UNIVERSITY OF CALIFORNIA
RIVERSIDE

Using Multithreaded Techniques to Mask Memory Latency on FPGA Accelerators

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Robert Joseph Halstead

March 2015

Dissertation Committee:

Dr. Walid A. Najjar, Chairperson
Dr. Vassilis Tsotras
Dr. Nael Abu-Ghazaleh
Dr. Zizhong Chen

The Dissertation of Robert Joseph Halstead is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I thank my committee, without whose help, I would not have been here.

To my parents and family.

ABSTRACT OF THE DISSERTATION

Using Multithreaded Techniques to Mask Memory Latency on FPGA Accelerators

by

Robert Joseph Halstead

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2015
Dr. Walid A. Najjar, Chairperson

The performance gap between CPUs, and memory memory has diverged significantly since the 1980's making efficiency memory utilization a key concern for any application developer. Modern CPUs will process orders of magnitude more data than their memory architectures can sustain. Multiple levels of caches are used by the major CPU architects to cope with this issue. Frequently used data is stored as close as possible to the core, which allows it to be retrieved in a few cycles. Compared to the thousands of cycles it would take to be retrieved from main memory. However, data locality is important for caches to be effective, and as applications become more and more irregular the CPU's performance drops. This causes many important applications (e.g. sparse matrices, graphs, hash tables) to suffer from poor performance. This thesis explores how custom hardware accelerators using memory masking multithreaded techniques can be used to improve performance. In hardware multithreaded designs thread states are managed directly in hardware, and with enough application parallelism they can fully mask memory latency without storing data in caches. Designs scale well to match the memory architectures bandwidth. The emergence of heterogeneous FPGA platforms has made it easier to build and test the design on real world hardware.

This thesis starts with the issue of programmability. Hardware development is notoriously difficult and time consuming. The CHAT tool, a C-to-VHDL compiler, is presented that assists developers by generating custom multithreaded kernels from high level software descriptions. The thesis proceeds by using CHAT to generate a custom Sparse Matrix Vector (SpMV) kernel. Results show how multithreading can provide data independent performance. Compared to the software and GPU performance which drops significantly as the benchmark's irregularity increases. Cache miss rates increase on the CPU, and memory cannot be coalesced as efficiently on the GPUs. Finally we use multithreading to accelerate two common database operations: Hash join, and aggregation. They are the first in-memory implementations on FPGA hardware. The hash join design shows an improvement of 2x over the best multicore software designs available, and does so with 33% less memory bandwidth. Aggregation shows comparable performance when generating hash tables. However, it generates multiple tables that need to be merged, and this step reduces performance on high cardinality datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor performance has significantly outpaced memory performance creating a latency gap. The major chip manufactures rely on multilevel cache designs to help their CPU processors maintain high performance. Caches leverage the spacial and temporal locality within an application to reduce the number of requests back to main memory. Frequently used values are stored closer to the processing core where the latency is much lower. The paradigm woks well for many applications, but there exists many important applications that do not work well on caches. We call them irregular applications, and by definition they have poor locality. Applications like sparse matrices, graphs, hash tables, etc. Developers working on these problem often try to remove irregularity or find ways to improve locality. They are contorting their problems to work with commodity processors. Table 1.1 shows a few processors Intel has released in its Xeon line. A trend emerges showing a continual growth in the shared L3 caches size. A similar pattern can be seen with other chip manufactures, and suggests that the caching model will be around for the foreseeable future.

Table 1.1: Cache sizes for Intel Xeon Processors.

Processor	# of Cores	L2 cache size	L3 cache size
E3-1290	4	256 KB	8 MB
E5-4620	8	256 KB	20 MB
E7-8880	15	256 KB	37.5 MB

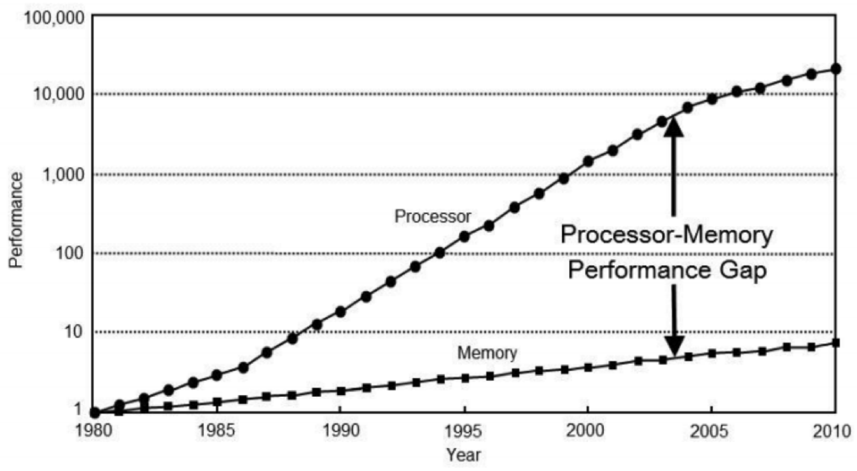


Figure 1.1: Graph comparing the performance of single core processors to memory from the 1980's to the 2010's.

Figure 1.1 shows how the gap between CPU and memory performance haw widened since the early 1980s up until the 2010s. Modern processors can now generate 3 orders of magnitude more requests than the memory architectures can fulfill. This high request rate coupled with long access latencies means that processors spend the majority of their time idle. Caches hold data for common memory addresses closer to the processing units, which shortens the access time by limiting requests to global memory. CPU cores request data from the on-chip cache, and if the request hits the value is returned within a few cycles. Much faster than the thousands of cycles needed to fetch data from main memory. When a request misses, or needs to be written back, the architecture will issue a global request, but it is only done when necessary.

Applications with predictable memory access patterns are called regular applications. They often have high locality, and are well suited for caching platforms. Application with unpredictable memory access patterns are called irregular applications. They often have poor locality, but developers still try to make them work on commodity hardware (i.e. caching CPUs). Their workloads pull data from many different memory locations jumping around memory in seemingly random ways. Two examples of irregular applications that this thesis addresses are hash tables, and sparse matrices. Hash tables rely on good hashing functions that randomly distribute keys across a range of values. Two jobs that accesses the table one after the other can be reading or writing to vastly different physical locations in memory. This thesis uses hash tables in the context of database applications, but they are also used for filtering, virus detection, and ironically caching. Sparse matrices are used for graphs, economic models, and simulations. They are often large with few non-zero elements. Storage formats are very important for fast processing, but because the non-zero values can be spread erratically around the matrix these formats introduce irregularity. Sparse Matrix Vector multiplication (SpMV) is a common operation that is considered in this thesis.

Latency masking multithreaded architectures are an alternative to the caching model common in todays CPUs. Commercial machines like the Tera MTA [4], or Cray XMT [45] have been used in research labs since the early 1990s. They came from research architectures like the Denelcore HEP [60], and Horizon [36]. They provide direct hardware support to manage a large number of threads on a single core. When one thread issues a memory request it relinquishes execution, and it is context switched out for another. The thread is unstalled once the request has been fulfilled, and ut can continue executing. These platforms support enough threads to fully mask the memory latency. If all threads were to issue a request one after the other by the time the last request is issued the data for the first thread will be available. Identifying enough parallelism in an application is the hurdle preventing these platforms from becoming general purpose machines. The Sun UltraSPARC T series processors offer a middle ground between multithreaded, and cache architectures. They rely on caches, but also offer direct hardware support for 8 to 16 concurrent threads.

Field Programmable Gate Arrays (FPGAs) have shown massive speedup potential for a wide range of applications. Their ability to support highly parallel designs, coupled with their re-programmability have made them very attractive platforms for regular applications. Custom pipelined datapaths allow the FPGA to execute in parallel what could take thousands of operations in software. They can outperform software designs by 100x to 1000x all while reducing energy consumption. Image processing [17, 69], computer vision [33], data mining [57], bioinformatics [21, 22], financial analysis [56], and streaming databases [44, 47] are just a few of areas where FPGAs can been used.

Programmability, and ease of use deter many software developers from expanding into hardware development. FPGAs are notorious for complex designs, long debug cycles, and difficult verification among other things. Some if these difficulties are inherent to the platform. Hardware design is fundamentally different from software design, and require a shift in thinking. However, some of these issues can be alleviated by advances in hardware design tools. Major FPGA manufactures are actively developing High Level Synthesis (HLS) tools to help software software developers utilize their boards. The goal for HLS is to compile a high-level language (C, C++, Java, etc.) directly to RTL. These tools could gain more interest as industry continues to develop new heterogeneous architectures that incorporate FPGAs. Companies like Convey Computers [16], Maxeler Technologies [42], and Pico Computing [53] are already already offer them. Software APIs allow developers to easily interface software execution with their hardware. Depending on the architecture data can be offloaded to the FPGA similar to GPU platforms, or the FPGA can have direct access to global memory. FPGAs now have easy access to significantly larger memory spaces, which allows researchers to consider much larger real-world problems. However, the larger memories come at a cost of higher latencies. It is an open question, the best way to deal with longer memory access times. Caches are tried, and true for CPUs, but FPGAs offer a level of parallelism that can be leveraged by other methods.

This thesis considers how multithreaded architecture approaches to latency masking can be applied to the emerging FPGA memory latency issue. It starts by addressing the FPGA programmability issue. The Compiled Hardware Accelerated Threads (CHAT) tool is a C-to-VHDL compiler that can be used to create custom multithreaded circuits. It is intended to improve development time for application developers who are not familiar with hardware by generating synthesizeable VHDL. A standard FIFO interface is used to request, and collect data making the circuit platform independent. Developers only need to interface it with their memory architecture. CHAT is then used to develop a Sparse Matrix Vector (SpMV) kernel. Finally, this thesis considers how latency masking multihreading can be applied to common relation database operations. It presents the first end-to-end in-memory hash join implementation entirely on an FPGA. Custom multithreaded circuits are presented for the join's build and probe phases. A custom multithreaded circuit is also presented for aggregation.

Chapter 2

Background

FPGAs are traditionally used for streaming, and regular applications. They provide developers a highly parallel platform that can allocate tens to thousands of individual components, and run them all concurrently. Custom pipelined datapaths can be created for regular applications because by definition they have predictable memory access patterns. In this domain FPGAs have been used to achieve significant speedups over software designs [44, 72, 47, 33].

The FPGA community has began pushing to include reconfigurable fabrics in heterogeneous architecture. Often this required labs to build one-off custom machines to test their designs. However, since the mid 2000's commercial platforms have began to enter the market. This thesis looks to expand the application domain of FPGAs by considering applications that are not streamable. That do not have regular memory access patterns. We incorporate known techniques like latency masking and multithreadding. The FPGA now decides the memory locations it needs, instead of being simply pushed new values. We apply our designs to sparse matrix and database applications.

2.1 Latency Masking Multithreaded Architectures

Multithreaded architectures provide direct hardware support to manage multiple threads on a single core. This differs from modern multiprocessor architectures that can execute multiple threads in parallel, but do so on independent cores. A single multithreaded core can handle 10s to 100s of concurrent threads while a single multiprocessor core handles only one, or two with ”simultaneous multithreading” technology.

In parallel applications a multithreaded architecture can offer better core utilization than a multiprocessor architecture. Multiprocessor architectures use multitasking to interleave thread execution, which allocates time slots for threads to be scheduled into. In the simple case each thread is scheduled in round-robin fashion and given equal time to execute. However, if a thread is waiting for a resource (i.e. a cache miss) it will still be scheduled into a slot, and during this time the multiprocessor core will be idle. In contrast the multithreaded architecture has dedicated hardware resources for thread management. When a thread needs an unavailable resource it issues the request and goes into a stalled state. The multithreaded core is then free to execute non-stalled threads, and it only goes idle if all threads are stalled.

The extra hardware to manage thread states typically results in slower clock frequencies for multithreaded arhcitectures compared to multicore ones. Longer pipelines are also common to manage the context switching, which could hurt performance on single threaded applications. However, this is more of a concern for sequential, than for parallel, applications.

2.1.1 Full Latency Masking Architectures

Memory masking multithreaded architectures are not a novel idea. They have existed in some form since the Denelcore HEP [60, 30], and Horizon Architecture [36, 68] was proposed in the late 1980's, if not earlier. The idea was simple. Architects first measured how many clock cycles it took to fulfill a memory request. In the Horizon's case most requests averaged between 50 to 80 cycles, but almost all requests could be handled within 128 cycles. The architects then built custom processors to supported that many outstanding requests; 128 threads in the case of the Horizon. The processors had very fast context switching (one clock cycle) so that once a request was issued by a thread it could immediately switch to another thread. In this way the processor was fully utilized. In the worst case all 128 threads would issue a memory request. However, by the time the 128th request was issued the 1st request would be fulfilled, and the processor could continue running without interruption. This technique is called memory masking, and is integral to a multihreaded architecture's performance.

The Tera Computer Company released its Tera MTA [3, 4, 61] machine. Each of its processors could run at 300 MHz, and they could support 128 hardware threads. The only physical machine, that we are aware of, was installed at the San Diego Supercomputer Center [10] and it contained 4 processors. Therefore, It could support 512 threads. To lower the network traffic the MTA's instructions were fetched through a shared cache. However, it had no data cache and relied purely on multithreading to mask the memory latency.

The Tera Computer Company eventually merged with the research division of Cray. They continued to develop their multithreaded platforms; Cray MTA-2 (2002), and Cray MTA-3/XMT (2009). The MTA-2 was not widely adopted with one unit being sold to the United States Naval Research Laboratory. The XMT was much more successful in the research community [45, 71, 25, 15]. It supported up to 8,192 processors each running at 500 MHz, and they could share 128 TBs of RAM [20]. However, the largest machines sold only contained 64 processors, but the Cray research labs built and tested machines with up to 512 processors.

The main challenge for multithreaded architectures has been to extract enough parallelism to fully utilize their processors. Identify parallelism at compile time or runtime is a non-trivial task. This issue is a major factor preventing these architectures from being general purpose machine, and being more widely adopted. However, they have found use in the High Performance Computing industry because they perform well on irregular applications. Modern CPUs rely on efficient caching to sustain a high performance, and they struggle with the poor spatial and temporal locality inherent to irregular applications. However, memory masking is unaffected by the irregular access patterns.

Table 2.1: Specifications for the UltraSPARC T series procesors.

	Year	Cores	Threads/Core	Cache	Clock Freq (GHz)
UltraSPARC T1	2005	8	4	3 MB L2	1.4
UltraSPARC T2	2007	8	8	4 MB L2	1.6
UltraSPARC T3	2010	16	8	6 MB L2	1.67
UltraSPARC T4	2011	8	8	4 MB L3	3.0
UltraSPARC T5	2013	16	8	8 MB L3	3.6

2.1.2 Sun UltraSPARC T Processors

Starting in 2005 Sun produced a line of multithreaded multicore processors based on its UltraSPARC architecture; Table 2.1. The first processor (UltraSPARC T1) had 8 cores that could support 4 threads concurrently. Each core would context switch between active threads each cycle. This increased the overall latency of a single thread, but allowed the cores to be better utilized. Threads did not need to be from the same application, but cache performance improved if the threads were accessing the same data locations. Sun stedaly improved the architecture by adding more floating-point units, memory bandwidth, and increased the number of cores. However, when the UltraSPARC T4 was released they decreased the number of cores, but dramatically increased the clock frequency. This was a move to broaden their target audience by improving the single thread performance.

2.2 Sparse Matrix Vector Multiplication

Sparse matrix Vector (SpMV) multiplication is a very important kernel to Scientific, and High-Performance computing. It has been widely studied for many years. Most research has focused on how to best store the data in memory; 1. minimize the area (memory footprint), and 2. improve a platform's memory accesses. Many formats have been proposed from the simple Coordinate (COO) or Compressed Sparse Row (CSR) formats to the specialized ELLPACK and Diagonal (DIA) formats. No one ideal format exists for all matrices on all platforms. In this section we present the common sparse matrix storage formats, and report the significant results and approaches for SpMV using hardware accelerators.

2.2.1 Sparse Matrix Formats

The first concern of any sparse matrix application is how to best store the matrix in memory for the given architecture. Many different approaches have been proposed throughout the years, and we highlight the most general ones here. The

$$\begin{bmatrix} 2 & 0 & 7 & 6 & 5 \\ 0 & 8 & 1 & 0 & 1 \\ 4 & 9 & 0 & 0 & 5 \\ 0 & 0 & 7 & 6 & 0 \end{bmatrix}$$

Figure 2.1: An example matrix that is used to show how different sparse matrix formats store data.

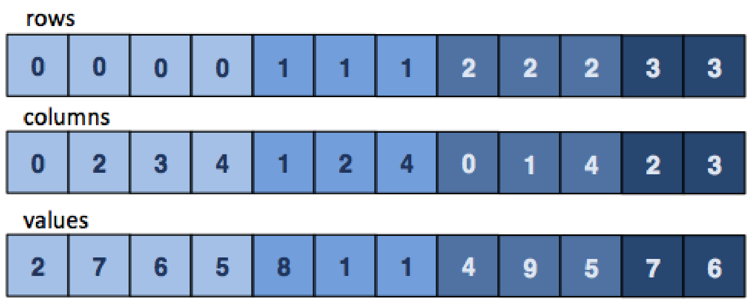
memory masking model is unconcerned with the access patterns, and therefore we are interested in finding the matrix format with the smallest memory footprint. We use the matrix in Figure 2.1 to help show how each format stores its data.

2.2.1.1 Coordinate Format

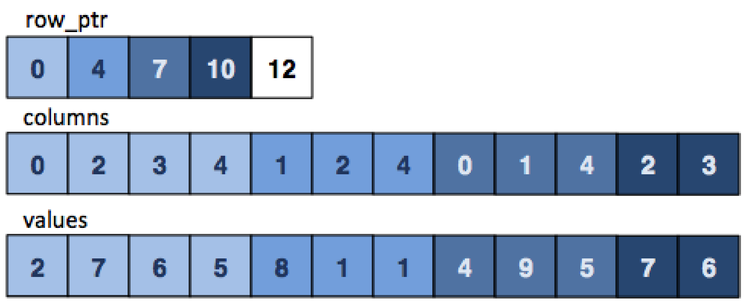
The coordinate format (COO) is the most intuitive approach to storing a sparse matrix: for each non-zero element it stores its row position, column position, and value. As shown in Figure 2.2(a) each data point is stored in a separate array. Each non-zero element is assigned a unique index, and this index is used to access each array. In our example we assign the indices in sequential order, but this is not a requirement of COO. Non-zero elements can be spread randomly throughout the array so long as the row, column, and value share the same index within their respective arrays.

