done by the Result instruction, which fills the columns defined earlier by ResultColumn with the values in the two registers starting with register 1, i.e. ID and INT1. Thus, if INT1 >= 0 for some particular row, ID and INT1 are emitted as values in the result tablet. Otherwise, evaluation is short-circuited using data flow (as opposed to using such a control flow construct as, say, Jump 10) and no results are emitted for that row.

f. Execution Modules

i. Overview

The execution modules enable the modularity and heterogeneous execution of SQL queries within the HASTE system. An instruction interpreter is written for each hardware and software component to be leveraged for parallel execution within HASTE. Each interpreter supports the opcodes representing a SQL statement by providing the ability to utilize the unique utilities available to the supported hardware or software. Three modules have been written to demonstrate the functionality of the HASTE system: a CPU module using pthreads, a GPU module using CUDA, and a Xeon Phi module using OpenMP.

An instruction interpreter will contain two components: a host module and a target module. The host module is generally a small segment of code responsible for the invocation of the target module, handle data transfer from the host to the target device, and to provide an endpoint for data transfer back to the host upon completion of execution.

ii. Data Transfer

Enabling execution modules to access the HASTE database is a two-step process. First, the database must be moved from the hard disk of the host system into the host's main memory. After loading into memory, data can be transferred to the resident memory of the device which will execute the query. Transfer from disk to main memory takes place before the execution module is invoked and query execution begins, at which time the tablets that contain the target database will be loaded in their entirety into main memory of the host. Once the query has completed execution on the target device, result values will be transferred back from the execution module to the host device and then written back to disk. Write-after-write data races are avoided as a result of the flagging of valid entries during execution. As input data is evaluated, values which pass the query criteria are marked as valid and are not written until execution of all rows in the tablet are complete, at which point they are linearly inserted into the result tablet.

iii. CPU Module

The CPU execution module is the simplest design of the three modules we have implemented in HASTE. Parallelism is achieved within the CPU module via POSIX threads surrounding the primary execution segment of the module. After the PARALLEL instruction is encountered, the pthread fork is invoked upon the action method as needed to occupy all available virtual cores on the CPU. Within the action method is a lookup function that maps instructions within the linked list of opcodes to definition blocks that preform the requisite action. A copy of the instruction list is created within each thread and traversed independently of the other threads.

Accessing the target database does not have additional overhead as we will observe in the other execution modules, as the CPU directly accesses main memory of the host machine where the database resides. When the CPU module begins execution, the database tablets are immediately available for processing. Data within the tablets are accessed by the CPU threads in contiguous blocks that evenly distribute the available rows amongst the total count of threads in use. When a thread discovers a value that matches the query criteria, a VALID flag is set within the tablet. Once the query has completed within all threads and the threads have joined, the main thread iterates through the data tablet to find valid flags and copies the appropriate values to a result tablet, which is then written to disk and the CONVERGE instruction is encountered, returning control back to the HASTE thread on the host system.

iv. GPU Module

The GPU execution module utilizes the large amount of execution units available within the GPU to process data at a large scale. Similar to the CPU module, each CUDA core is provided with an individual copy of the instruction list. Once the PARALLEL instruction is encountered and the GPU module invoked, the CUDA host thread launches the HASTE GPU kernel and the threads are allowed to diverge as they process the database.

As the GPU lacks direct access to main memory, we must rely on efficient data transfer techniques to reduce the large overhead experienced when moving data on and off the device. Transferring data to the GPU is accomplished via the use of pinned, mapped memory. Host memory containing the target database is pinned in place to ensure that it cannot be swapped to disk and will not change its virtual address. The pinned segment of memory is mapped to the GPU so that virtual addresses on the host can be directly addressed by the CUDA kernel and transferred to the device as needed alongside execution of the SQL statement.

Mapped memory is also utilized to write data back to the host device in full duplex alongside reads, thus allowing up to double the data throughput. Once a valid result is identified, it is immediately transferred back to the host's main memory and placed within a result tablet. When all data is processed, the CUDA main thread returns control to the HASTE host thread and the filled result tablet is written to disk. By using pinned mapped memory we greatly reduce the need to cease processing data in order to transfer to and from the GPU.

v. MIC Module

Intel's Xeon Phi does not support software-specialized parallel operations like what is available with CUDA. Parallelism requires the use of OpenMP pragmas to offload C++ code to the MIC (MPI is also a supported technology). #pragma offload statements surround an execution function which processes the linked list of opcodes. Within the offload pragma, a #pragma omp parallel block executes the SQL statement independently across the available Xeon PHI cores which independently process the target database.

Data transfer to the Xeon Phi takes place within the #pragma offload statement. Data tablets are moved to the memory of the MIC en masse at the point that the offload pragma is encountered, up to one half the available memory on the device. The remaining half of the memory is used to house the result tablet buffer as it is filled during execution. While the processing function is executing on the MIC cores, when a valid row is encountered it is copied to the next available location within the result tablet. Once the query has executed on all rows currently located within MIC memory, the execution function ends and the result tablet is transferred back to the host memory. If any additional tablets remain to be processed, the process will repeat and new tablets will be sent to the MIC to have the query run upon them. When no more tablets remain unprocessed, control is returned to the HASTE host thread and the result tablet are merged and written to disk.