2.2.1.2 Compressed Sparse Row

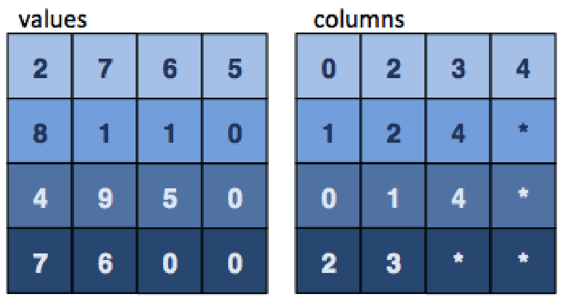
The compressed sparse row (CSR) is a natural extension to the COO format and is the most commonly used format. It stores all the column position, and value points in two separate arrays; just like COO. However, it saves memory by compressing the row position. As shown in Figure 2.2(b) some row positions are repeated multiple



(a) COO format



(b) CSR format



(c) ELL format

Figure 2.2: COO, CSR and ELL sparse matrix storage formats for the example matrix in Figure 2.1.

times. Instead of storing repetitive values the CSR format uses a row_ptr array, which only points to the first element of each row. To be functionally correct the column, and value array must be sorted by row such that all elements in row i are placed before any element in row $i + 1$.

The compressed sparse column format (CSC) is almost identical to CSR, but instead of compressing the row array it compresses the column array. The CSC format is less widely used because its workloads are harder to distribute evenly. SpMV has one output per row (not per column), and with CSR each output can be computed within a single processing element (PE). CSC would require synchronization across multiple PEs to prevent race conditions.

2.2.1.3 ELLPACK

The ELLPACK (ELL) format rose in popularity with vector processors. Peak performance required each processor to run the same workload. To achieve this ELL would zero pad the smaller rows until they were the same length as the largest. Figure 2.2(c) shows how our example matrix would be stored in ELL format. Because all rows are the same length the ELL only requires 2 arrays; one for the columns, and one for the values. The column position for zero padded data can have any value, but it should be less than the number of columns in the matrix to avoid out-of-bound memory accesses.

ELL is not as general a sparse matrix format as COO or CSR. It is only useful when all rows within a matrix have a small variance on the number of non-zero elements per row. If one, or a few rows are much larger than the rest it can cause too much zero padding. The extra work from processing zero values would eliminate any of performance gains.

2.2.1.4 Hybrid format

A hybrid (HYB) format was proposed by [8] for GPU accelerators. It splits the matrix storage between the ELL, and the COO formats. Data that can be reasonably (low zero padding) stored in ELL will be, and the remaining values are stored in COO format. This method both reduces the zero padding caused by longer rows, and it still allows the GPU to benefit from ELL's structured memory; at least for part of the computation.

2.2.2 FPGA Approaches

The FPGA architecture is well suited to streaming application paradigms, and as such much FPGA research has focused on streaming applications. Sparse Matrix Vector multiplication (SpMV) is no exception. Most published research has focused on removing all irregularity for SpMV, and then using an FPGA to improve performance. The most common approach is to store the vector (i.e. the point of irregularity) in on-chip BRAMs. Allowing each vector accesses to be handled in one cycle regardless of the access pattern. Modern high-end FPGA do have large quantities of BRAMs on-chip [76], but they are at most a few megabytes. Not reasonable for the high performance computing environment where SpMV matters.

Substantial SpMV work has focused on how to best implement the floating-point Multiply Accumulate (MAc) circuit. Floating-point multiplication takes around 12 cycles, and addition can take up to 27 cycles. Managing states during the accumulation is non-trivial because row lengths can vary drastically even within the same benchmark. Adder tree structures [64, 80] with feedback loops can be used to handle multiple row elements in one cycle. However, the number of non-zero elements per row must be larger than the number of channels. Otherwise the design is underutilized, and loses performance. Other approaches have considered statically assigning the partial dot products to multiple processing engines [19, 58]. Here a control unit manages the communication, and ensures proper execution within the design. However, the number of concurrent rows in the MAc circuit is limited to at most two at any given time. However, multiple MAc circuit can be placed on the FPGA to support more rows in parallel [78]. [24] developed an accumulation reduction circuit that supports an arbitrary number of rows, and can read a new value every cycle. However, all data from one row must enter the circuit before any data from another row enters. A control unit arbitrates the data flow between the floating-point addition unit and temporary buffers.

The emergence of heterogeneous FPGA platforms by Convey Computers, Maxeler, and Pico has allowed researches to easily implement their designs on real world hardware. A custom SpMV personality was develop for the Convey HC-1 [49]. The design caches memory requests locally in case data needs to be reused, but because of the irregular nature of SpMV it could only sustain 40% of the peak performance.

2.2.3 GPU Approaches

GPUs are large vector machines that rely on SIMD parallelism to achieve peak performance. The have a fixed architecture that was developed to quickly process graphics applications. However, with the introduction of the CUDA programing API in the late 2000's GPUs have begun branching further into high performance computing applications. Because the hardware is not customizable much of the research has focused on how to best reshape the data to best utilize the platform. Performance depends on either having a lot of data executing on the same instructions (SIMD), and/or coalescing the memory requests to fully utilized the memory bandwidth. GPUs often have a major memory bandwidth advantage over CPU, and FPGA platforms. The Nvidia Tesla K20 has 208 GB/s of bandwidth, and the newer Nividia Tesla K40 has 288 GB/s of bandwidth.

The ELLPACK sparse matrix format was developed to run sparse matrix applications on early vector processors, and is a common starting point for many SpMV GPU ports. Slice ELLPACK [46] partitioned adjacent rows together in strips, and each strip was stored as its own ELLPACK matrix. The strips could be reordered by size to improve the GPU's SIMD performance. R-ELLPACK [70] reordered the matrix such that rows accessing the same vector locations would be closer to each other, and therefore improving the locality. The Hybrid (HYB) [8] format stores most of the matrix in ELLPACK, but some elements (in rows where the number of non-zero elements is larger than the average) are stored in COO format.

There is no "one best" format for SpMV on GPU. If the data is random a generic format like CSR or COO may be best. If the data has dense areas spread throughout the matrix a blocked format may be best. If the data is structured a custom format may be best. The research community have empirically shown this to be true [7, 40]. The HYB format gives bester results, on average, when the matrix is unoptimized [8], but CSR has been shown better when the rows are first reordered into clusters [52]. The clSpMV tool [62] is an OpenCL SpMV solver that analyzes the matrix and chooses a format from one of three distinct categories (Diagonal, Flat, or Blocked).

2.3 Accelerating Database Operations with FPGAs

Data analytics is a driving force behind many businesses. Begin able to quickly identify a custom base, and their needs is paramount in the fast paced and competitive environment. Commercial platforms like IBM's Netezza [31] and Teradata's Kickfire [65] offer FPGA solutions for Database Management Systems. They cover a full range of database operations from selection and projection, to joins and aggregation. They work in conjunction with software to analyze workloads and run queries on the best available resources (software or hardware). Academia is also working to improve these systems with new and innovative hardware designs.

2.3.1 Query Processing & DBMSes

Academia has developed tools that compile quires down to hardware circuits using custom library components. The Glacier library [48] is a set of specialized building blocks that can be combined to make an engine for streaming queries. However, it is only practical for common queries that have a high re-use rate. The synthesis time to build an engine is high, and needs to be amortized over many runs to be practical. The technique has been shown useful for event processing systems like high frequency trading [56]. The Q100 [75] architecture is a fixed platform with many ASIC database processing units. A query stream is scheduled through the necessary units. Resources may go unused for a given query, but the platform avoids long build times.

Netezza [31] is a complete DBMS that uses FPGAs as a filter between the hard disk and main memory. Customizable queries are sent to the FPGAs which utilize their close proximity to the hard disk to quickly filter relations before sending them to memory. The platform tries to reduce the costly data transfers from disk to main memory [23]. The trade off for this approach is that all requests must start on disk. In-memory databases cannot leverage the addition hardware FPGAs.

Another full DBMS, Kickfire [65], uses FPGA hardware accelerators connected through either PCIe or hypertransport. It defines various database operations as HARP logic [35] that consists of a hardware circuit and a large memory systems. All queries are analyzed by Teradata's C2 software, which decides if it should handle the job itself, send it back the the DBMS, or offload it to HARP logic. The customized hardware supports many common relational database operations [12, 29, 43]

2.3.2 Join Operations

The stream join operation is a common in on-line algorithms where a complete result is not needed. The algorithm is only concerned with joining tuples within a window of data, and because of this the entire window can be streamed to a processing element and handled. This differs from the traditional relational join which has to find all matches between two relations. The streaming nature made it an ideal target for FPGA acceleration. The Handshake Join algorithm [66] treats the two relations as athletes congratulating each other after a match. Data streams through the FPGA in opposite directions. Two windows of data (one for each relation) are held entirely on the FPGA, and custom hardware comparators look for matches. When a match is found the tuples are joined and output. The design can be easily extended to larger window sizes as the FPGA chips continue to grow in resources, or multiple FPGAs can be chained together.

Joining tuples can be an issue for the design. When multiple tuples match their outputs need to be serialized through a limited number of channels. However, the Handshake Join implementation has been extended to include an admission control unit [50, 51]. The unit stalls the stream until the FPGA has enough resources available to continue. This throttling of inputs is important on applications that join a lot of tuples.

The relational join operations is very important to OLAP workloads, and the push for real time analysis has lead researchers to explore FPGA designs. Sort-merge join, and hash join are the most common implementations. In software hash join has been shown to outperform sort-merge [1], but a switch has been predicted as CPUs increase their SIMD register sizes [5]. Counter intuitively the FPGA community has focused their efforts on sort-merge implementations [55, 75, 14]. The reason for this is simple. FPGAs excel at streamable regular applications like merging sorted lists. Therefore the trick is to develop a streamable sorting algorithm in hardware, which can be done with sorting networks [37, 32].

Chapter 3

CHAT - Compiled Hardware Accelerated Threads

Designing a hardware accelerator is a difficult task. Not only are the platforms significantly different from the traditional Von-Newman model, but so too are the development cycle and tools. The learning curve is often steep even for classically trained software developers. The divide is so drastic that developers who write hardware accelerators often do not refer to themselves as software developers. In this section we present the CHAT tool to help bridge the gap between hardware and software development.

The CHAT (Compiled Hardware Accelerated Threads) tool is designed to assist developers with implementing irregular applications on FPGAs. Ease of programmability has historically been a roadblock preventing many developers from exploring FPGA design options. A steep learning curve coupled with a long debug cycle has made the startup costs too high. However, there are many performance and energy benefits to be gained from using reconfigurable custom hardware. CHAT attempts to lower the startup costs by allowing developers to describe their circuits in a high level language, and simply compile the design into a Hardware Description Language (HDL). We are not alone in this research. Many academic and industry initiatives are attempting to address this issue with High Level Synthesis (HLS) tools. Among the most mature are Vivado HSL[77] (formerly AutoESL[79]), Altera OpenCL (AOCL)[2], ROCCC[72, 54] and LegUp[13, 39].

HSL tools make differing assumptions about their target applications, their target architectures, and their design goals. These limitations are necessary because the nature of hardware development is so different from that of software development. FPGA accelerators are typically used for regular/streaming applications where the data has a high spatial and temporal locality. The FPGA's highly parallel nature allows it to execute many operations concurrently for a massive performance increase. The typical HLS tool targets these regular applications, but it is often at the expense of irregular application support.

Vivado HLS, and AOCL are developed by Xilinx and Altera, respectively, and therefor designs are tightly coupled to their company's FPGA platforms. Furthermore, both these tools only generate designs where all relevant data is stored local to the FPGA chip. Doing so allows the tools to handle both regular and irregular applications because all data accesses can be handled in one cycle. However, major restrictions are placed on the size of problem sets. The largest FPGAs currently available only offer a few MBs of local storage, and many interesting problems often require many megabytes, or more of data.

ROCCC and LegUp both assume an off-chip memory source, which makes them more generalized in the size of problems they can solve. However, ROCCC and LegUp make different assumptions about their target architectures. ROCCC is very general and generates VHDL that is synthesizes on any platform. Designs can be unrolled, and the number of I/O memory channels are customizable. In contrast LegUp generates Verilog specifically for Altera boards, and designs are limited to two memory channels. Therefore, LegUp kernels cannot sustain good performance on high bandwidth machines. These tools also make different assumptions about their target algorithms. LegUp is very general and can interface with a TigerMIPS soft-core processor to handle operations that are unreasonable as digital circuits. ROCCC always assumes streamable regular applications which allows it to generate highly optimized kernels. These difference arise because both tools were designed with different goals in mind. ROCCC builds faster, and scalable designs for specific applications, and LegUp supports a wider range of applications.

CHAT is a complement to the ROCCC Compiler. Like ROCCC it is platform and memory independent allowing for easy porting to emerging FPGA architectures; Convey Computers[16], Pico Machines[53], Maxeler Technologies[42], etc. However, where ROCCC generates highly optimized kernels for streaming applications CHAT generates less optimized, but still highly efficient, kernels for irregular applications.

3.1 Taxonomy of Irregular Applications

FPGA accelerators are not a general purpose solution for any software application with poor performance. They are specialized platforms with a set of features that are well suited to particular applications. Historically, this meant streaming and compute bound applications. However, FPGAs also offer potential benefits for highly parallel irregular applications. Modern CPUs rely on good caching algorithms to maintain performance, but by definition irregular applications have poor spatial and temporal locality. Multithreaded architectures were proposed [36, 68, 4, 3, 20] as a possible solu-

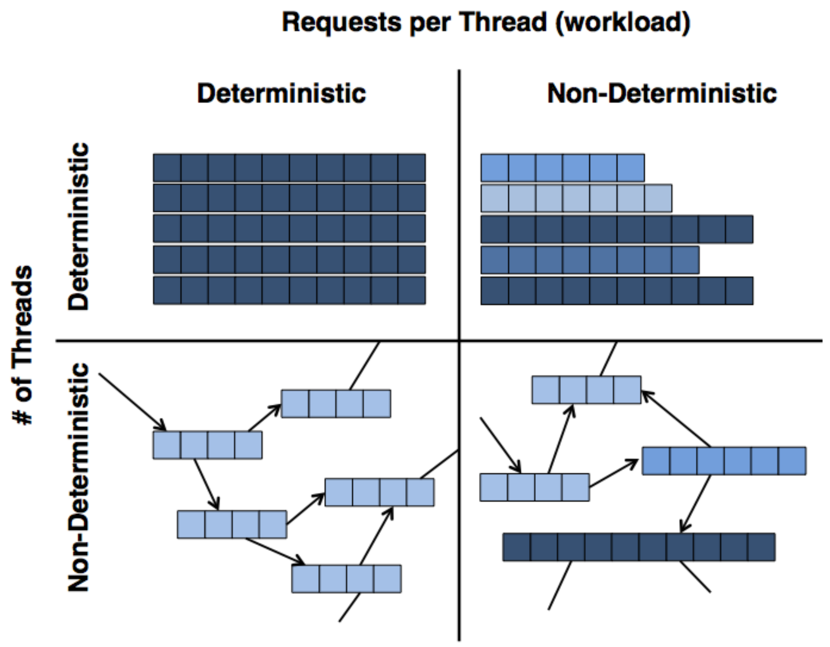


Figure 3.1: A taxonomy of irregular applications where the number of threads and the workload sizes are, or are not deterministic.

tion, but their design was not reasonable for general purpose computing. FPGAs offer a middle ground where CPUs can offload certain jobs to custom multithreaded FPGA accelerators.

The multithreaded paradigm requires sufficient parallelism within an application to fully mask long memory latencies. Therefore, not all irregular applications are reasonable on such architectures. Here we classify the different types of irregular applications could have. We also describe how these applications may look within C code, and explain the various FPGA constructs needed to implement these applications as digital circuits.

The CHAT tool defines a thread as any output result to memory, and every application will have at least one. Distinct threads can access the same memory location, or even overwrite previous results. CHAT applications can fall into one of 4 classes as shown in Figure 3.1. At runtime the application's total number of threads can be deterministic, or not. In addition each thread can have a deterministic, or non-deterministic number of reads and writes.

Applications where the number of threads are deterministic can be optimized to reduce, or eliminate redundancy. For example consider an application where N threads read from the same memory location. The compiler can merge them into a single request, and propagate the value to the N datapaths eliminating $N-1$ requests. Applications with a non-deterministic number of threads cannot merge datapaths so they need constructs to prevent, or limit redundancy. For example consider traversing a graph with breadth-first search. Each node in the graph is treated as a thread. New threads are continually generated during the traversal. However, Two nodes, A and B, could both point to the same third node C. Without synchronization two threads would be generated for node C, and performance would begin to decrease exponentially as more redundant threads are created and processed.

A single hardware Processing Engine (PE) is assigned a thread, or multiple threads. To improve a design's parallelism and performance multiple PEs can be used in the hardware kernel. However, doing so requires scheduling of threads to the available PEs, and this can be done statically or dynamically depending on the type of application. If an application's workload (requests per thread) is deterministic the scheduling can be optimized at compile time. For example, in the simple case all threads have the same workload. Therefore, the datapath can assign threads to PEs in round-robin fashion. If an application's workload is non-deterministic then the hardware needs a runtime Thread Management Unit (TMU). Each PE uses flags to signal when it can handle a new job. The TMU queues up jobs, and assigns them dynamically as PEs become available. This dynamic scheduling allows long jobs to occupy a single PE while the other PEs handle many smaller jobs.

3.2 The CHAT Compiler

CHAT is a C to VHDL compiler for irregular applications. It is based around research done for the ROCCC [72, 54] tool, but extends the infrastructure to support irregular applications. CHAT analyzes kernels at two levels. First high-level analysis builds a Data Flow Graph (DFG) using the SUIF 2.0 toolset [74] and generates an intermediate representation. Next low-level analysis creates a Control Flow Graph (CFG) using the LLVM compiler [38] and generates the VHDL design.

Algorithm 1 C code for a simple irregular application.

```
void passthrough (int *A, int *B, int *C, int length) {
    int i;
    for (i = 0; i < length; ++i)
        C[i] = B[ A[i] ];
}
```

3.2.1 Hi-CIRRF in SUIF 2.0

High-level analysis reads the hardware kernel described in the C language. However, because C was developed as a software language, not all its constructs make sense in a hardware context; e.g. Dynamic Allocation, Recursion, pointer arithmetic, etc. Therefore, CHAT only supports a subset of the language. For example arrays are treated as streams of data. They occupy contiguous blocks of memory, but they can be randomly accessed. CHAT does not currently support pointer chasing. Rows for multi-dimensional arrays are assumed to be stored one after the other. Constant values are stored in hardware registers. Variable values are temporarily stored in registers,

Algorithm 2 Sample of the CIRRF generated by CHAT.

```
...
CHAT_init_inputscalar(length) ;
CHATInputStreams(A,B) ;
...
for (i = 0 ; (i < length) ; i = i + 1) {
    CHATInputFifo1_0(A, i , suifTmp2 , 0) ;
    CHATDataFifo1_2(B, suifTmp2 , suifTmp3 , 0);
    suifTmp4 = CHATIntToInt(suifTmp3 , 32) ;
    ...
    CHATOutputFifo1_2(C, i , suifTmp4 , 0) ;
    CHAT_output_C_scalar() ;
    CHATOutputStreams(C) ;
}
```

and the values are moved around the hardware datapath. Branches in execution will generate multiple datapaths that are filtered through a multiplexer.

The goal of high-level analysis is to identify the hardware components, and create a dataflow graph (DFG). New passes are added to the SUIF 2.0 [74] compiler to achieve this goal. We use Algorithm 1 to highlight the major steps in CHAT, but each step may require multiple passes in SUIF. First CHAT's high-level analysis will identify 3 data streams (A, B, and C), and one registered value ($length$). Two streams will be identified as input streams (A, and B) because they read values, and one stream will be identified as an output stream (C) because it is written too. CHAT does not support streams that both read, and write. Temporary registers are created, called 'suifTmps', to direct the datapath. Finally, the CIRRF [27] is output. A sample of the CIRRF file is shown in Algorithm 2.

User inputs are used to further customize the Hi-CIRRF pass. Parallelism can be increased by unrolling the for loops which will duplicate the datapaths in the DFG. Doing so could cause redundant hardware. CHAT will analyze the unrolled design, and merge components working on the same exact data.

3.2.2 Lo-CIRRF in LLVM

CHAT's low-level analysis reads the DFG from the Hi-CIRRF pass. It creates a control flow graph (CFG), and then generates the synthesizable VHDL. Thread management is a key consideration during this phase. Each thread must maintain its state locally on the FPGA, but because of the FPGA's parallelism multiple threads can be changing states in the same clock cycle. All thread data is stored on-chip in BRAMs, which are configured as FIFOs. This can be done because CHAT assumes all memory requests are returned in-order. However, the compiler could be extended to support out-of-order memory requests. In this case a design would implement CAMs instead of FIFOs.

The Lo-CIRRF compilation is implemented by a number of different passes in in the LLVM [38] compiler. Here we give a high level overview of what happens, but just as in the previous section each step may require many different passes. First the compiler assigns each CIRRF statement into their own basic block, which allows parallel scheduling for non-sequential operations. A element must block until its dependencies have valid data, but other elements are free to execute. Buffers are placed throughout the datapath to limit stalling, and alleviate back-pressure. Paths requiring memory requests will also be padded with large buffers to allow multiple outstanding requests.

Portability is another goal for the CHAT tool. Designs assume nothing about the FPGA board it will be implemented on. However, the compiler does create hooks for developers to leverage for performance. They are important for complex operations (i.e. division, or floating point operations) where custom DSP blocks are often available, but are usually board dependent. The compiler will generate simple FIFOs for small buffers, but for larger buffers the compiler provides hooks to the developer. Custom IP cores can therefore be used to improve timing, and area utilization.

3.3 Simple Irregular Applications

In this section we use the CHAT tool to build two simple irregular applications. We also show throughput performance results on a Convey HC-2ex machine. Both are summation circuits. The first design uses a 1-dimensional stream as the index, and the other design uses a 2-dimensional stream.

Algorithm 3 Summation kernel with a 1-dimensional index stream.

```
void summation(int **A, int *B, int *C, int m, int p) {
    int i, j;
    for(j = 0; j < m; ++j)
        for(i = 0; i < p; ++i)
            C[j] += A[j][B[i]];
}
```

3.3.1 One Dimensional Indexing

We implement the basic irregular application expressed in Equation 3.1. The actual CHAT code for this equation is shown in Algorithm 3. In this design array B has a regular access pattern, which iterates from 0 to m. However, array A has an irregular access pattern because it uses values from B in its index. The values in B can be anything.

$$C[m] = \sum_{i=1}^{p} A[m, B[i]] \tag{3.1}$$

The CHAT tool will generate two input controllers (A, and B), and a single output controller (C). It will also generate two counter components for variables i and j. Finally the compiler will create a summation datapath. Data from the i counter will be used for the B input controller to issue memory requests. As the results return they will be routed to the A input controller where they are combined with the j counter for a new memory request. The memory requests for A are routed into the summation datapath, and once p elements have been accumulated the result is sent to the C output controller. The output controller writes the final result to memory.

The design requires only three memory channels, and should be replicated to utilized the Convey's 16 memory channels. Replicating the design is done through the compiler by unrolling a for loop. Unrolling the inner for loop increases thread level parallelism. Each cycle a single thread issues multiple request, which go into a summation tree. Unrolling the outer for loop increases application level parallelism. Each cycle multiple threads are executing in parallel. However, this also means the same *B* values can be used by all *A* controllers. The compiler identifies this, and generate a single *B* input controller, and routes the value to all *A* input controllers.

Algorithm 4 Summation kernel with a 2-dimensional index stream.

```
void summation(int **A, int **B, int *C, int m, int p) {
    int i, j;
    for(j = 0; j < m; ++j)
        for(i = 0; i < p; ++i)
            C[j] += A[j][B[j][i]];
}
```

3.3.2 Two Dimensional Indexing

We also implement an irregular application that is indexed by a 2-dimensional stream as shown in Algorithm 4. CHAT will optimize this kernel very differently than the 1-dimensional kernel. The *B* input controller will provide different values for each row in *A*, and therefore it cannot be merged into a single shared memory channel. Multiple *B* input controllers must be created for each *A* input controller.

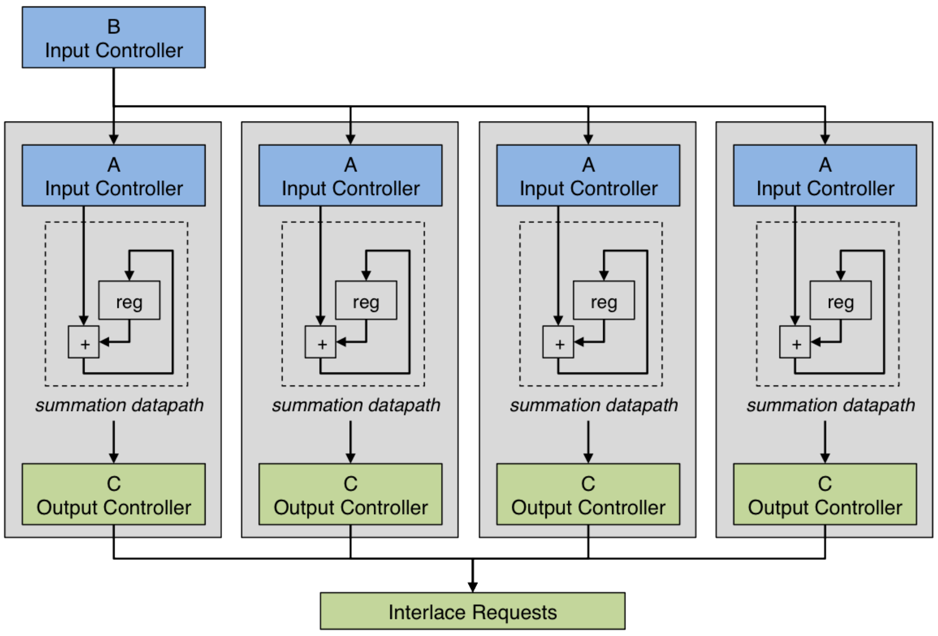


Figure 3.2: FPGA components for the 1-dimensional kernel as generated by CHAT. Notice that the B input controller is optimized to shared its data between all A input controllers.

3.4 Experimental Evaluation

In this section we show how the CHAT tool can be used to generate kernels for real world architectures. We implement both kernels on a Convey HC-2ex, and explain the our design considerations.

3.4.1 Convey HC-2ex Implementation

The CHAT tool has no limit to how many times a design can be unrolled, but in practice architecture limitations must be considered. The Convey HC-2ex used here uses Xilinx Virtex-6 760 boards with 16 memory channels per FPGA. The summation circuit is very space efficient so our design is memory channel limited, but larger designs may be area limited. The developer must consider these limitations when using CHAT. To best utilize the memory channels we must consider how CHAT will optimize the

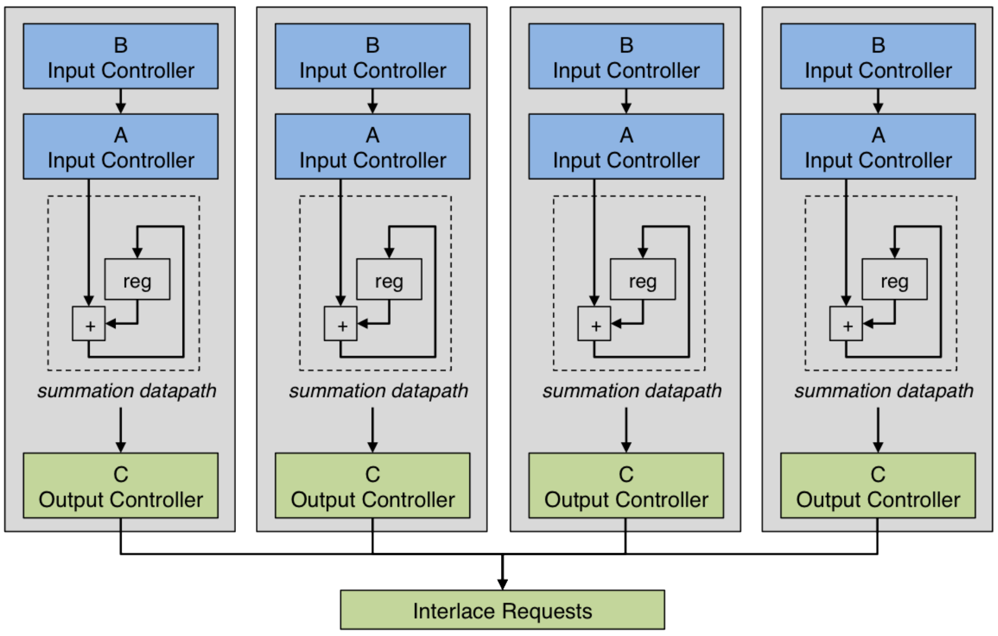


Figure 3.3: FPGA components for the 2-dimensional kernel as generated by CHAT. Notice that the B input controller cannot share its data between separate A input controllers.

1-dimensional kernel. The layout of components is shown in Figre 3.2. Each input memory controller (A and B) will continually request data, and therefore need their own dedicated channel. The design will have only a single B controller which is shared. Each output memory controller only has a single valid request per row, and therefore it is not continually writing results. Therefore, the C controllers can be interlaced into a single channel. Considering the HC-2ex's restrictions we can unroll the design 14 times. One memory channel is used by the B controller, another channel is used for all 14 C controllers, and 14 channels are used for 14 A controllers.

CHAT cannot optimize the 2-dimensional kernel to share a single B input controller. The layout of components is shown in Figure 3.3. All controllers for A and B still continually request data, but now there are multiple B controllers. The C controllers can still be interlaced into one memory channel. Therefore, the design can only be unrolled 7 times for the Convey HC-2ex. 7 channels are used for the

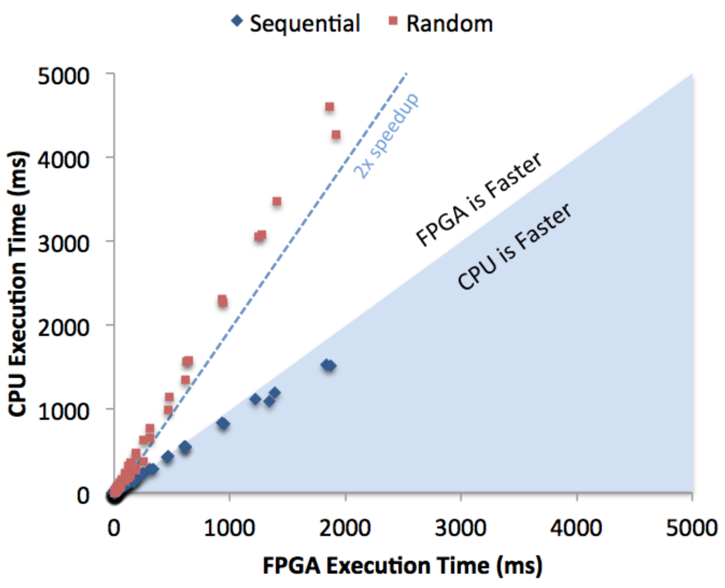


Figure 3.4: FPGA vs CPU runtime performance comparing sequential and random data on a 1-dimensionally index array. Dataset sizes range from 1 million 8-byte integers to 10-billion 8-byte integers.

B controllers, 7 channels are used for the A controllers, and 2 channels are used to interlace the C controllers.

Our experiments compare the performance of 2 FPGAs to one Xeon E5-2643 CPU. We use two FPGAs because the summation kernels presented here are memory bounded, and each FPGA only has 19.2 GB/s of bandwidth, which is much lower than the CPU's 51.2 GB/s. Using 2 FPGAs doubles the bandwidth to 38.4 GB/s, which is still lower than the CPU.

3.4.2 Runtime Performance

To test feasibility we gather runtime results on two different datasets. First we use purely sequential data to get a base-line measurement. This will yield the best performance for the cache dependent CPU tests. It also ensures the FPGA memory accesses will have few bank conflicts, which should minimize any stalling. The second

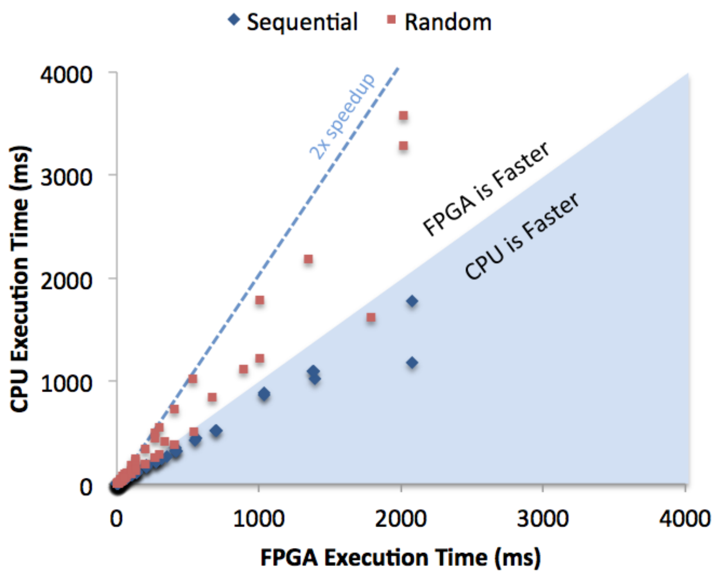


Figure 3.5: FPGA vs CPU runtime performance comparing sequential and random data on a 2-dimensionally index array. Dataset sizes range from 1 million 8-byte integers to 10-billion 8-byte integers.

datasets uses randomly generated values because we are interested in performance on irregular applications. Results are reported for both the 1-dimensional kernel (Figure 3.4) and the 2-dimensional kernel (Figure 3.5).

Looking at results for the 1-dimensional kernel, Figure 3.4, we can see that performance is comparable for sequential data. This is occurs for two main reasons. First sequential data has excellent spacial locality. Second this is a memory bounded application, and the bandwidth for the FPGA(38.4 GB/s) is comparable to the CPU(51.2 GB/s). The CPU does enjoy slightly faster performance, but it comes from the 33% higher bandwidth, and its order of magnitude faster clock frequency. However, on random data we begin to see the advantage of multithreading. The random data loses all spacial locality, and the CPU performance drops accordingly. Multithreading performance relies on an application's parallelism, and therefore the FPGAs performance is unaffected by different data. The results show that the FPGA can sustain about 2.4x speedup over software in spite of it's slower clock and smaller bandwidth. Results are similar for the 2-dimensional kernel, Figure 3.5. The sequential data is comparable for both the FPGA, and the CPU results. The FPGA's runtime on random data is only about 2x faster than the CPU. Performance drops on both architectures because the FPGA can no longer share hardware for the B array, and the CPU loses the temporal locality of the B array.

3.5 Conclusion

In this section we presented the CHAT HLS compiler for irregular applications. Unlike the current HLS tools being developed by industry and research labs it focuses on large applications with poor temporal and spatial locality. The tool can generate custom hardware kernels which use a memory masking multithreaded model to efficiently utilize the FPGAs bandwidth regardless of the memory access pattern. Results showed around a 2x speedup over software on two irregular kernels. 2x is not usually a significant performance increase for FPGA applications which have shown speedups of 100x to 1,000x. However, our results are on memory bounded not compute bounded applications. We achieved our 2x speedup while yielding a 33% bandwidth advantage to software.

Chapter 4

A Multithreaded Sparse Matrix Mulitplication Kernel

Sparse Matrix Vector Multiplication (SpMV) is a memory bounded problem that often has a poor Flop/Byte performance. The operations are simple (multiplication, and addition) so throughput depends on how fast the Processing Engines can receive data. Even though the algorithm is simple computationally, it is a very important application to many fields in computer science (i.e. Scientific Computing, HPC, etc). In this chapter we explore how multihreading can be used to implement and improve this common operation. The first concern addressed is how the sparse matrix can be best store in memory to minimize the number of memory accesses. Then based on the matrix format a custom multithreaded memory masking kernel is generated by the CHAT tool. Finally, the hardware kernel's limits and scalability are explored in the results section. The performance is also compared with state of the art approaches in software, and GPU platforms.

4.1 A Multithreaded Sparse Matrix Kernel

We use the CHAT compiler to generate a multithreaded SpMV kernel. Our design is based around the Compressed Sparse Row (CSR) format from Section 2.2.1.2. Two design choices were important in selecting CSR. First it requires a minimal amount of memory while being able to support any matrix. ELLPACK uses less memory on some matrices, but is not a general purpose solution. Second CSR can be easily broken into independent threads.

Algorithm 5 CHAT SpMV kenrel source code.

```
void spmv_csr (int *row, int *val, int *col,
               int *vec, int *out, int lenght)
{
    int r, c, tmp;
    for (r = 0; r < lenth; ++r) {
        for (c = row[r]; c < row[r+1]; ++c)
            tmp = tmp + val[c] * vec[ col[c] ];
        out[r] = tmp;
    }
}
```

4.1.1 SpMV Kernel Code

Sample code for a SpMV kernel is shown in Algorithm 5. This is, line for line, the code used by the compiler. All arrays are treated as streams of data into the FPGA. Most (*row*, *val*, *col*) are accessed in a streaming (regular) fashion. However, the *vec* array is accessed by the *col* array therefore is treated as an irregular accesses. Thread workloads are determined by the two adjacent elements in the *row* stream. Threads are issued in order, but they are not required to have the same workload size. Thus the *out* array can write to memory out-of-order. The kernel writes whenever a thread finishes. The designer can unroll the outer for-loop to generate multiple PEs yielding higher parallelism.