IV. RESULTS

a. Overview

We have designed a suite of experiments for the HASTE system that examines performance of execution units and multiple datasets and queries. Two datasets are used for experimenting – one large synthetic dataset comprised of random values and one smaller dataset created from real-world data representing user data queries for satellite imagery. Each dataset has two queries preformed upon it to evaluate the performance of running both simple and complex operations. These queries are executed on several hardware configurations: CPU with variable cores, GPU with pinned/mapped and standard memory allocation, MIC with variable thread use, and tandem use of CPU+MIC, CPU+GPU, and MIC+GPU.

b. Data Setup

i. Synthetic Data

Our synthetic dataset allowed for experimenting of large block databases containing a variety of data types. A generation function was written to allow the creation of HASTE formatted databases of arbitrary size and content. The synthetic database contains six data columns composed of three integer and three floating point columns, along with a seventh integer column for storing the sequential ID of the column, which is also utilized as the primary key. Contents of each cell are generated at random using a Gaussian distribution for values between -1,000,000 and 1,000,000, with two decimal places allowed for the floating point values. Two different sized databases are created, containing 1,000,000 and 8,000,000 records and resulting in on-disk sizes of 138 MB and

1.13 GB.

Two queries are employed to test the performance of both simple and complex SQL queries. The simple query operates upon only a single data column, along with the index, intended to measure minimal processing time for one comparison operation and data transfer. This query is input as:

```
SELECT ID, INT1
FROM test
WHERE INT1 >= 0;
```

This will return approximately one half of the input database as the result (500,000 and 4,000,000 rows). A more complex query with boolean logic and multiple evaluations evaluates the performance of six comparison operations and five boolean operations. Our complex query is entered as:

```
SELECT ID, INT1, INT2, INT3, FLOAT1, FLOAT2, FLOAT3
FROM test
WHERE ((INT1 > 0 AND FLOAT1 < 0.0) OR (INT2 < 0 AND FLOAT2 >
0.0) OR (INT3 < -500000 OR FLOAT3 > 500000.0));
```

Which results in a result set of approximately 56.25% of the input size, or 562,500 and 4,500,000 rows for 1 million and 8 million source rows.

ii. Real-World Data

For more realistic performance measurements, we have employed a real-world dataset. The data was supplied by the United States Geologic Survey Earth Resources Observation and Sciences Center and represents several years of logs they provide.

EROS provides satellite imagery of the Earth's landmasses to users worldwide for research and private purposes. Requests are stored with the requested image (in X/Y coordinate format with a time value), ID of the requesting user, and a timestamp for the request, along with an integer ID acting as the primary key. All columns store integer values. USGS supplied 4,882,305 rows of usable requests spanning a 3 year period from 2008-2011. We processed and modified the source data to convert it into supported data formats. The original data received from EROS was 1,670 KB, and after the required modifications and trimming, the database was reduced to 1,536 KB. Data modifications are performed with an external utility and required negligible overhead time outside of the initial time to design a new format.

As with the synthetic database, two queries are designed to test performance under multiple use cases both simple and complex. A simple query was crafted to select all images requested by a single user, input as

```
SELECT ID, XCORD, YCORD, CAPTIME, UID, TIMESTAMP
FROM eros
WHERE UID = 75032;
```

Which results in a small result set for the user whose ID is 75032, yielding seven records (.00014% of input rows). A larger query was also designed, which searches for all requests submitted in a one year period from 12:00 AM on January 1, 2009 to 11:59 PM December 31, 2009. This query is input as:

```
SELECT ID, XCORD, YCORD, CAPTIME, UID, TIMESTAMP
FROM eros
WHERE TIMESTAMP >= 01012009000000 AND TIMESTAMP <=
123109115959;
```

which results in 1,402,198 result rows (28.72% of input values).

c. Execution environment

We have assembled a unified environment for executing the experiments of the HASTE system. The database files are located within a 250 GB SSD connected to the host through a SATA III connection, with 1.1 GB uncached sequential reads transferring at an average of 474 MB/s. The HASTE testbed has 64 GB of synchronous main memory in a dual-channel 8x 8GB configuration at 1600 MHz. Two eight-core Intel Xeon ES-2650 CPUs with hyper-threading power the host system, providing a total of 32 virtual cores. Each physical CPU contains 20 MB of L3 cache, 2 MB of L2 cache, and 512 KB of L1 cache.

Our GPU module executes on an Nvidia Tesla K20m GPGPU with 2,496 706 MHz cuda cores and 5 GB of shared GDDR5 memory, connected to the host over a PCI Express 2.0 x16 interface. The MIC module executes on an Intel Xeon Phi 5110p. The Xeon Phi architecture provides 60 physical cores (240 logical) operating at 1.053 GHz, 8 GB of onboard GDDR5 memory, and connects via PCI Express 2.0 x16.

Source code is compiled for the CPU execution module and the HASTE host program using GCC 4.8, CUDA 5.0 is used to compile the GPU portion of HASTE, and MIC components are compiled with Intel C++ Compiler 13.1.3.

d. Experimental Results

When evaluating the performance of HASTE, it becomes apparent that the total execution time was not an appropriate metric for comparison. The total execution times are overwhelmingly high for the Xeon Phi in all experiments. In Figure 4 we can observe that even the best Xeon Phi execution time is 15% slower than a single threaded CPU experiment. In contrast to the Xeon Phi, the K20 GPU presents response times are comparable to the multi-threaded CPU execution and is the fastest device to process the simple query. For all results in this chapter, response times are measured from the point that the HASTE main thread invokes one or more execution engines to the point where control is returned back to the main thread. No time spent preforming program startup, syntax parsing, database creation, or presentation of results to the user are included in any measurement herein.

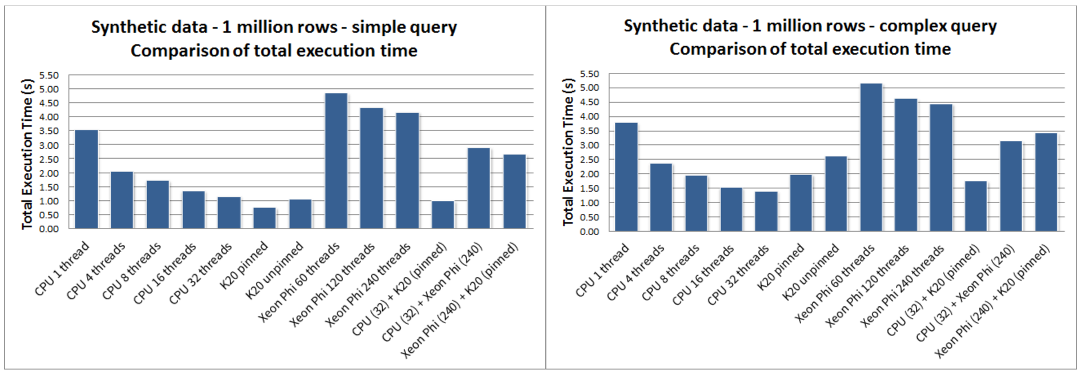


Figure 4 - Comparison of execution times for simple and complex queries on 1 million rows of data

If we compare the two graphs in Figure 4 we see that the addition of more data to the table has opposite effects on the total execution time of the GPU and MIC architectures against the baseline CPU result. In Figure 5, we see that both pinned and unpinned GPU response times are equal to or slightly faster than the 32 thread CPU results (using all cores plus hyperthreading). In contrast, the move from one million to eight million rows of data has a detrimental impact on the Xeon Phi response times, with results taking up to twice as long over a single thread.

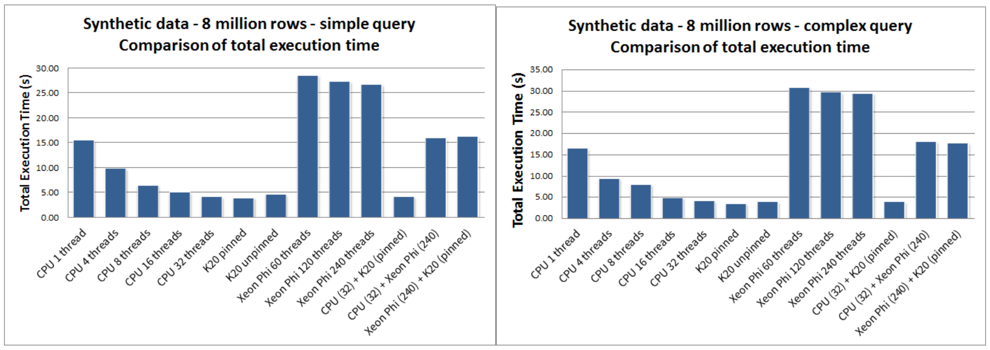


Figure 5 - Comparison of execution times for simple and complex queries on 8 million rows of data

The total execution time proves to be misleading when comparing the processing power of the CPU, GPU, and MIC components of HASTE. By decomposing the total execution time into the constituent metrics of computation time and the time for memory transfer + overhead, we see a much clearer picture of the strengths and weaknesses of the respective architectures. Figures 6-9 below illustrate the key differences between our experiments, namely the ability to transfer source data and results, as well as the speed in which computation is performed.

In the below figures, compute time is defined as the amount of time (as reported by the system) that processor cycles are spent executing the code on each device. Memory + overhead and compute time is the remainder after subtracting compute time from the total measured execution time, which represents the time it takes for the devices to become ready, copy and initialize code, and to transfer data to and from main memory (and device memory, in the cases of the Xeon Phi and K20). As we would expect, the CPU experiments has the lowest memory transfer and overhead time, as it has the least amount of overhead and direct access to the system main memory.

CUDA architecture allows for the use of ‘pinned’ memory, which causes system memory to become unable to be swapped to disk and ensures that the virtual addresses for memory within the pinned blocks will never change. This results in the ability to pass pointers from the host device to the GPU so that the GPU has direct access to system memory and can transfer as much as needed on an ad-hoc basis. The GPU pinned memory experiments showed an approximate 30% reduction in wait time for memory transfer over unpinned memory as a result of the ability to access system memory via pointer and transfer data to the device asynchronously.

The Xeon Phi experiments reveal that up to 98% of the total execution time is spent on handling the memory transfers and overhead. If we choose to examine only computation time, our findings give much more weight to the potential of co-processor accelerated performance gains over simple CPU execution. For 8 million rows of data, the 240 thread Xeon Phi experiment preforms 1.7x as quickly as the 32 thread CPU experiment, and the K20 completes its query 5.7x as fast as the same CPU experiment.