4.1.2 Processing Element

The bulk of the SpMV's work is done by the processing element (PEs). These engines operate independently and the number of PEs is limited by the resources available on the FPGA (i.e. number of memory channels). Each thread assigned to a PE will generate one output, which is the sum-of-products for a row. Thread states must maintain the running sum, and the start/end positions for the memory requests. As requests are fulfilled the data is sent to a summation unit which produces the final sum-of-products. Each PE manages the requesting, multiplying, and summing for multiple threads (rows) concurrently.

The PE's component layout is shown in Figure 4.1. Each PE manages memory requests to the column, value, and vector arrays. Our implementation is for the Convey HC-2ex machine, which supports in-order memory requests. The physical accesses to memory are fulfilled out-of-order, but the HC-2ex uses a custom crossbar to reorder the data before returning it to the PE. Thread states can therefore be stored in FIFO buffers.

A PE will uses a busy flag to stall new threads from entering the datapath. Once a thread job is assigned to a PE the flag is asserted until all initial requests for

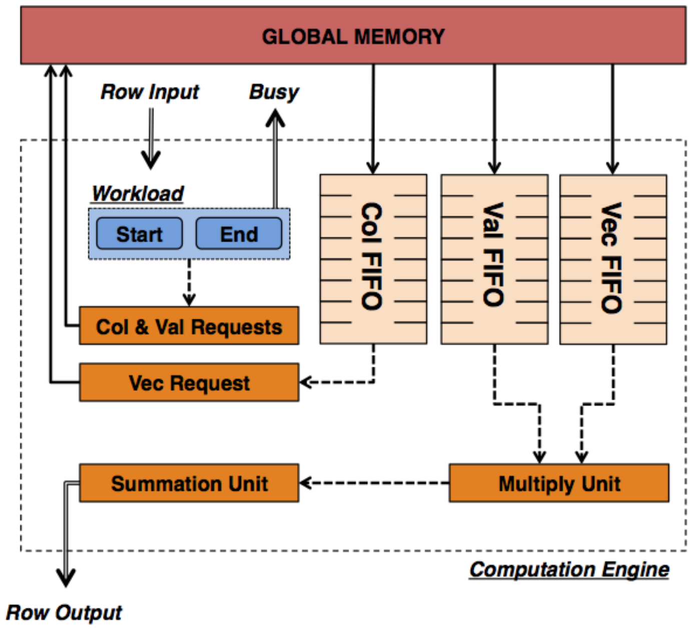


Figure 4.1: Each PE is assigned a thread. It requests the necessary data (Column, Vector, and Value) from global memory. Returned data values are pushed through the multiply pipeline, and summation unit.

data have been made. Because threads can be small they may not require enough data to fully mask memory. After all the initial requests have been made a new thread job will enter the PE, and multiple requests for different threads can be outstanding at any given time. Workloads are balanced across PEs because new jobs are not assigned until the PE is ready. A long job will prevent only one PE from getting new jobs while other PEs are handling many smaller jobs.

FIFO buffers within each PE are large enough to support all the outstanding requests. As memory returns the data for the column array it is used to generate the memory requests for the vector array. The data returned for the value array is held in buffer until the corresponding vector request is fulfilled. As data is returned from memory it is buffered in the Value and Vector FIFOs with its thread ID (row index). The summation unit uses the thread IDs to manage concurrent threads. Our compiler

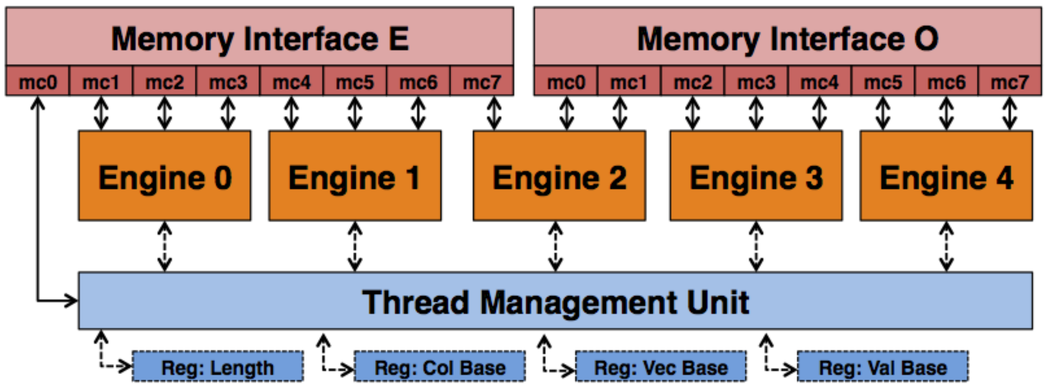


Figure 4.2: The MT-FPGA architecture on one AE. Control signals specify the number of jobs (length), and the base addresses of the sparse matrix arrays. All memory channels of the AE are utilized.

uses a circuit similar to the one described in [24]. It handles multiple rows concurrently, and can read a new element every cycle. The circuit assumes all data for one row enters the datapath before any data from another row enters. This assumption holds for our kernel because of the HC-2ex's in-order requests. This reduction circuit is only needed if the kernel is compiled for floating point operations because addition requires multiple cycles.

4.1.3 Thread Management

The Convey HC-2ex has 4 Virtex 6 LX760 FPGAs, called application engines (AEs). Figure 4.2 shows the SpMV kernel layout for one HC-2ex AE. The design can be replicated to all four AEs at runtime. Each AE has 16 memory channels, and each PE requires 3 memory channels; for the column, value, and vector requests. Memory channels are the designs bottleneck, which is limited to 5 PEs per AE. Control registers in the AE specify the number of threads (i.e. rows) needed by the SpMV kernel. They are also used to also specify the base addresses for each memory array used by the kernel. When using multiple AEs the values are partitioned to balance the workload.

One thread management unit (TMU) communicates with the 5 PEs. It creates thread workloads with values from the row pointer array. Access to the row pointer incurs the same memory latency as all other requests. The TMU buffers threads when all PEs are busy. Assignment is done dynamically in round robin fashion.

As PE threads complete the output value is buffered by the TMU until it can be written back to global memory. The TMU manages 5 *out* streams (one per PE) as well as the *row* stream. Reads and writes to these two streams are infrequent, occurring once per thread, compared to the column, value, and vector streams. The TMU uses one memory channel, and interleaves its read and write requests. Read and write request conflicts are resolved by a control unit in favor of the write data, which prevents a deadlock.

4.1.4 HC-2ex FPGA Implementation

Integrating a kernel into the HC-2ex requires all memory requests to communicate with Convey's memory interface. Designs are placed and routed varying the number of kernel PEs. Area utilization (including the wrapper) for a single AE is shown in Table 4.1. The design uses only one third of the available slices and BRAMs because it is limited by the memory channels. As discussed above each PE requires 3 channels, and the TMU interleaves its requests though the remaining channel.

Table 4.1: FPGA utilization when varying the number of PEs.

PE(s)	Slices (118,560)	BRAMs (720)
1	25,788 (21%)	107 (14%)
2	29,040 (24%)	133 (18%)
3	32,638 (27%)	179 (24%)
4	36,520 (30%)	209 (29%)
5	39,395 (33%)	239 (33%)

4.2 Experimental Evaluation

In this section we compare our MultiThreaded FPGA (MT-FPGA) design and compare it to modern CPU & GPU platforms. We analyze the memory utilization of different sparse matrix formats. Dense matrices are used to evaluate the HC-2ex's best sustainable performance. We also use real world benchmarks [67] to compare throughput performance of the architectures using the CSR format.

Table 4.2: An architecture specification for the various hardware used in SpMV comparisons.

Company	Platform	Type	Clock (MHz)	Cores	Memory Bandwidth
Convey	HC-2ex	FPGA	150	n/a	76.8 GB/s
Nvidia	Tesla C1060	GPU	1300	240	102.0 GB/s
Nvidia	Tesla K20	GPU	705	2496	208.0 GB/s
Intel	E5540	CPU	2530	4	25.6 GB/s

4.2.1 Experimental Setup

We use the Convey HC-2ex to measure the performance of our MT-FPGA approach. We compare the overall throughput to 3 different architectures; 2 GPUs, and one CPU. Table 4.2 shows the performance specifications for each architecture. The FPGA has the slowest clock frequency. It is 4.7x slower than the K20 GPU, and an order of magnitude slower than the other GPU and CPU. However, SpMV is a memory bounded application, and the clock frequency does not have as much affect on performance as the memory bandwidth. The FPGA has 3x more bandwidth than the CPU, but it is also 2.7x lower than the newest (K20) GPU.

We compare against the best general purpose CPU and GPU approaches we can find. We use the CUSP Library [8, 18], developed by researchers at NVidia, to test the GPUs' performance. We obtain GPU results on a Tesla architecture (GPU-Tesla) that was released in late 2008. The FPGA's Virtex-6 architecture was released in early 2009. The more recent GPU Kepler architecture (GPU-Kepler) is also used to gauge performance on the best hardware currently available. Software performance is measured on a Intel Xeon E5540 CPU using the Poski library [73] developed at the University of California, Berkeley.

Table 4.3: Sparse matrix dimensions, and their throughput (DP GFLOPS) on our hardware accelerators. All benchmarks are taken from the UF Sparse Matrix Collection [67].

Suite 1

Benchmark Name	Rows	Non-Zero	NNZ/Rows
dw8192	8,192	41,746	5
t2d_q9	9,801	87,025	9
epb1	14,734	95,053	6
raefsky1	3,242	294,276	91
psmigr_2	3,140	540,022	172
torso2	115,967	1033,473	9
Geo Mean			

Suite 2

Benchmark Name	Rows	Non-Zero	NNZ/Rows
Dense	2,000	4,000,000	2,000
Protein	36,417	4,344,765	119
FEM/Cantilever	62,451	4,007,383	64
FEM/Harbor	46,835	2,374,001	51
QCD	49,152	1,916,928	39
Economics	206,500	1,273,389	6
Epidemiology	525,825	2,100,225	4
FEM/Accelerator	121,192	2,624,331	22
Circuit	170,998	958,936	6
Webbase	1,000,005	3,105,536	3

Suite 3

Benchmark Name	Rows	Non-Zero	NNZ/Rows
cage15	5,154,859	99,199,551	19
circuit5M	5,558,326	59,524,291	11
Flan_1565	1,564,794	117,406,044	75
Serena	1,391,349	64,531,701	46

We use 20 benchmarks to evaluate our kernels performance. They were chosen based on published results from other research groups [73, 8, 49]. They are broken into three categories based on the number of non-zero elements. The small benchmarks range

Table 4.4: The memory storage requirement (MB) for each benchmark in the three sparse matrix formats. ELL pads all smaller rows to the length of the longest. When this is unreasonably long (> 10 GB) we mark its size as N/A.

Suite 1

Benchmark Name	CSR	ELL	HYB
dw8192	0.7	1.0	0.7
t2d_q9	1.5	1.4	1.4
epb1	1.6	1.7	1.7
raefsky1	4.7	5.6	7.1
psmigr_2	8.7	115.0	13.0
torso2	17.5	18.6	16.7
AVERAGE	5.8	23.9	6.7

Suite 2

Dense	64.0	64.0	96.0
Protein	69.8	118.0	84.6
FEM/Cantilever	64.6	77.9	75.3
FEM/Harbor	38.4	108.0	51.9
QCD	31.1	30.7	30.7
Economics	22.0	145.0	28.9
Epidemiology	37.8	33.7	33.7
FEM/Accelerator	43.0	46.5	55.5
Circuit	16.7	965.0	18.7
Webbase	57.7	N/A	58.7
AVERAGE	74.0	196.0	88.3

Suite 3

cage15	1,620.0	3,870.0	1,920.0
circuit5M	996.0	N/A	1,200.0
Flan_1565	1,890.0	N/A	2,020.0
Serena	1,040.0	4,470.0	1,190.0
AVERAGE	1,380.0	N/A	1,580.0

from a few thousand non-zero elements up to 1-million non-zero elements. The medium benchmarks range from 1-million non-zero elements up to 5-million non-zero elements. The large benchmarks all have 60-million non-zero elements, or more. We obtain all our benchmarks from the University of Florida Sparse Matrix Collection [67]

4.2.2 Memory Footprint of Sparse Matrix Storage Formats

Throughput on the MT-FPGA design is contingent only on how much data has to be streamed to the FPGA. This differs from modern CPU and GPU architectures, which require effective caching to maintain a high throughput. Memory masking is effective regardless of the latency (assuming a large enough dataset), or any number of cache misses. Therefore, the MT-FPGA kernel's priority is to use a sparse matrix format with the smallest memory footprint.

In Table 4.4 we compare the memory footprint of common sparse matrix formats. Intuitively we know CSR will always be less than COO; except in the special case where there is only one element per row. They have the same three arrays, but CSR does compression on the row. ELL, and HYB are less clear. ELL uses 2 matrices, and assumes the rows size to be static. If the sparse matrix is uniform then this format could require less memory than CSR. HYB is a combination of ELL and COO, which makes its footprint difficult to estimate as well. However, the empirical results from Table 4.4 show that CSR is usually the smallest.

4.2.3 Dense Matrix Experiments

The MT-FPGA approach copes with long memory latencies by issuing many outstanding requests, and maintaining all the thread states locally. In the early stages of execution performance is poor because the FPGA does not have enough concurrency. As the datasets grow larger these startup costs are amortized over the entire execution. Eventually the TMU will buffer enough, and the PEs will be processing enough threads so the latency is not an issue.

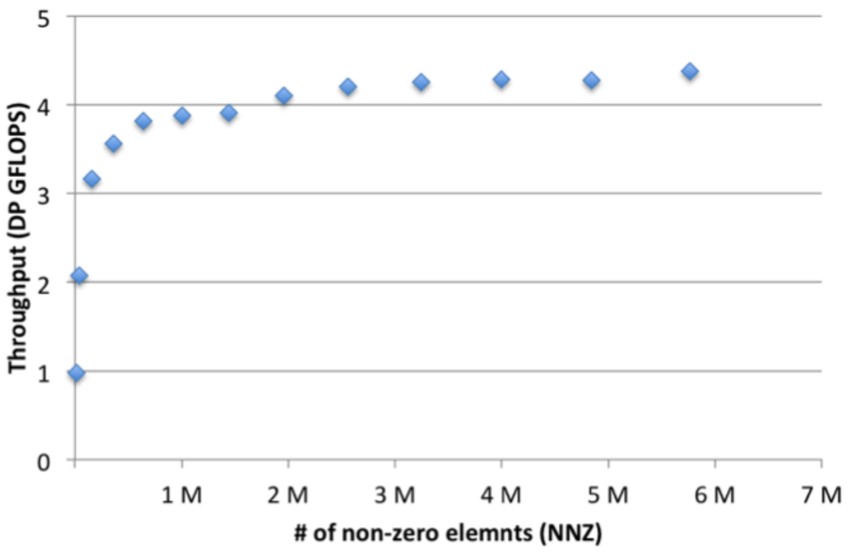


Figure 4.3: The sustained SpMV throughput as the matrix sizes increase. The matricies are dense, but stored in CSR format. The Convey HC-2ex is fully utilized with 20 PEs.

We run a set of test on the Convey HC-2ex to determine what dataset sizes are "large enough" to amortize the startup costs. Results are shown in Figure 4.3. The experiment uses a dense matrix, but it is stored in CSR format. We intentionally do this to remove any irregularity. In doing so we can see how startup costs affect the throughput. The experiment also uses 20 PEs spread across all 4 FPGAs. This fully utilizes the HC-2ex's memory channels and fully saturates the memory. In Figure 4.3 we can see that the sustained throughput quickly approaches 4 DP GFLOPS as the number of non-zero elements approaches 1 Million. From there slowly rises to a peak sustained throughput of 4.5 DP GFLOPS. This is 75% of the peak (6 DP GFLOPS) possible throughput. We can see that the MT-FPGA, on the Convey HC-2ex, is only a reasonable alternative when the sparse matrix has over 1-million non-zero elements. Other architectures with shorter latencies could support smaller matrices.

We were also interested in how well the memory system would cope as the number of utilized memory channels increased. For this experiment we again used a

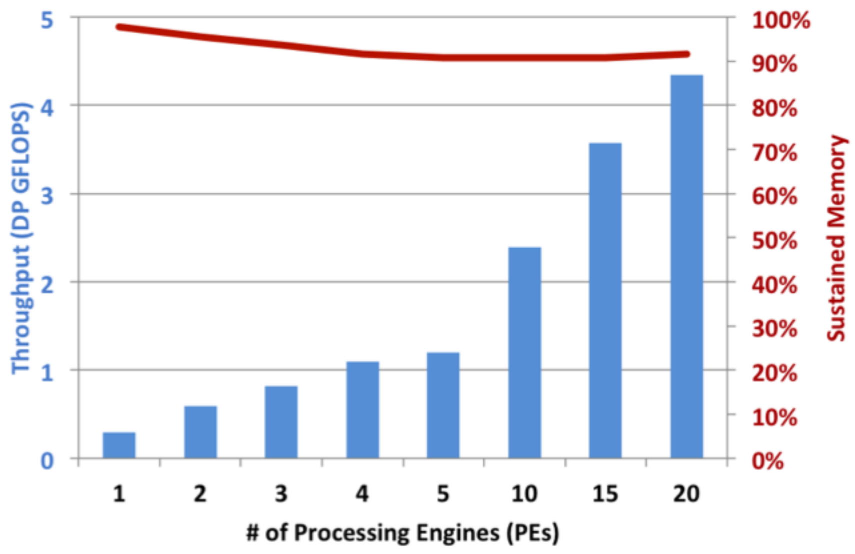


Figure 4.4: The sustained memory bandwidth, and throughput as the number of PEs increases. All test use a dense 4 million non-zero element dense matrix stored in CSR format.

dense matrix stored in CSR format. All runs use a 4 million element (2,000 x 2,000) matrix. Results are shown in Figure 4.4. We start with a single AE, and increase the number of engines from 1 (3 channels) up to 5 (15 channels) on a single FPGA. Then we increase the number of AEs used from 1 (15 channels) to 4 (60 channels). The blue bar-graph shows how the throughput scales, and the red line shows the percentage of time memory issued no stalls. These results are HC-2ex specific, and they show performance is limited by the memory architecture not the FPGA design.

4.2.4 Throughput Comparison with CSR

Compressed Sparse Row (CSR) is the most common format for sparse matrix operations. In this section we compare how the MT-FPGA approach compares to CPU, and GPU approaches. The FPGA uses a multithreaded approach as described in the previous sections. The CPU uses the Poski library developed at the University of Cali-

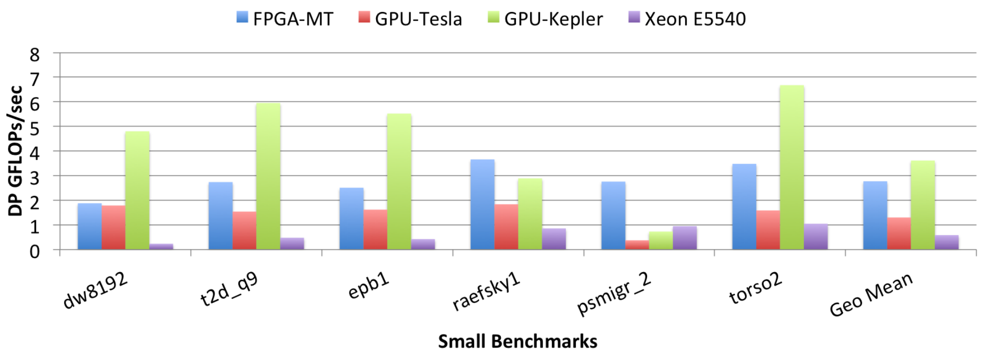


Figure 4.5: FPGA, CPU, and GPU throughput performance on suite 1.

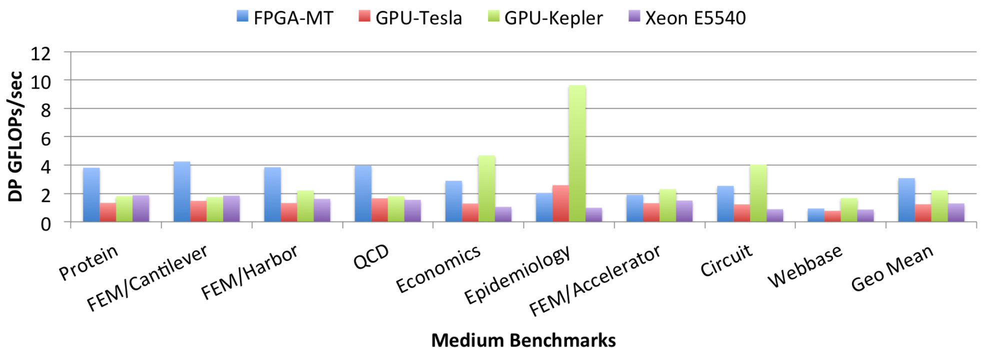


Figure 4.6: FPGA, CPU, and GPU throughput performance on suite 2.

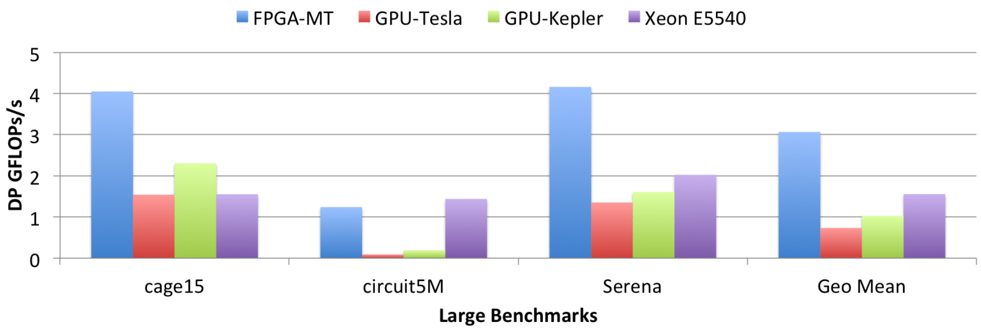


Figure 4.7: FPGA, CPU, and GPU throughput performance on suite 3.

fornia, Berkeley [73]. Both GPUs use the CUSP library developed by NVidia [8, 18]. As mentioned in Section 4.2.1 we classify our datasets by the number of non-zero elements, and the results are reported separately.

Figure 4.5 show the different architecture's performance on small data sizes. Because each matrix in this dataset is below or about 1 million non-zero elements we do not expect the FPGA to sustain it's peak performance. The startup costs will dominate the computation. In addition the matrices are small enough that most data can fit on the CPU cache. Regardless the MT-FPGA still outperforms the CPU, and the Tesla GPU. However, the newer Kepler GPU does outperform the FPGA in most cases. This is attributed, in part, to it's significantly higher memory bandwidth. The GPU has an almost 3x advantage with 208 GB/s compared to the FPGA's 76.8 GB/s. Designs looking at small matrices should look to the GPU, or other non-multithreaded FPGA designs.

Figure 4.6 shows the different architecture's performance on medium data sizes. Here we can see the FPGA's consistency. For the majority of tests it's throughput is around 4 DP GFLOPs/sec. However, it does drop when the NNZ per row is small ($<$ 10). This happens because each PE has a thread startup cost. It takes a few cycles to initialize the counters, and ID management. During this time the PE does not issue new requests. Regardless, in the average case the FPGA still outperforms all other architectures.

Figure 4.7 shows the different architecture's performance on large data sizes. Here the FPGA performs substantially better than the GPUs and CPUs. The FPGA has a mean performance of 3.06 DP GFLOPS while the Kepler GPU only achieved 1.03. Again the FPGA performance is consistent at 4 DP GFLOPS except on Circuit5M where the NNZ/row is 11.

These results show one major benefit of the multithreaded approach, and that is it's consistency. While the GPU performs very well on some rums it also performs very poorly on others. Assuming the run is "large enough" the FPGA performs around 3.5 to 4.5 DP GFLOPs/sec consistently.

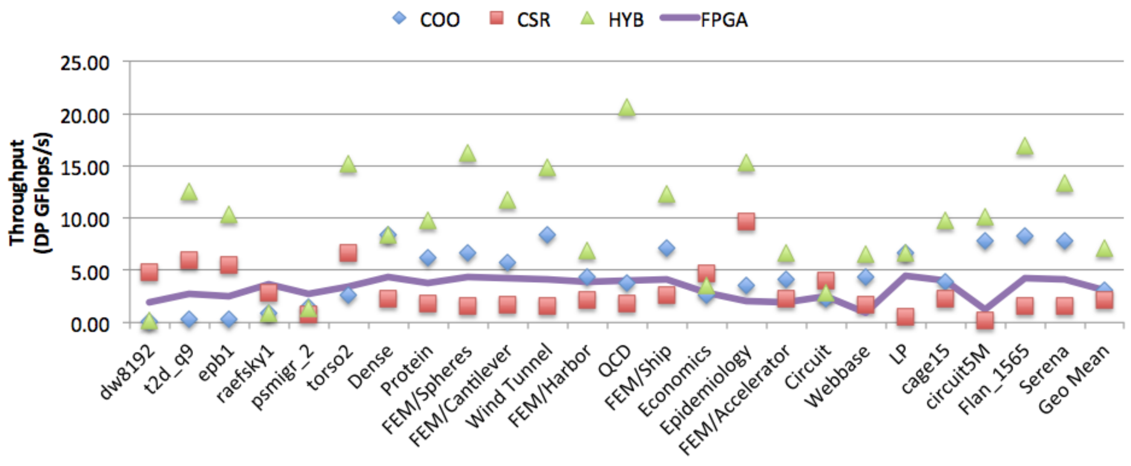


Figure 4.8: FPGA vs Kepler GPU throughput performance. The GPU uses different matrix formats to improve performance.

4.2.5 FPGA vs. GPU Throughput with different Sparse Matrix Formats

GPUs are custom purpose architectures, which excel at SIMD parallelism. Performance depends on each processor issuing the same instruction per cycle, which is hard to guarantee when CSR can have irregularity in the thread sizes. A format like ELLPACK, where all threads are zero padded to the same length, is better suited to GPU architectures. However, as mentioned in Section 2.2.1.3 this format can be inefficient on some matrices, and is not a general purpose solution. A modified format using both ELLPACK, and COO [8] has been shown to outperform CSR on the GPU. In Figure 4.8 we show the GPU-Kepler's performance, using 3 storage formats, to the

MT-FPGA using CSR. The GPU shows higher overall throughput, but that is to be expected because it also has 2.7x the FPGA's bandwidth.

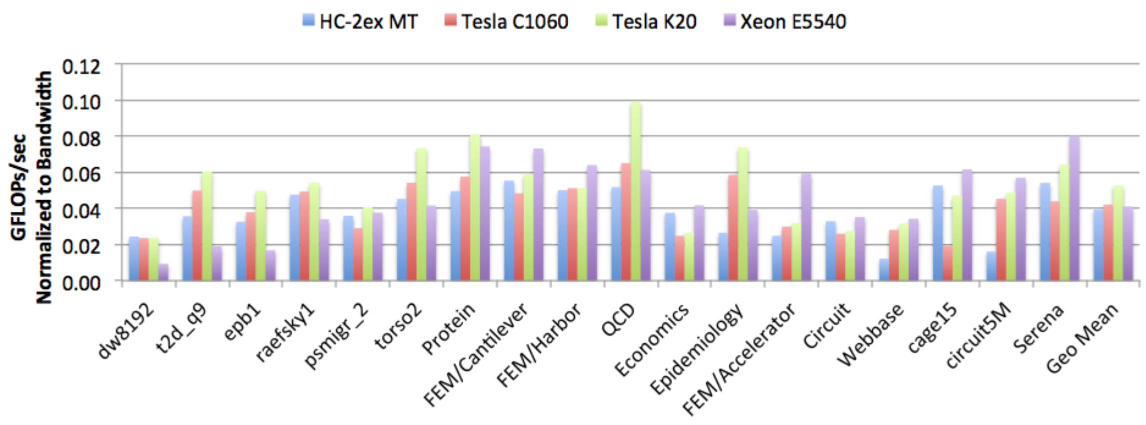


Figure 4.9: Throughput performance for each architecture normalized to the available bandwidth.

4.2.6 Normalized Throughput

It is difficult the guage the performance of different platforms simply because they are so different. They vary drastically in clock frequency, and memory bandwidth. They also require signifcanlty different appraoches to achieve peak performance. The CPU must optimize for its caches. The GPU must efficently interlace its memory request. The FPGA must sustain thousands of outstanding requests. In Figure 4.9 we normalize each platform's best performance to its available bandwidth. We do this because SpMV is a memory bounded problem, and its performance is dominated by how quickly the processing elements can receive data. In doing so we see that on average all platforms perform equally well. However, this is a result of the matrix benchmarks used as we will elaborate upon in the next section.

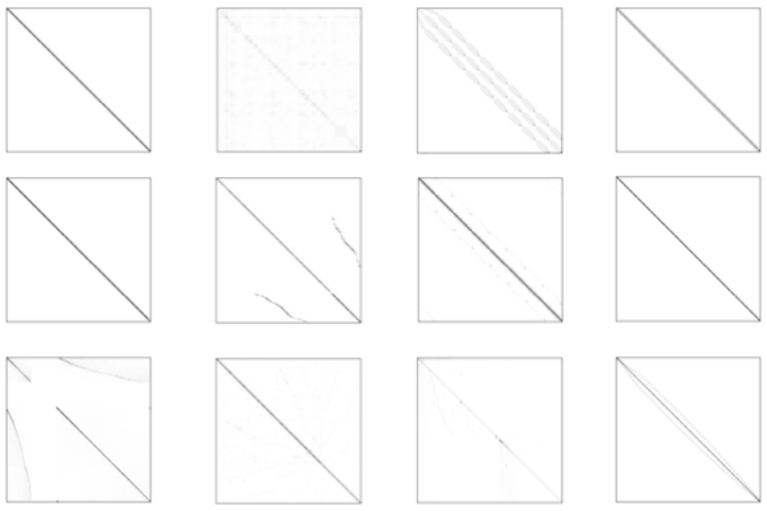


Figure 4.10: Visual printout of the sparse matrices used to benchmark Section 4.2.4 results.

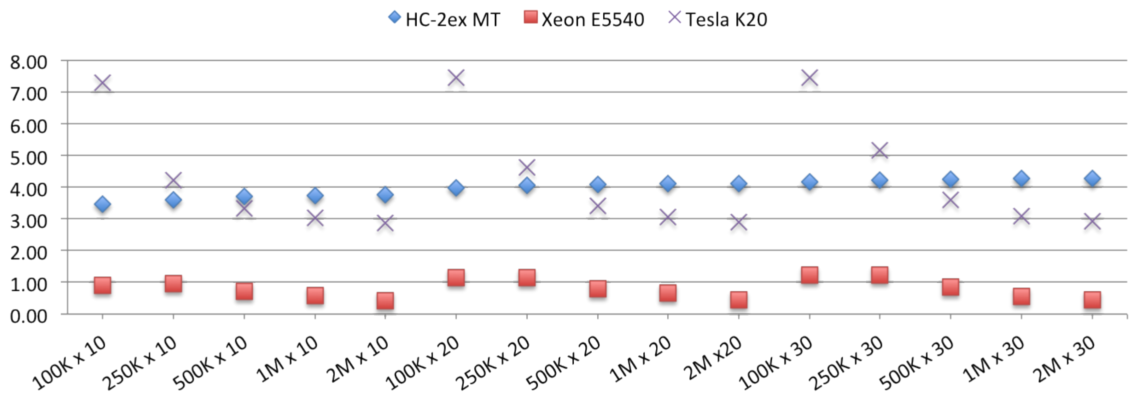


Figure 4.11: FPGA, CPU, and GPU throughput performance on truly irregular data.

4.2.7 Throughput on Truly Irregular Data

The benchmarks used in Section 4.2.4 were chosen from publication done by other lab groups. Taking a closer look at the types of sparse matrices chosen we see a pattern emerge. Most all of them have a diagonal where the majority of non-zero elements are, and a few elements sparsely distributed. Figure 4.10 shows a printout of these matrices. Looking at them row by row we can see that a lot of data will be shared, which is very beneficial for the cache dependent CPU designs.

In this section we consider how performance is affected if a user had truly irregular data. We generate 3 sets of matrices. One set has 10 non-zero elements per row, the next set has 20 non-zero elements, and the last set has 30-non zero elements per row. The elements are distributed randomly over the entire row. 5 matrices are generated for each set with a total number of elements between 100 thousand and 2 million non-zero elements. Throughput results are reported in Figure 4.11.

We see a drastic performance difference between the architectures. Most noticeably the GPU and CPU performance drops as the number of non-zero elements increases within sets. It is more noticeable on the GPU because it has much higher bandwidth. This occurs because the smaller matrices have fewer rows, and fewer rows means the vector being multiplied against can better fit in the caches. However, as the matrices get larger so to do the vectors, and the caches lose their effectiveness. By contrast the multithreaded approach is unaffected by any differences in the matrices. The FPGA performance is always around 4 DP GFLOPs/sec. There is a slight performance increase (for the FPGA) as the number of elements per row increases. This occurs because the PEs have a small startup cost for new threads, and it is better amortized with 30 elements per thread than it is with 10.

4.3 Conclusion

In this chapter we showed how multithreading can be beneficial to a real world application like Sparse Matrix Vector Multiplication. Our MT-FPGA implementation performed as well as modern GPU and CPU platforms on commonly used matrix benchmarks. However, on truly irregular data we showed that the multithreading approach can significantly improve performance over the same CPU and GPU platforms. As matrix size increases the CPU and GPU performance drops drastically while the FPGA performance is consistent independent of the size. It is this consistency and predictability that make multithreading a enticing alternative to the current cache based platforms.

Chapter 5

FPGA based Multithreading for In-Memory Hash Joins

Large relational databases often rely on fast join implementations for good performance. Recent paradigm shifts in processor architectures has reinvigorated research into how the join operation can be implemented. The FPGA community has also been developing new architectures with the potential to push performance even further. Hashing is a common method used to implement joins, but its poor spatial locality can hinder performance on processor architectures. Multithreaded architectures can better cope with poor spatial locality by masking memory/cache latencies with many outstanding requests.

5.1 Related Work

Many recent works consider the in-memory implementation of joins (hash or sort-merge). [41] was the first work, which emphasized the importance of TLB misses in partitioned hash joins and proposed a *radix clustering* algorithm to keep the partitions cache resident. Later [9] studied the performance of hash joins by comparing simple hardware-oblivious algorithms and *hardware-conscious* approaches (since the radix clustering algorithm is tightly tailored to the underlying hardware architecture). Results showed that the simple implementations surpass approaches based on radix clustering. However recently, [6] applied a number of optimizations and found that hardware-conscious solutions in most cases are prevalent over hardware-oblivious.

The implementation of sort-merge joins on modern CPUs was studied in [34], which explored the use of SIMD operations for sort-merge joins and hypothesized that its performance will surpass the hash join performance, given wider SIMD registers. Subsequently [1] implemented a NUMA-aware sort-merge algorithm that scaled almost linearly with the number of computing cores. This algorithm did not use any SIMD parallelism, but it was reported to be already faster than its hash join counterparts. Recently, [5] reconsidered the issue and found that hash joins still have an edge over sort-merge implementations even with the latest advance in width of SIMD registers and NUMA-aware algorithms.

While the software community has examined both hash and sort-merge joins the FPGA community has concentrated on sort-merge approaches. The reasons for this are twofold. Firstly, sorting and merging implementations are straightforward for parallel FPGA architectures. For example, sorting networks like bitonic-merge [32] and odd-even sort [37] are well established designs for FPGAs; [14] developed a multi-FPGA sort-merge algorithm, while [55, 75] used sort-merge as part of a hardware database processing system. Secondly, efficiently building an in-memory hash table is non-trivial because of the required synchronization.

Commercial platforms like IBM's Netezza[31] and Teradata's Kickfire[65] offer FPGA solutions for Database Management Systems. They cover a full range of database operations from selection and projection, to joins and aggregation. However, because of their proprietary nature specific implementation details, and measurements are difficult to obtain. Some patent information is available [12, 29, 43], but it is difficult to determine, specifically, how operations are handled with the available literature. By contrast the scope of this work is much narrower. We look at how FPGAs can be used to improve only the join operation, which has been historically a time intensive operation.

5.2 Proposed Approach

In this section an implementation outline is presented for the build phase and probe phase processing engines used for the multithreaded hash join algorithm. Then a discussion about how existing research can be applied to this work, and how it could potentially improve performance further.

When building, and probing the hash table, all writes occur during the build phase while the probe phase only reads the hash table. Because of this separation the algorithm's hash table interactions are simplified, for both the CPU and FPGA, compared to other algorithms using hash tables (i.e. aggregation, duplicate elimination).

5.2.1 Build Phase Engine

Our target datasets are too large to keep in local FPGA BRAMs. Therefore, our design trades off small and fast on-chip memory for larger and slower off-chip memory. The build engine copes with the long memory latencies by issuing thousands of threads and maintaining their states locally on the FGPA. Because of the inherent FPGA parallelism, multiple threads can be activated during the same cycle while other threads are issuing memory requests and going idle.