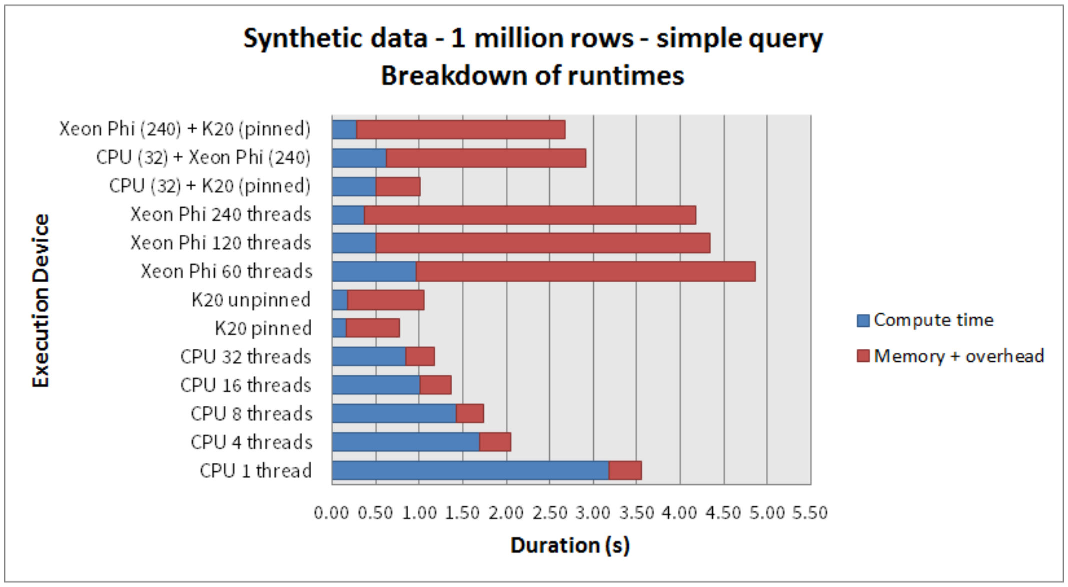


Figure 6 - Compute and memory + overhead times for 1 million row simple query

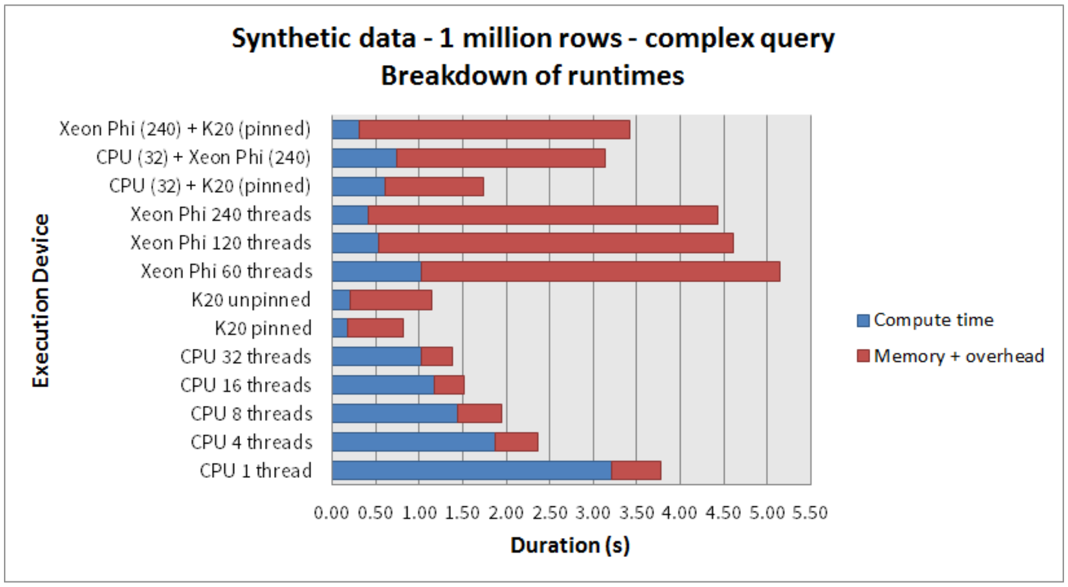


Figure 7 - Compute and memory + overhead times for 1 million row complex query

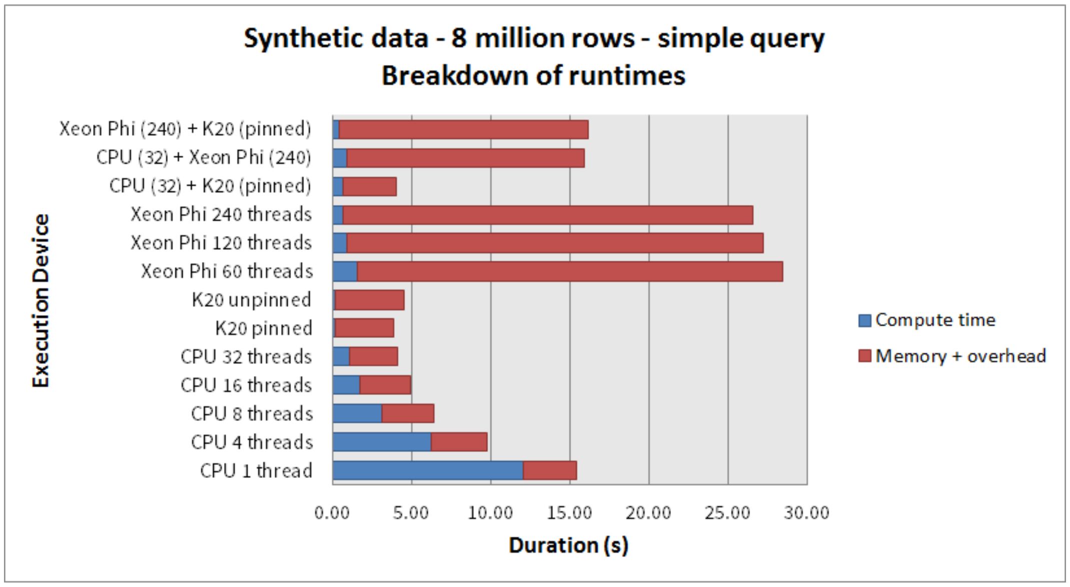


Figure 8 - Compute and memory + overhead times for 8 million row simple query

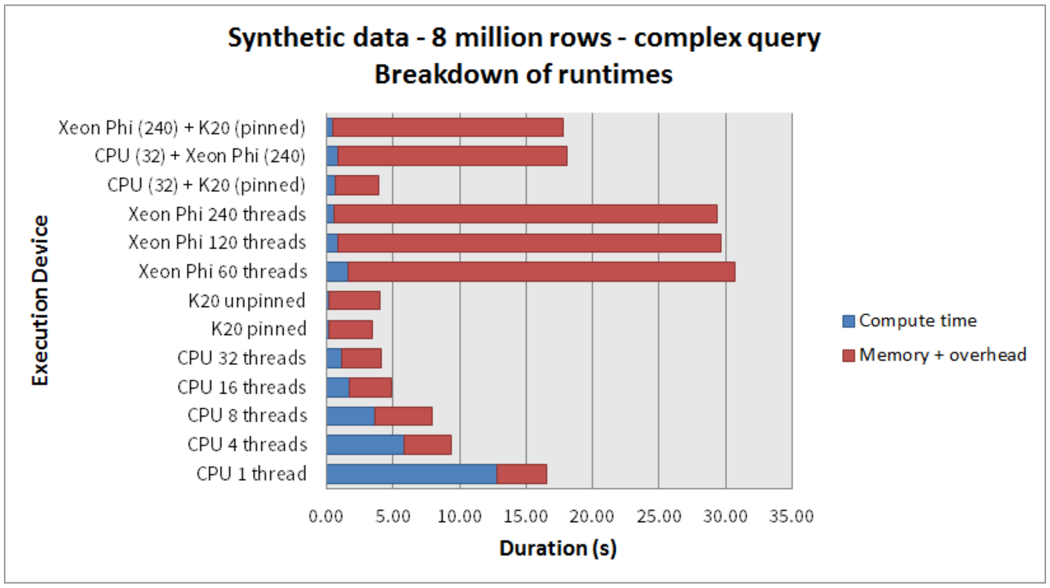


Figure 9 - Compute and memory + overhead times for 8 million row simple query

The best performing experiments from each HASTE module are summarized below in figures 10 and 11, divided into compute times and memory + overhead times. We can see in the Figure 10 the compute times have a much smoother curve and are tightly aligned with the other points in each series. The GPU curve has almost no variance across all experiments, emphasizing its mature feature set and tendency towards numerical processing. Of all the experiments, the Xeon Phi does show a much larger gap in its compute times, attributable to its nascence and more generalized set of applications. This gap is emphasized even further in the figure 11 wherein the Xeon Phi exhibits a massive jump to almost 30 seconds in memory transfer + overhead time while the CPU and K20 take on the order of 3.5 seconds. In our development of the Xeon Phi module for HASTE, we noted a lack of refinement around memory transfer capabilities and limitations on the way in which data may be moved into and out of the Xeon Phi. Basically, every call to offload computation to the Xeon Phi must complete the transfer of all required data to the device memory before any code can be executed and similarly all result data must be transferred back to the host memory before control can be returned to the CPU. Compounding this issue, the Xeon Phi does not offer the same memory transfer functionality as supported by the K20.