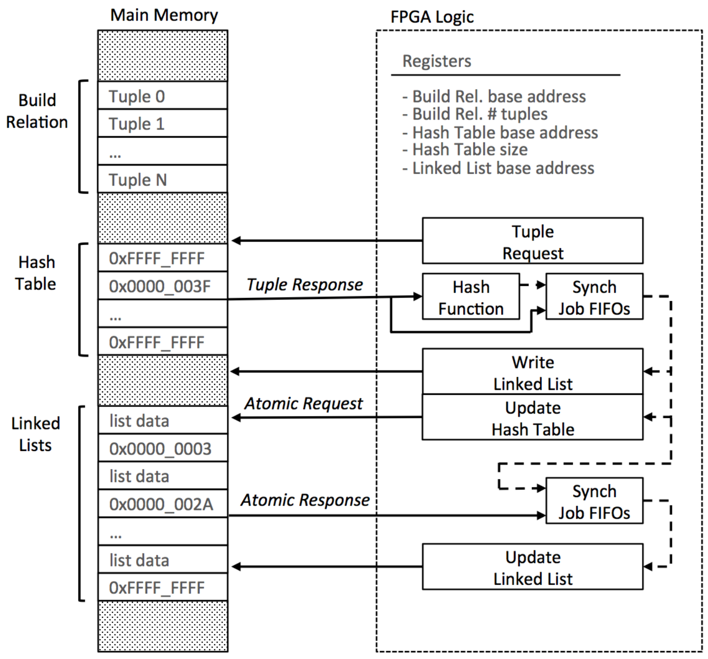


Figure 5.1: The FPGA Build Phase Engine.

The entire build relation along with the hash table and the linked lists are stored in main memory (Figure 5.1). Our hash table uses the chaining collision resolution technique: all elements hashed to the same bucket are connected in a linked list, and the hash table holds a pointer to the list's head. A unique value (0xFFF...FFF) is used to represent empty buckets in the hash table.

Figure 5.1 also shows how the build engine (FPGA logic) makes requests to the main memory data structures using 4 channels. In the FPGA logic, local registers are programed at runtime and hold pointers to the relation, hash table, and linked lists. They also hold information about the number of tuples, the tuple sizes, and the join key position in the tuple. Lastly, the registers hold the hash table size, which is used to mask off results from the hash function. The *Tuple Request* component will create a thread for each tuple and issues a request for its join key. The design assumes the join key size is between 1 and 8 bytes, and it is set at runtime with a register. The tuple can be of arbitrary size. If the key is split between two memory locations the *Tuple Request* component will issue both requests, and merge the responses. Requests are continually issued until all tuples have been processed, or the memory architecture stalls. When a thread issues a request the tuple's pointer is added to the thread state, and the thread goes idle.

As join key requests are completed, the thread is activated, and the key along with its hash value are stored in the thread's state. The *Write Linked List* component writes the key and tuple pointer to a new node into the appropriate bucket linked list. The *Update Hash Table* component issues an atomic request to read, and update the hash table. The old bucket head pointer is read while the new node pointer replaces it. An atomic request is needed here because a single engine can have hundreds of threads in flight, and issuing separate reads and writes would create race conditions. While the atomic request is issued the new node pointer is added to the thread's state.

As the atomic requests are fulfilled the thread is again activated, and the *Update Linked List* component updates the bucket chain pointer. If no previous nodes hashed to that location then the atomic request will return the empty bucket value, which is used to signify the end of a list chain. Otherwise, the old head pointer is used to extend the list.

5.2.2 Probe Phase Engine

The probe engine also assumes that all data structures are stored in main memory. Like the build engine it has to use memory masking to cope with high memory latency and maintain peak performance. Because no data is stored locally, the same FPGA used for the build engine can be reprogrammed with the probe engine (which

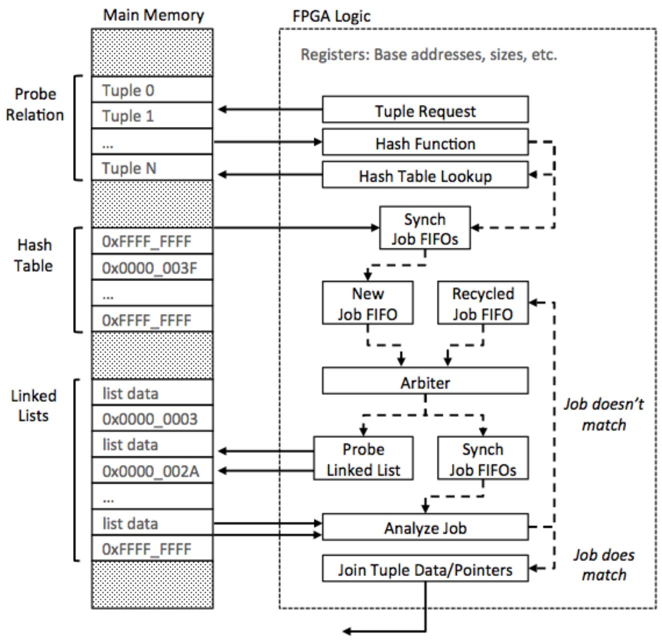


Figure 5.2: The FPGA Probe Phase Engine.

can be useful in the case of a small FPGA). Larger FPGAs can hold both engines and switch state depending on the required computation.

Figure 5.2 shows how the probe engine makes requests to the data structures in main memory (using 4 channels). Issuing threads, tuple requests, and hashing are handled the same way as in build engine. Again, the join key and the tuple's pointer are stored in the thread's state. Because the probe phase only reads data structures, there is no need for atomic operations. The thread only looks up the proper head pointer by hashed value from the table. The value (0xFFF...FFF) is again used to identify empty table buckets; if this value is returned then the probe tuple cannot have a match and is dropped from the FPGA datapath. Otherwise, the thread is sent to the *New Job FIFO*.

During the probe phase each node in a bucket chain must be checked for matches. A thread is not aware of the bucket chain length without iterating through the whole chain. Therefore, threads are recycled within the datapath until they reach the last node in the chain. The *Probe Linked List* component takes an active thread and requests its list node. Two channels are devoted to this component because it issues the bulk of read requests, and its performance is vital to the engine's throughput.

After the node is returned from memory the *Analyze Job* component determines if there was a match. Matching tuples are sent to the *Join Tuple* component. If a node is the last in the bucket chain then its thread is dropped from the datapath. Otherwise, its next node pointer is updated in the thread's state and is sent to the *Recycled Job FIFO*. The datapath can be improved to drop threads once a match is found, but this is only possible if the build relation's join key is unique. An *Arbiter* component is used to decide the next active thread, which will be sent to the *Probe Linked List* component. Priority is given to the recycled threads, thus reducing the number of concurrent jobs and ensuring that the design will not deadlock. Otherwise, when the recycled job FIFO fills, its back pressure would stall the memory responses, causing the memory requests to stall, thus preventing the arbiter from issuing a new job. As matches are found, the *Join Tuple* component merges the probe tuple's pointer (from the thread) with the build tuple's pointer (from the node list) and sends the result out of the engine.

5.2.3 Possible Optimizations

In practical workloads, joins are typically combined with selections and projections, in an effort to minimize intermediate result sizes (e.g., push selections and projections close to the relation). This approach can also be used here to further improve performance.

Predicate evaluation could filter out tuples, and alleviate memory utilization by creating gaps in the FPGA datapath. This could improve the build phase performance because it removes some of the costly atomic operations. The gaps could also mitigate back-pressure in the probe phase caused by long node chains. By adding the selection hardware on the FPGA, the latency will increase but because it is fully pipelined [63] it would not decrease the throughput.

Projection and the join step (i.e., using the tuple pointers to actually create the joined result) are ideal candidates for FPGA acceleration. Both are naturally parallel and streamable. Many works have leveraged these operations to improve performance [66, 55, 28]. In the special case where an entire tuple fits in one memory word the probe engine presented in this work can be easily extended to perform the join step. The engine already joins the pointers, but a little modification can replace them with the values instead. In order to capture the real effect of FPGA multithreading in the join operation, our implementation does not consider the selection, projection and join step.

Another common optimization applied to multi-core hash joins is partitioning, which eliminates the costly thread synchronization and allows to keep partitioned tuples cache resident [59]. However our FPGA engines cannot abandon synchronization completely. Even with partitioned data, each engine still has hundreds of outstanding read and write requests. Since all these requests are processed in a pipelined manner the only way to avoid race conditions will be to use atomic operations or some form of locking. Moreover our approach does not cache results on the FPGA BRAM, hence decreasing the number of tuples processed by each thread via partitioning will not have any effect on FPGA performance.

5.3 Experimental Results

We proceed with a description of the target architecture, the Convey-MX, and discuss how engines can be duplicated to match the available memory bandwidth. The

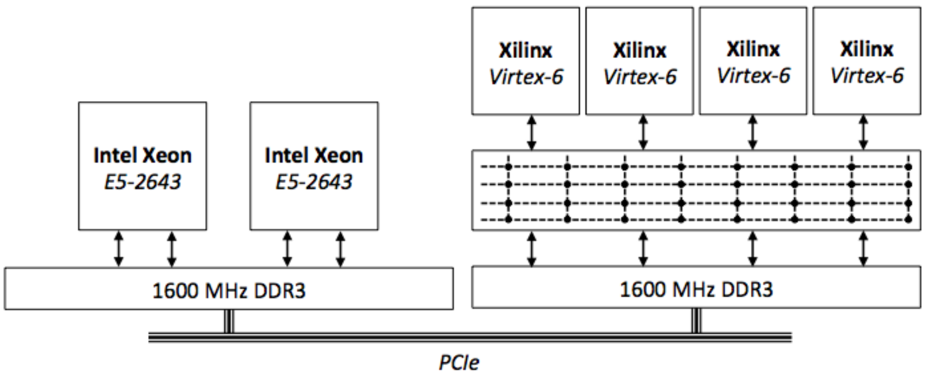


Figure 5.3: The Convey MX software and hardware regions.

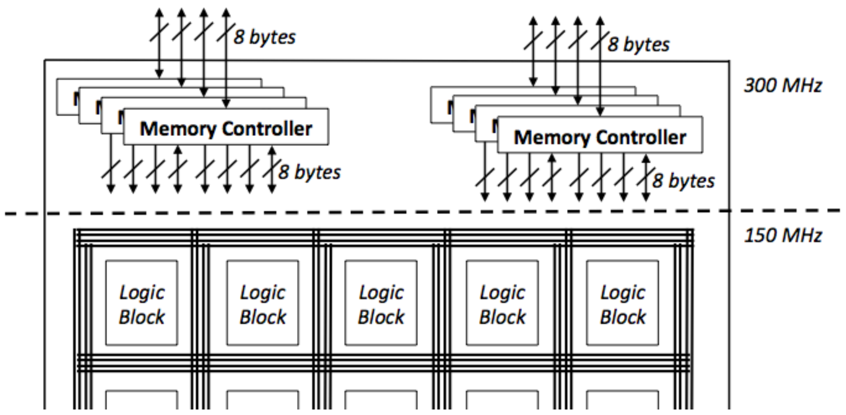


Figure 5.4: Each Convey MX FPGA AE has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells.

FPGA hash join implementation is compared in terms of overall throughput against the best multi-core approach [6]. We match the FPGA's and CPU's memory bandwidth as best we can (38.4 GB/s for the FPGA vs 51.2 GB/s for the CPU) to give the best comparison possible. We also present experiments on the scalability of the FPGA designs and their space utilization. Synthesizing FPGAs is well known to be a time intensive task; nevertheless, all designs presented here are capable of processing different join queries **without** needing to re-synthesize the FPGA logic.

5.3.1 Convey-MX Platform

The Convey-MX is a heterogeneous platform with a global memory shared between the CPUs and the FPGAs, allowing us to directly compare hardware and software in-memory hash join applications on the same memory architecture. Figure 5.3 depicts the MX's memory architecture. It has two regions (the software and hardware) connected though a PCIe bus. Each processor (CPU or FPGA) can access data from both regions, but data accesses across PCIe are significantly longer.

The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache. The multi-socket architecture treats each processor with the memory, attached to it, as a separate NUMA node. The NUMA asymmetry coefficient of described architecture is equal to 2.0. In total the software region has 128 GB of 1600 MHz DDR3 memory. Each NUMA node has a peak memory bandwidth of 51.2 GB/s.

The hardware region has 4 Xilinx Virtex-6 760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300 MHz (Figure 5.4). The FPGA logic cells run in a separate 150 MHz clock domain to ease timing and are connected to the memory controllers through 16 channels. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s. The MX memory also has locking bits at every word block allowing the FPGA to handle synchronization and atomic operations.

5.3.2 FPGA & Software Implementations

We choose to implement our FPGA designs on a Convey-MX platform, but the designs themselves are platform independent. The only requirement needed by the FPGA platform is in-order responses to memory requests. Given this assumption the

Probe Engine can be easily ported. The *Build Engine* requires some form of atomic operations. We choose the Convey-MX because it is the only FPGA platform we know of with direct support for atomic operations. However, with additional effort to enforce synchronization a Maxeler [42], Pico [53], or even a Convey-HC [16] board could be used.

Additional requirements are needed for the FPGA to achieve high throughput. First, it should have a high memory bandwidth. Second, it should handle multiple outstanding memory requests. The longer memory latency a platform has the more outstanding requests it will need be able to support. Peak performance will be achieved when memory latency can be fully masked.

Peak performance is dependent on the total number of concurrent engines, and the clock frequency. The number of engines is determined by the memory bandwidth. We next show how this applies to the Convey-MX, but the same could be done for other platforms. Sustained performance is dependent upon the memory architecture as shown in Section 5.3.4.

On the Convey-MX each FPGA has 16 individual memory channels which is more than what a single build or probe engine would need. To fully utilize the available bandwidth and increase parallelism, we duplicate the number of engines per FPGA. Since the build engine requires four channels (Section 5.2.1), four build engines can be stored on a single FPGA. Given that each FPGA runs at 150 MHz and, assuming no stalls, one could achieve a peak throughput of 600 MTuples/s per FPGA for the build phase. Similarly, the probe engine (Section 5.2.2) requires 4 channels, but also jointly uses a channel to write the output result to memory. Therefore only 3 probe engines could be placed on a FPGA. Assuming no stalling the FPGA can reach a peak throughput of 450 MTuples/s per FPGA for the probe phase.

As the state-of-the-art multi-core hash join approach we use the implementation from [6]. It includes 2 types of hash join algorithms: a hardware-oblivious non-partitioning join and a hardware-conscious algorithm, which performs preliminary partitioning of its input[1]. Both implementations perform the traditional hash join with build and probe phases, however they differ in the way multithreading is used. The non-partitioning approach performs the join using the hash table which is shared among all threads, therefore relying on hyper-threading to mask cache miss and thread synchronization latencies. The partitioning-based algorithm performs preliminary partitioning of the input data to avoid contention among executing threads. Later during the join operation each thread will process a single partition without explicit synchronization. The *Radix clustering* algorithm, which is a backbone of the partitioning stage needs to be parameterized with the number of TLB entries and cache sizes, making the approach hardware-conscious. In our experiments we use a two pass clustering and produce 2^{14} partitions, which yields the best cache residency for our CPU.

5.3.3 Dataset Description

Our experimental evaluation uses four datasets. Within each dataset we have a collection of build and probe relations ranging in size from 2^{20} to 2^{30} elements. Each dataset uses the same 8-byte wide tuple format, commonly used for performance evaluation of in-memory query engines [11]. The first 4 bytes hold the join key, while the rest is reserved for the tuple's payload. Since we are only interested in finding matches (rather than joining large tuples), our payload is a random 4-byte value. However, it could just as easily be a pointer to an actual arbitrarily long record, identified by the join key.

[1]All software experiment were implemented, and run by Ildar Absalyamov.

The first dataset, termed *Unique*, uses incrementally increasing keys which are randomly shuffled. It represents the case when the build relation has no duplicates, thus keys in the hash table are uniformly distributed with exactly one key per bucket (assuming simple modulo hashing). The next dataset (*Random*) uses random data drawn uniformly from a 32 bit int range. Keys are duplicated in less than 5% of the cases for all build relations having less then 2^{28} tuples. The largest relations have no more than 20% duplicates. For this dataset, bucket lists average 1.6 nodes when the hash table size matches the relation size, and 1.3 nodes when the hash table size is double the relation size. The longest node chains have about 10 elements regardless of the hash table size. To explore the performance on non-uniform input, the keys in the final two datasets are drawn from a Zipf distribution with coefficients 0.5 and 1.0 (*Zipf_0.5* and *Zipf_1.0* respectively); these datasets are generated using the algorithms described in [26]. In *Zipf_0.5* 44% of the keys are duplicated in the build relation. The bucket list chains have on average 1.8 keys regardless of the hash table size, while the largest chains can contain thousands of keys. In *Zipf_1.0* the build relations have between 78% and 85% of duplicates. Their bucket list chains have on average from 4.8 to 6.7 keys. The longest chains range from about 70 thousand keys in the relation with 2^{20} tuples to about 50 million in the 2^{30} relation.

5.3.4 Throughput evaluation

We report the multi-core results for both partition-based and non-partitioning algorithms. Results are obtained with a single Intel Xeon E5-2643 CPU, running on full load with 8 hardware threads. However because of the memory-bounded nature of

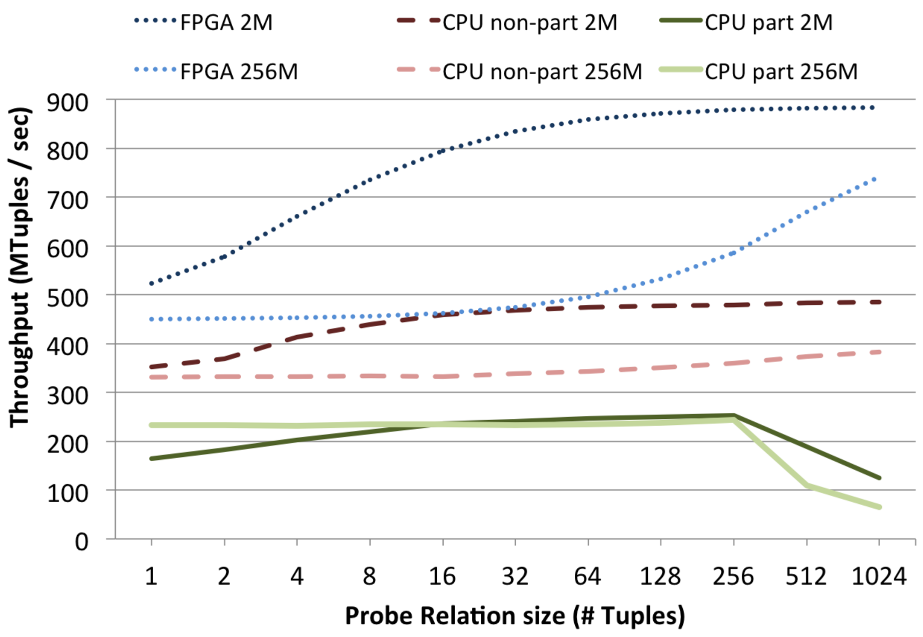


Figure 5.5: *Unique* dataset throughput as the build relation size is increased.

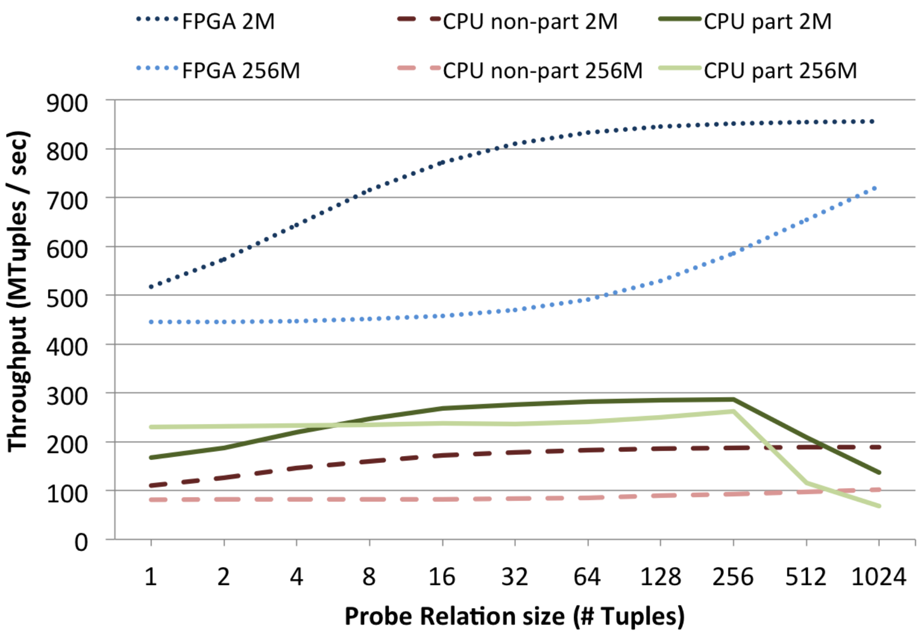


Figure 5.6: *Random* dataset throughput as the build relation size is increased.

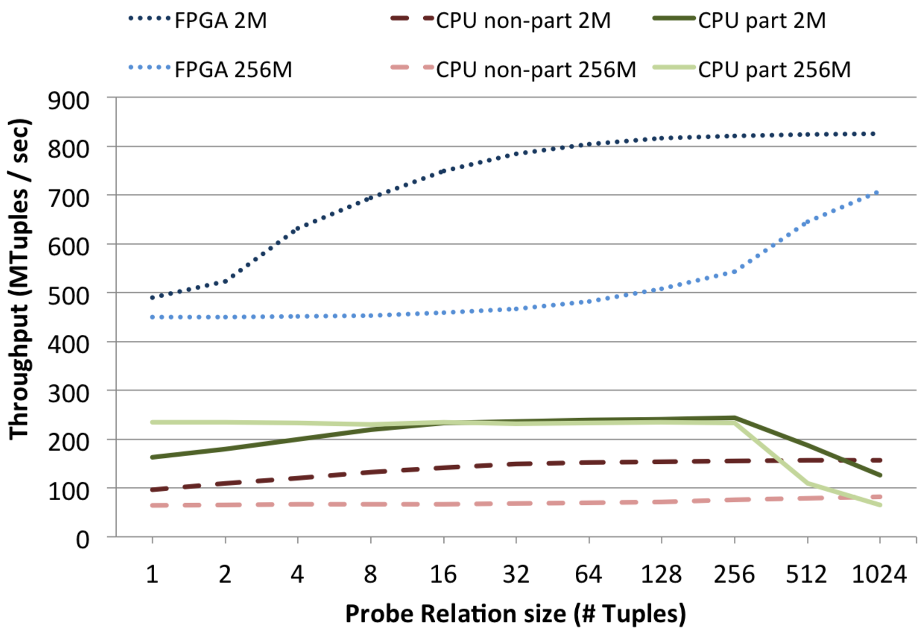


Figure 5.7: *Zipf_0.5* dataset throughput as the build relation size is increased.

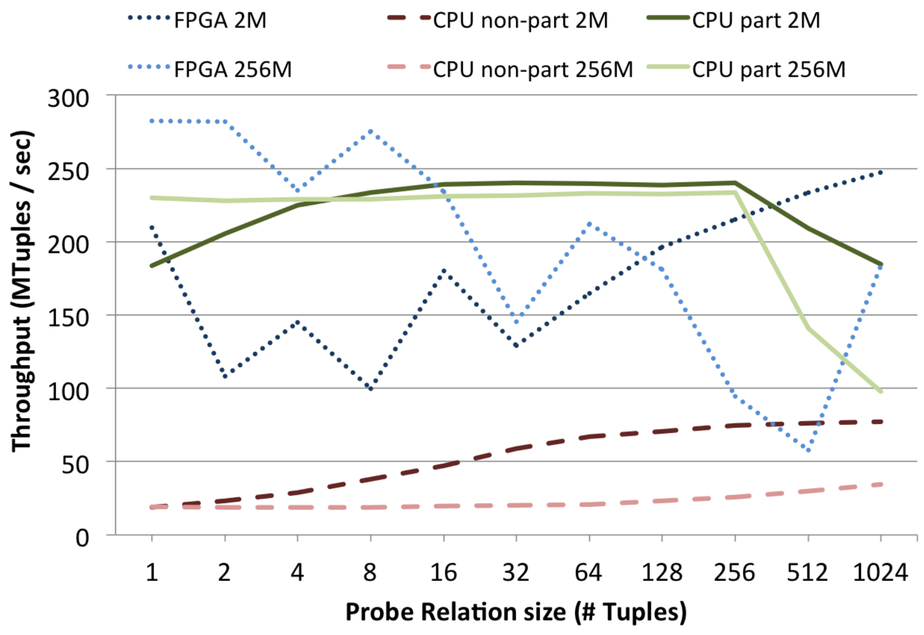


Figure 5.8: *Zipf_1.0* dataset throughput as the build relation size is increased.

hash join we use two FPGAs to offset the CPUs bandwidth advantage: a single CPU has 51.2 GB/s of memory bandwidth while two FPGAs have 38.4 GB/s (even with this bandwidth adjustment, the CPU still has almost a 30% advantage). By matching the bandwidth be can get a more accurate comparison between the approaches. Obviously, given of the parallel nature of hash join, the CPU and FPGA performance could easily be improved by adding more hardware resources.

Figures 5.5 to 5.8 shows the join throughput for two build relations, with 2^{21} and 2^{28} tuples respectively, while increasing the probe relation size from 2^{20} to 2^{30} for all datasets mentioned in Section 5.3.3. The FPGA performance shows two plateaus for the *Unique*, *Random* and *Zipf_0.5* data distributions on Figures 5.5, 5.6 and 5.7. The FPGA sustains throughput of 850 MTuples/s when the probe phase dominates the computation (that is, when the size of the probe relation is much larger than the size of the build relation) and it is close to the peak theoretical throughput of 900 MTuples/s which can be achieved with 8 engines on 2 FPGAs. When the build phase dominates the computation, atomic operations restrict FPGA throughput to about 450 Mtuples/s (in the FPGA 2^{28} plot, the throughput stays almost constant until the probe relation becomes comparable in size to the build relation). Clearly, in real-world applications the smaller relation should be used as the build relation. In the worst case we can expect FPGA throughput to be 600 MTuples/s when both relations are of the same size. For the extremely skewed dataset, *Zipf_1.0*, (shown in Figure 5.8) the FPGA throughput decreases significantly and varies widely depending on the specific data. This happens because extremely long bucket chains create a lot of stalling during the probe phase that greatly affects throughput.

The CPU results are consistent with those reported in [6]. The partitioned algorithm peak performance is around 250 MTuple/s across all datasets, regardless of whether computation is dominated by the build or the probe phase. It is also not affected by the data skew. For the non-partitioned algorithm, the throughput depends on the relative sizes of the relations, since like in the FPGA case, the throughput of the build phase is lower than the probe phase. The non-partitioned algorithm behaves always worse than the FPGA approach. Interestingly, for the *Unique* dataset, the non-partitioned version has better throughput than the partitioned one, because the bucket chain lengths are exactly one. As the average bucket chain length increases (moving from the *Unique* to the *Random* to the skewed datasets) the throughput of non-partitioned approach decreases. For the extremely skewed *Zipf_1.0* dataset, it falls approximately to 50 MTuples/s.

Averaging the data points within all datasets yields the following results: the FPGA shows a 2x speedup over the best CPU results (non-partitioned) on *Unique* data, and a 3.4x speedup over the best CPU results (partitioned) on *Random* and *Zipf_0.5* data. The FPGA shows a 1.2x slowdown compared to the best CPU results (partitioned) on *Zipf_1.0* data.

5.3.5 Scalability

To examine scalability, in the next experiments we attempt to match the bandwidth between software and hardware as closely as possible: every four CPU threads are compared to one FPGA (note that this still provides a slight advantage to the CPU in terms of memory bandwidth). We examine two cases, when the probe relation is much larger than the build one, and when they are of equal size. Figures 5.9, 5.11 and 5.13

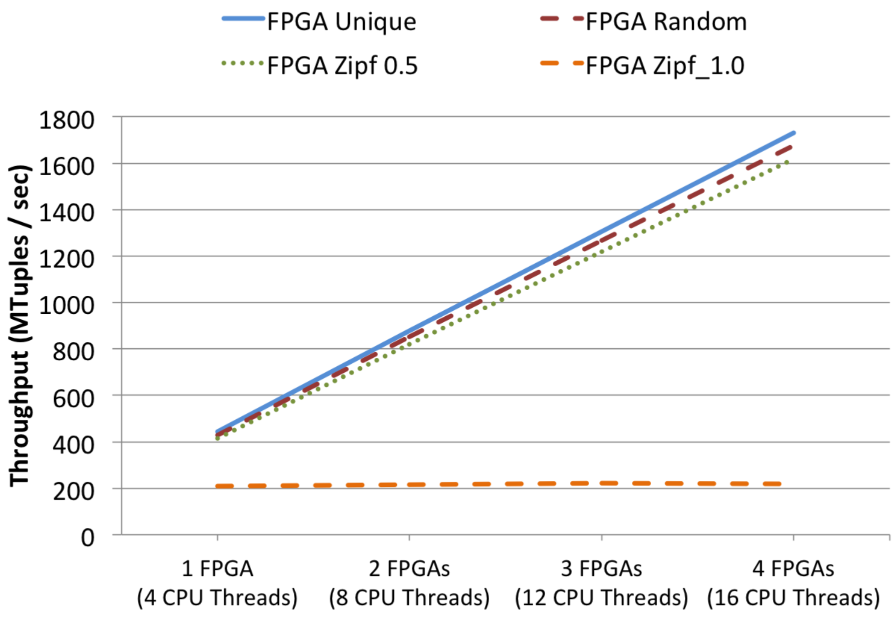


Figure 5.9: FPGA throughput scaling. The Build Relation has 2^{21}, Probe has 2^{28} tuples

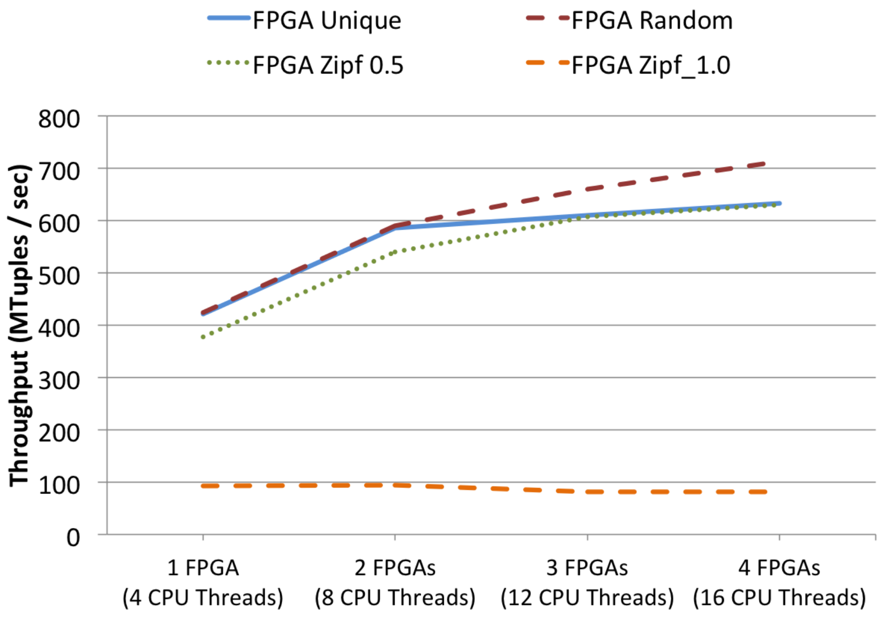


Figure 5.10: FPGA throughput scaling. The Build and Probe Relations both have 2^{28} tuples

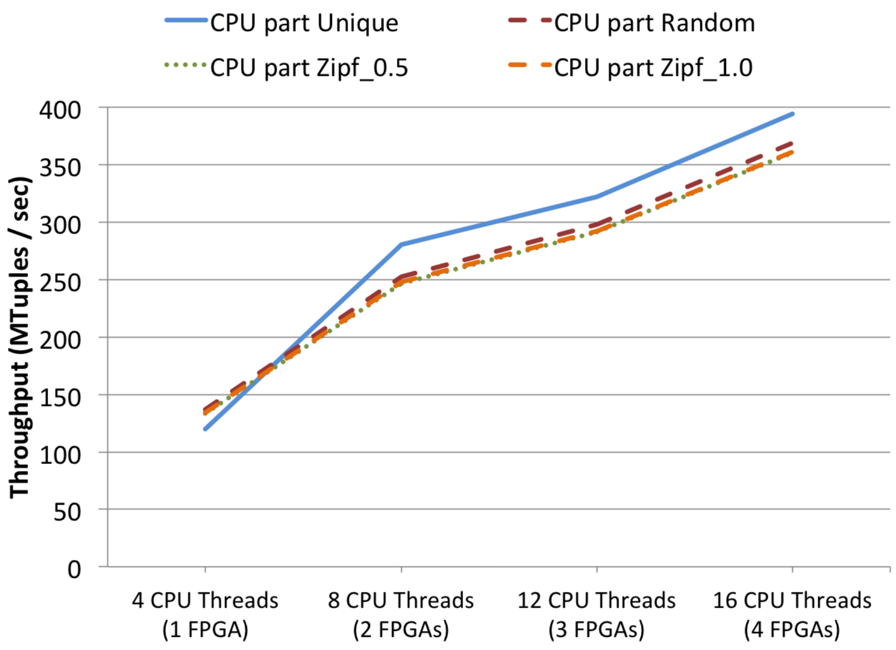


Figure 5.11: Partitioned CPU throughput scaling. The Build Relation has 2^{21}, Probe has 2^{28} tuples

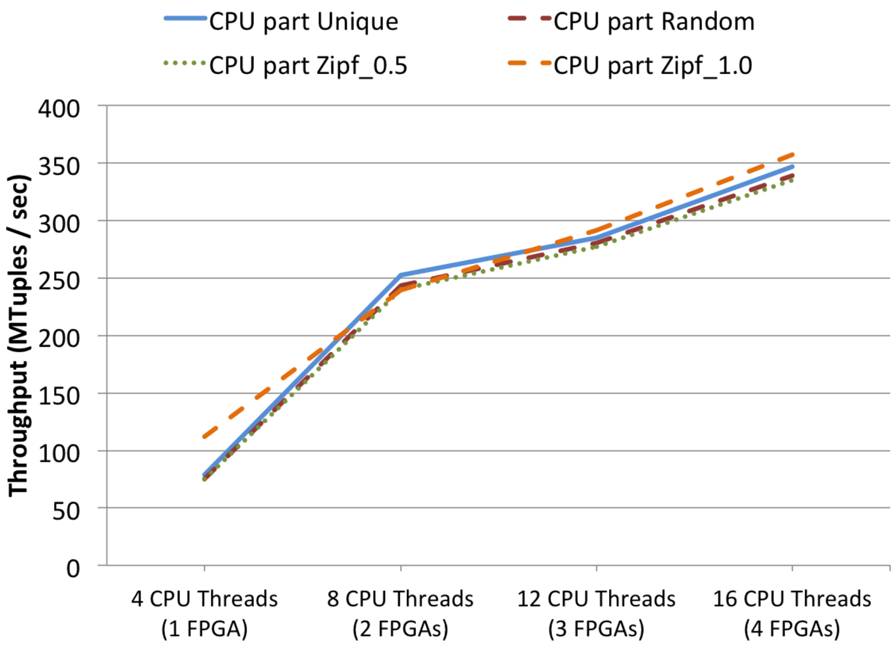


Figure 5.12: Partitioned CPU throughput scaling. The Build and Probe Relations both have 2^{28} tuples

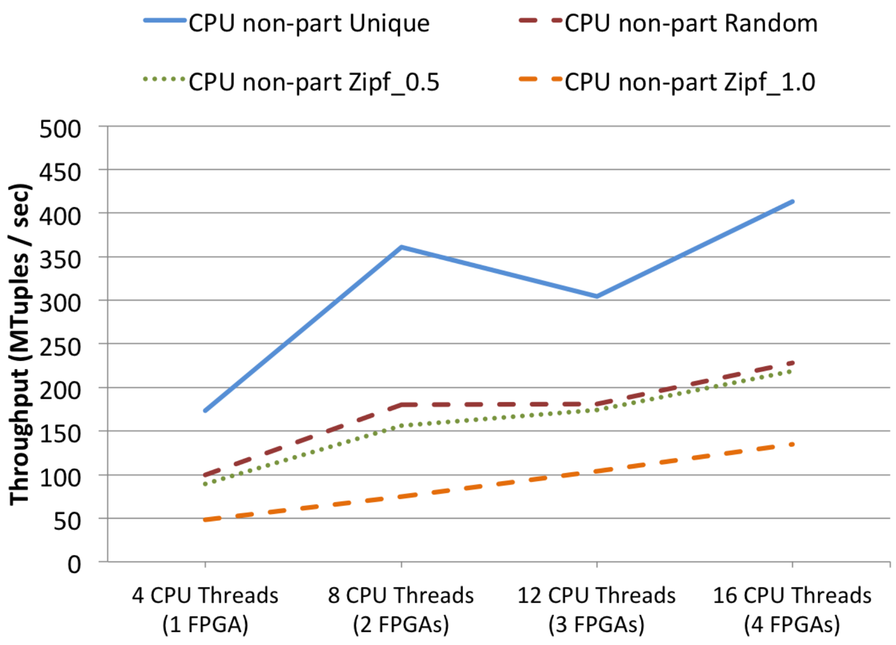


Figure 5.13: Non-Partitioned CPU throughput scaling. The Build Relation has 2^{21}, Probe has 2^{28} tuples

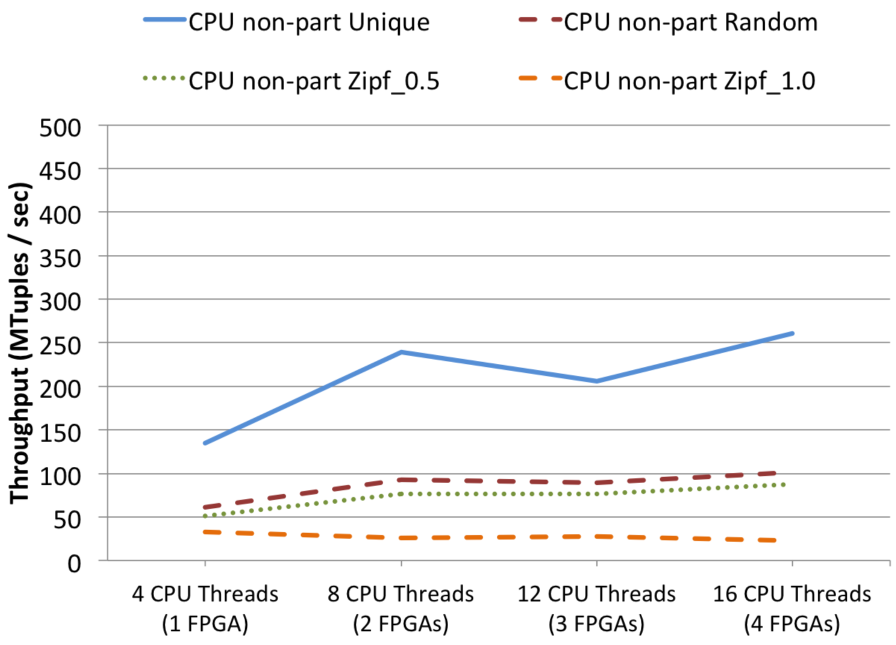


Figure 5.14: Non-Partitioned CPU throughput scaling. Build and Probe Relations both have 2^{28} tuples

show the results when the probe phase dominates the computation. The FPGA scales linearly on datasets *Unique*, *Random* and *Zipf_0.5* (Figure 5.9).