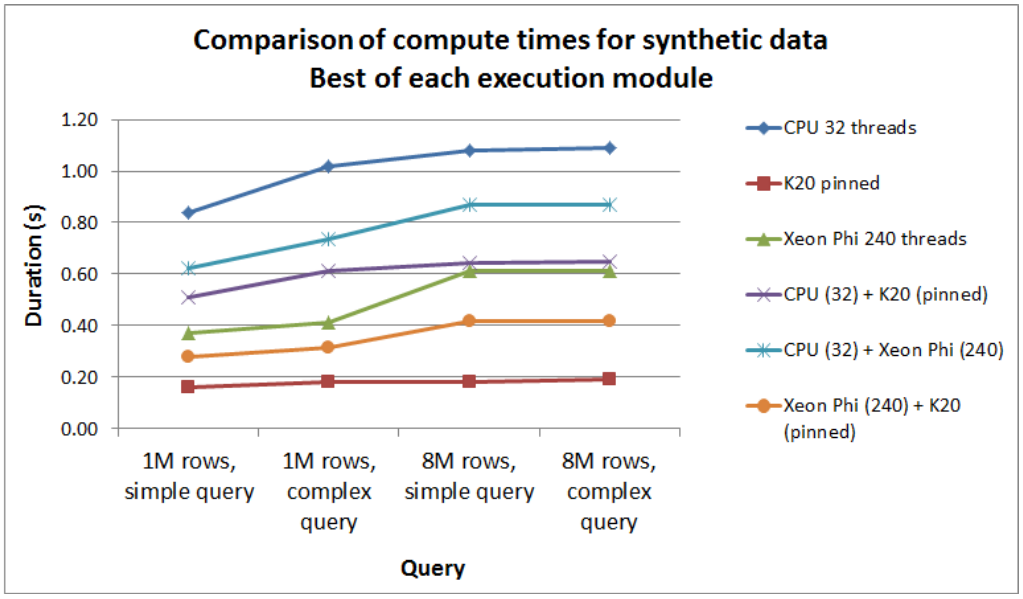


Figure 10 - Comparison of compute times for the best executions of each HASTE module

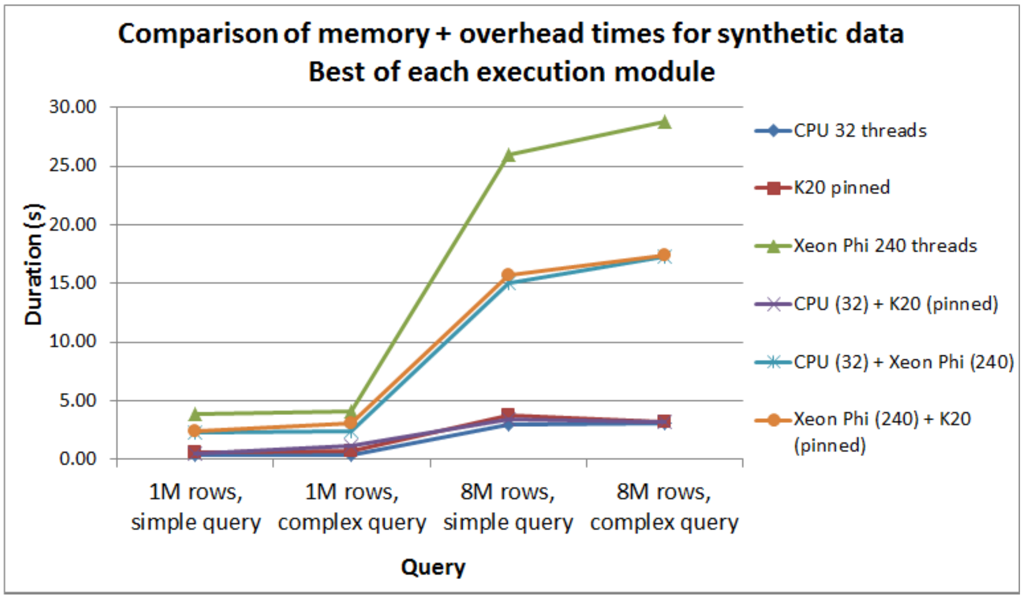


Figure 11 - Comparison of memory + overhead times for the best executions of each HASTE module

Results of our real-world experiments on the EROS data validate the accuracy of our synthetic benchmarks and exhibit the same strengths and weakness as the previous experiments have indicated. When examining pure execution time as in figure 12 we once again see that the Xeon Phi appears to be far inferior to all other HASTE modules and the GPU module provides the fastest execution time of all EROS experiments. However, when decomposed in figure 13 we see the same crippling memory transfer bottleneck that reduces the Xeon Phi's effectiveness and masks the potential gains of its heightened response time. Although the K20 does have the fastest compute time of up to 7.25x the fastest CPU time, the Xeon Phi still provides a 3.6x improvement over CPU compute times.

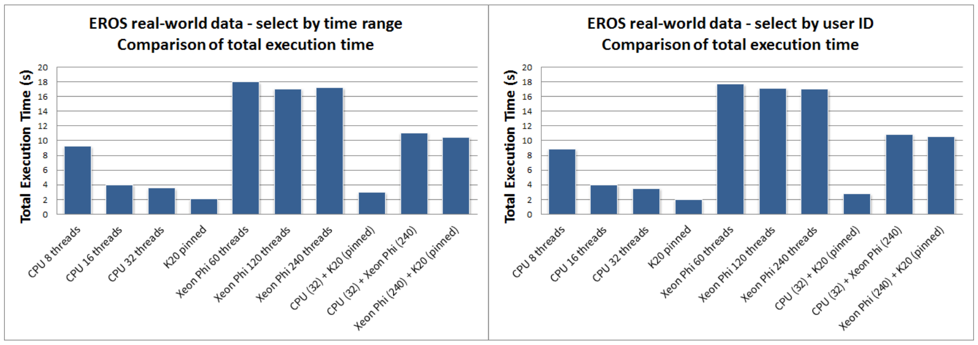
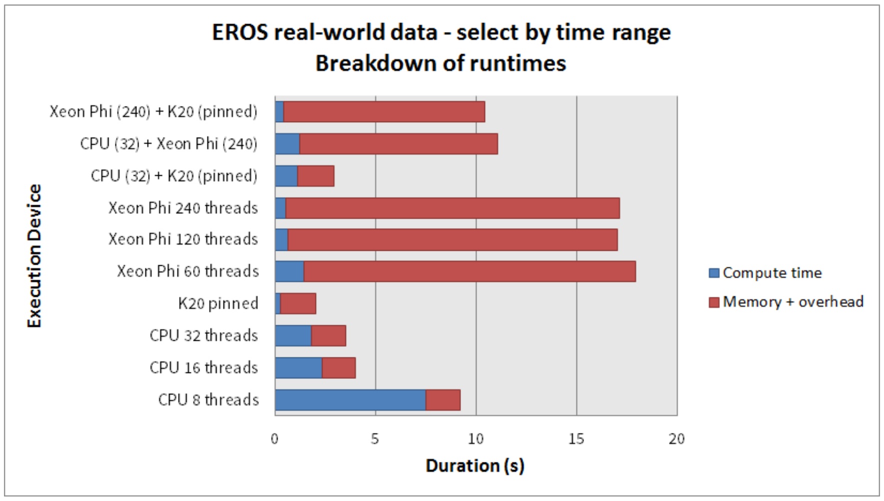


Figure 12 - Total execution time for each query on real-world data from EROS



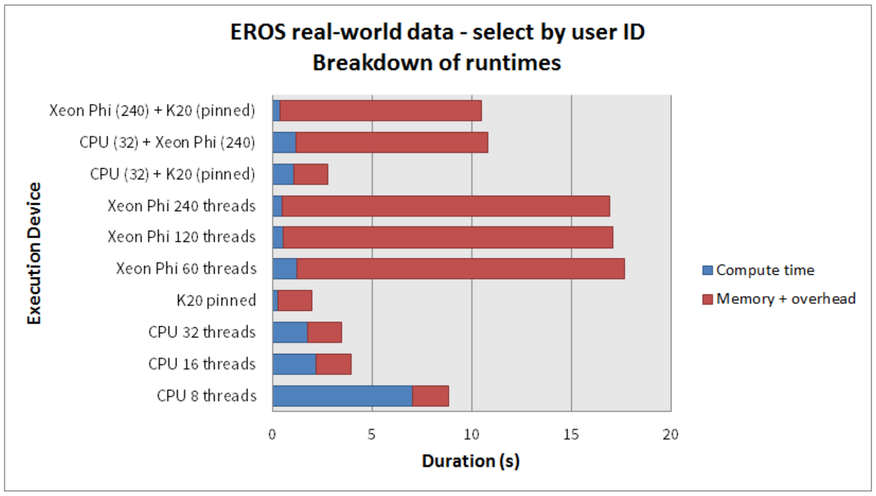


Figure 13 - Breakdown of compute and memory + overhead times for experiments on real-world data from EROS

V. CONCLUSION AND FUTURE WORK

Differences in hardware capabilities mean that a 1:1 comparison would be biased towards devices which lack refinement in certain areas. Two areas emerged which must be measured separately to understand device performance: computation capabilities and memory transfer / overhead. The Intel Xeon Phi's computation capabilities provide sizable gains of up to 3.6x a 32-thread CPU experiment, but are ultimately hindered by its reliance on slow memory transfers both to and from the device. Nvidia's Tesla K20 is shown to realize the most substantial gains in both total execution time and computation time over the fastest CPU-based experiment. At its best, the K20 outperformed the 32 thread CPU experiment by 725% in computation time and 178% in total execution time.

Each suite of experiments in the previous chapter includes a series of combinations of CPU+GPU, CPU+Xeon Phi, and MIC+GPU to demonstrate HASTE's ability to diversify execution across multiple execution modules. As the single-module experiments revealed, there are clear distinctions in performance between the various modules and thus the combined experiments do not outperform the best single-module experiment. However, we do note that there is virtually zero overhead in the HASTE engine from diversifying execution across two or more modules. This is a core component of the HASTE design and provides the foundation to scale to much larger execution environments with access to a wider array of components which may be utilized to further improve the performance on a larger dataset.

We have seen that both the K20 and Xeon Phi can provide a considerable performance improvement over typical CPU based execution. With future hardware revisions and additional refinement of the Xeon Phi hardware and software, many of the limitations exposed over the course of this research may be resolved. The goal of the HASTE project is to accelerate database performance through parallel programming in order to provide more responsive queries on large datasets. Our research has shown that the K20 is capable of scaling performance as the size of the dataset grows and enables it to be a viable prospect for high performance database acceleration. The Xeon Phi, while superior to CPU in query responsiveness, has displayed its inability to scale with large datasets due to its large memory transfer bottleneck and, as such, is not yet ready to be utilized as a reasonable means to achieve high performance database processing.

HASTE is a framework which is designed to be extensible and support the development of new execution modules for additional hardware and software platforms not yet experimented in the scope of this research. Currently there are a great number of parallel technologies that we have not been able to experiment HASTE on, such as cluster computing environments using MPI, multiprocessor platforms using OpenMP, reconfigurable FPGAs, custom designed ASICs, and many more, including technologies still in research or not yet available for experimenting. Any hardware that can communicate with the HASTE core module running on a host system may be utilized for database processing through the HASTE architecture. In order to become a more viable DBMS, HASTE must be provided with a more complete query parser and opcodes to support the full functionality of SQL statements in use today. Additionally, refinements could be made to the HASTE system wherein more complex queries could be analyzed to determine which of the available execution environments would be suitable for a given component of a user’s search, enabling the heterogeneous nature of HASTE to support choosing ideal processing methodologies.

In conclusion, the ability to gain more performance on database queries over ever growing datasets is most certainly a reality that can be achieved through hardware accelerated parallelism. There are many hardware platforms in existence that may be employed to realize such gains, and this project has only utilized three such devices. With the HASTE project, newer execution modules may be easily created for many more parallel computing platforms and work in unison to bring us closer to the processing power needed to support the vast growth of data we see today.

REFERENCES

Bakkum, P., & Skadron, K. (2010). Accelerating SQL database operations on a GPU with CUDA. *3rd Workshop on General-Purpose Computation on Graphics Processing Units* (pp. 94-103). ACM.

Bellatreche, L., Benkrid, S., Crolotte, A., Cuzzocrea, A., & Ghazal, A. (2012). The F&A Methodology and Its Experimental Validation on a Real-Life Parallel Processing Database System. *Sixth International Conference on Complex, Intelligent and Software Intensive Systems* (pp. 114-121). Palermo: IEEE.

Breß, S., Schallehn, E., & Geist, I. (2013). Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems. *New Trends in Databases and Information Systems*, 27-35.

Bureau of Labor Statistics, US Department of Labor. (2014). *Database Administrators.* Retrieved from Occupational Outlook Handbook, 2014-15 edition: http://www.bls.gov/ooh/computer-and-information-technology/database-administrators.htm

Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, B., Weaver, S., & Zhou, J. (2008). SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB Endowment*, 1265-1276.

Chiu, T., Fang, D., Chen, J., Wang, Y., & Jeris, C. (2001). A Robust and Scalable Clustering Algorithm for Mixed Type Attributes in Large Database Environment. *Seventh ACM SIGKDD International Conference on Knowledge Discovering and Data Mining* (pp. 263-268). New York: ACM.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM, 51*(1), 107-113.

DeWitt, D. J., & Gray, J. (1990). Parallel database systems: the future of database processing or a passing fad? *ACM SIGMOD Record - Directions for future database research & development, 19*(4), 104-112.

Gartner. (2013). *Magic Quadrant for Operational Database Management Systems.* Retrieved from Gartner: http://www.gartner.com/technology/reprints.do?id=1-1M9YEHW&ct=131028&st=sb

Govindaraju, N., Gray, J., Kumar, R., & Manocha, D. (2006). GPUTeraSort: high performance graphics co-processor sorting for large database management. *ACM SIGMOD International Conference on Management of Data* (pp. 325-336). ACM.

Graefe, G. (1991). Data Compression and Database Performance. *Symposium on Applied Computing* (pp. 22-27). Kansas City: IEEE.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys, 25*(2), 73-169.

Hilbert, M., & Lopez, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information. *Science, 332*(6025), 60-65.

Li, W.-S., Po, O., Hsiung, W.-P., Candan, K. S., Agrawal, D., Akca, Y., & Taniguchi, K. (2003). CachePortal II: Acceleration of Very Large Scalar Data Center-Hosted Database-Driven Web Applications. *29th International Conference on Very Large Data Bases* (pp. 1109-1112). Berlin: VLDB Endowment.

Lo, J., Barroso, L., Eggers, S., Gharachorloo, K., Levy, H., & Parekh, S. (1998). An analysis of database workload performance on simultaneous multithreaded processors. *25th annual international symposium on Computer architecture* (pp. 39-50). IEEE.

Milloy, J., Fanerty, B., & Gerber, S. (2012). Tempest: GPU-CPU Computing for High-Throughput Database Spectral Matching. *Journal of Proteome Research*, 3581-3591.

Rahman, N. (2013). SQL optimization in a parallel processing database system. *26th Annual IEEE Canadian Conference on Electrical and Computer Engineering* (pp. 1-5). Regina: IEEE.

Rognes, T. (2000). Six-fold Speed-up of Smith-Waterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. *Bioinformatics*, 699-706.

Satish, N., Kim, C., Chhugani, J., Nguyen, A., Lee, V., Kim, D., & Dubey, P. (2010). *Fast Sort on CPUs, GPUs and Intel MIC Architectures.* Intel Labs.

Schneider, D. A., & DeWitt, D. J. (1990). Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *16th VLDB Conference* (pp. 469-480). Brisbane: VLDB Endowment.

Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sarma, J., . . . Liu, H. (2010). Data warehousing and analytics infrastructure at Facebook. *ACM SIGMOD International Conference on Management of data* (pp. 1013-1020). ACM.

Willhalm, T., Oukid, I., & Miller, I. (2013). Vectorizing Database Column Scans with Complex Predicates. *Accelerating Data Management Systems Using Modern Processor and Storage Architectures.*

Wu, H., Diamos, G., Cadambi, S., & Yalamanchii, S. (2012). Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. *45th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 107-118). IEEE.

Zidan, M., Bonny, T., & Salama, K. (2011). High performance technique for database applications using a hybrid GPU/CPU platform. *Great lakes symposium on VLSI* (pp. 85-90). ACM.