However, for the *Zipf_1.0* dataset, the performance does not scale because of the extreme skew. Each probe job searches through an average of 4.8 to 6.7 nodes in the linked list. Therefore most jobs are recycled through the datapath multiple times. Having too many jobs being recycled limits the new jobs entering the datapath causing back pressure and stalling. The partitioned algorithm scales as the number of threads increases but at a lower rate than the FPGA approach (depicted on Figure 5.11). The non-partitioned algorithm shows a drop in performance while moving from 8 to 12 threads because of the NUMA latency emerging while moving from 1 to 2 CPUs (Figure 5.13).

The FPGA scales at a lower rate when the build and probe relation are of the same size (Figure 5.10), since the throughput of the build phase remains constant while the probe phase scales. The slope of the scale graph is almost comparable to the CPU implementations (shown on Figures 5.12 and 5.14). Again the extreme skew case does not scale for the FPGA.

5.3.6 Throughput Efficiency

To get a direct comparison of throughput we normalize it to the available bandwidth. As discussed in Section 5.3.1 each FPGA has 19.2 GBs of bandwidth, and each CPU has 51.2 GBs. The normalized results are shown in Figure 5.15 and 5.16. When the probe relation dominates the computation (Figure 5.15) the FPGA shows speedup between 3.2x and 6x on the *Unique* dataset. It shows a speedup between 4.4x and 10x on the *Random* and *Zipf_0.5* datasets. Finally, it shows speedup between 0.6x

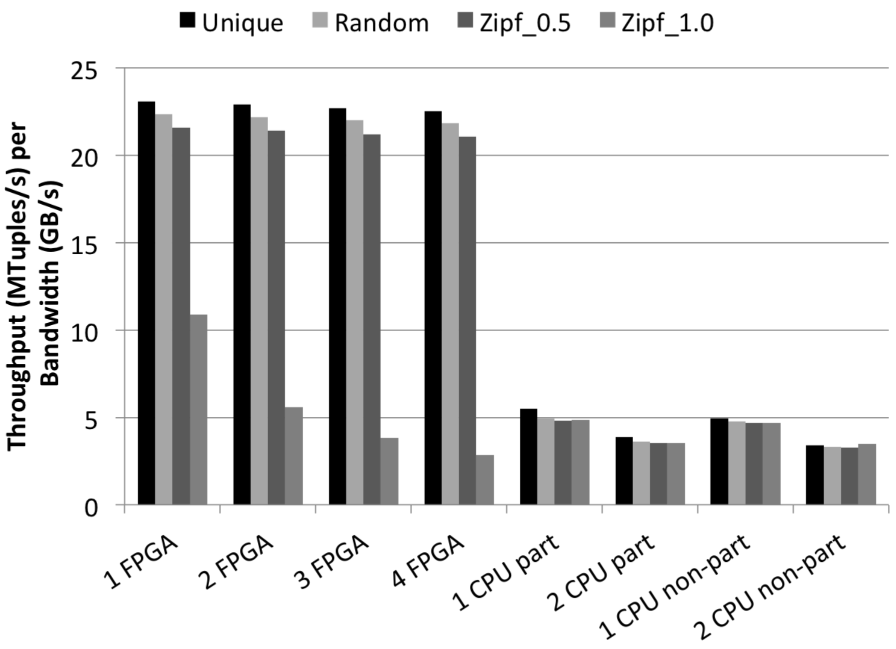


Figure 5.15: Throughput efficiency when Build Relation has 2^{21}, Probe has 2^{28} tuples

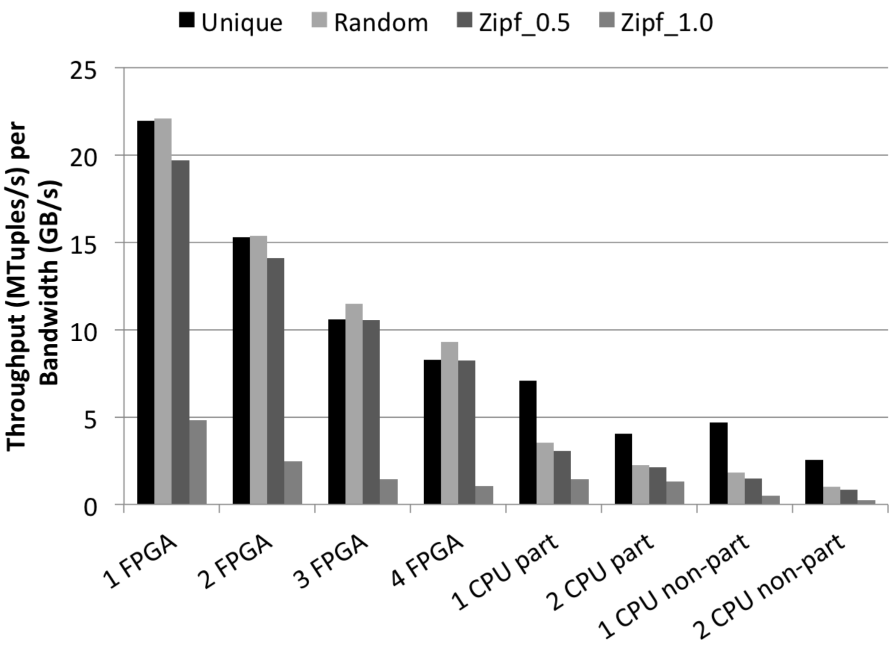


Figure 5.16: Throughput efficiency when Build and Probe Relations both have 2^{28} tuples

and 8.3x on the *Zipf_1.0* dataset. When neither relation dominates the computation (Figure 5.16) the FPGA shows speedup between 1.7x and 8.6x on the *Unique* dataset. It shows a speedup between 1.7x and 23.1x on the *Random* and *Zipf_0.5* datasets. Finally, it shows speedup between 0.2x and 21.6x on the *Zipf_1.0* dataset.

Table 5.1: FPGA Resource utilization.

# engines	Registers	LUTs	BRAMs
1 probe	65678 (7%)	62521 (13%)	104 (14%)
2 probe	81712 (9%)	74951 (16%)	133 (18%)
3 probe	94799 (10%)	86200 (18%)	154 (21%)
1 build	112476 (16%)	118169 (33%)	41 (4%)
2 build	117202 (17%)	123890 (35%)	48 (5%)
3 build	121408 (17%)	129592 (37%)	55 (6%)
4 build	125588 (18%)	135908 (38%)	62 (7%)

5.3.7 FPGA Area Utilization

Table 5.1 shows the resource utilization (registers, LUTs and BRAMs used) for the different FPGA designs. We observe that many resources are shared between engines as their number increases. For example, one probe engine uses 7% of the available registers, whereas three engines utilize only 10% of the register file. Note that the build phase uses much more logic resources (LUT) due to its atomic operations, but it also has very low BRAM utilization. Overall, the space utilization on the FPGA is low, leaving sufficient space to extend out design for with various optimizations (selections, projections, join step).

5.4 Conclusion

We have presented the performance benefits of the first end-to-end FPGA implementation of hash joins. Our approach is different as the entire hash-table is built in memory, leveraging FPGA multithreading to deal with long memory latencies. As hashing itself is a basic building block for many relational operator implementations, the presented FPGA design could be extended to support other operations like group-by aggregations, duplicate elimination, unions, intersections etc. Furthermore, we are examining how partitioning and thread load balancing can be utilized on the FPGA approach so as to deal with extremely skewed datasets.

Chapter 6

FPGA based Multithreading for In-Memory Aggregation

Aggregation is widely used in relational databases to group information, or to count the occurrence of various values. It is a more challenging algorithm, from a hardware accelerator point-of-view, than hash join. Both designs maintain a hash table in memory, but they access it in very different fashions. Join runs two distinct phases sequentially without any overlap. During the build phase tuples from one relation are used to build the hash table, and every job inserts (i.e. writes) a new node. During the probe phase tuples from another relation are used to read the hash table, and no job writes to the table. The aggregation operation builds a histogram, and before it inserts a new node into the table it must first verify the key does not already exist. In short aggregation has the option to update an existing, or create a new node, which means each job must read and write to the hash table.

In this section we present a multithreaded FPGA kernel design for the aggregation operation. We attempt to address the atomic operation performance issue from

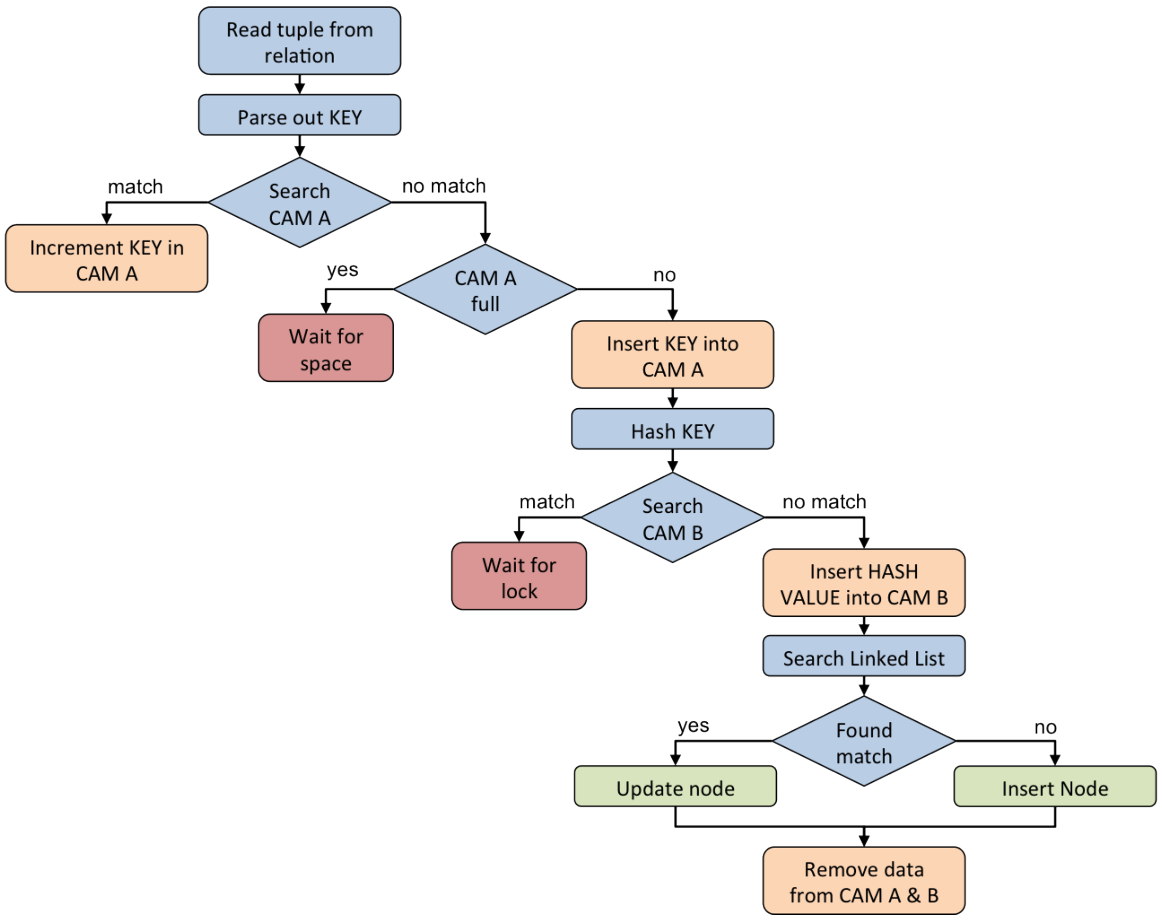


Figure 6.1: A flow chart for an aggregation job through the kernel.

Chapter 5 by utilizing on-chip CAM logic. Results are compared against five different software approaches.

6.1 A FPGA Aggregation Kernel

The aggregation kernel is design for large scale database workloads, and the datasets are assumed to be too large to store locally on the FPGA. Accesses to global memory incur long latencies. Our kernel uses a multithreaded approach to mask latency and efficiently utilize the available bandwidth. The implementation uses custom CAM

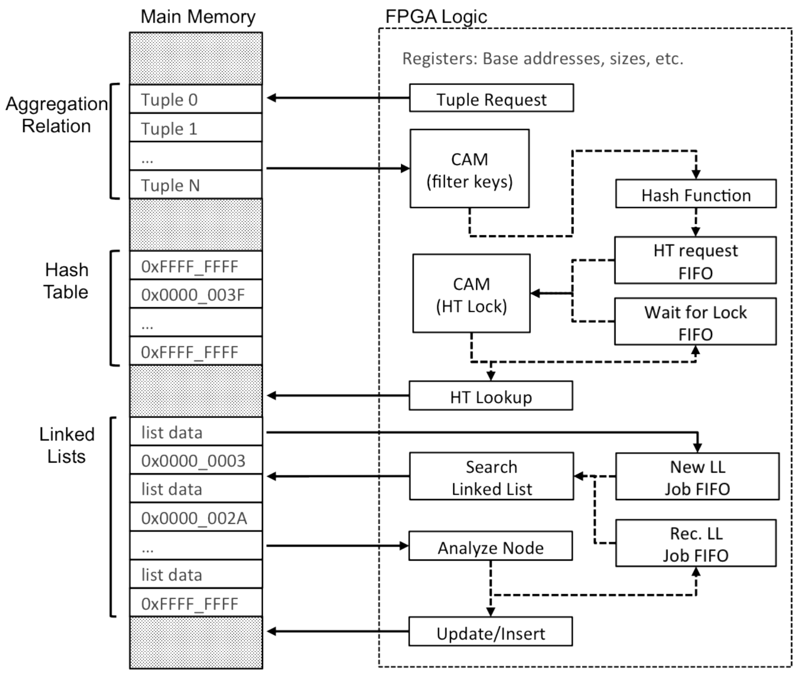


Figure 6.2: The FPGA aggregation engine.

units (local to the FPGA) to reduce the memory requests, and enforce atomic operations (e.g. lock memory locations). The decision to utilize CAMs is made for performance reasons. They allow the kernel to reduce the global memory requests by accumulating data from identical keys locally. They also allow the FPGA to enforce locking on specific memory channels instead of across all channels. In Chapter 5 Section 5.3.4 the Convey MX's atomic operations were shown to be a bottleneck.

Figure 6.1 shows a flow chart for one job through the aggregation engine. Figure 6.2 shows the layout, and memory channels of the aggregation engine. Each tuple from the relation is treated as a unique job, and is assigned its own thread on the FPGA. Jobs are first streamed from global memory by the *Tuple Request* component. Upon arrival they are sent to the first data CAM *filter keys* which combines duplicate keys into a single job on the FPGA. As an example assume the first 5 tuples have the following keys: A, B, A, C, A. The CAM will merge data from the 4th and 5th tuples with they data from the 1st tuple because their keys match. The design assumes a count aggregation function, and therefore the CAM maintains an occurrence count of duplicate keys. Continuing with the example, the first tuple with key A misses in the CAM so it updates the CAM with the key value pair (A, 1). The job then continues through the FPGA to search for a matching node in the hash table. The second tuple with key A hits in the CAM so it updates the key value pair to (A, 2), and the FPGA terminates the job. The third tuple with key A is also terminated after updating the key value pair to (A, 3). Eventually the first job (which was not terminated) will complete and increment an existing node with the value of 3, or create a new node for key A with an initial value of 3.

Jobs that do not get merged into other threads by the first CAM search the hash table for an existing node with the same key. The design uses linked lists as the conflict resolution strategy (i.e separate chaining). Memory latency plays an key part in this phase of the execution. If two keys share a hash value, and neither find a match in the linked list they both will try and insert a new node to the chain. Because the latency can be hundreds of cycles a race condition is created. Both threads will create a new node, and attempt to connect it with the same existing node. To prevent this condition a second CAM, *HT Lock*, is used to lock the hash table's location that holds the linked list's head pointer. Jobs are sent to the *Wait for Lock FIFO* and stalled until the lock is released.

Once a job obtains the lock, and passes through the second CAM it begins searching through the linked list. First it requests the head pointer, which is stored in the hash table. A job follows the linked list chain one node at a time. If it ever finds a matching key it updates the node. If it reaches the last node in the list the thread allocates a new node, and inserts it into the chain. When the thread needs to update or initialize a node it reads the value from the *filter keys* CAM, which holds the count for all jobs merged into this thread. The thread's key value is flushed from both CAMs freeing the locks, and allowing other jobs to unstall and continue their execution. The engine is done once no more threads are in the datapath.

6.2 Implementation Considerations & Limitations

By incorporating CAMs into the FPGA design we can increase the portability. To my knowledge the Convey MX is the only commercially available platform offering support for atomic operations. Using generic CAMs to enforce locks means the atomic operations are all internal to the FPGA chip, and can be routed on any platform with sufficient area. This also allows the design to lock only the memory channels which need locking. On the MX if any channel needs atomic operations then a custom wrapper is needed which has atomic logic for all channels.

CAMs allow fast lookups, which the design uses merge duplicate keys into a single job. It reduces the number of global memory requests because fewer jobs are in the engine. However, CAMs are difficult to implement on reconfigurable fabrics. Their ability to check multiple locations within a single cycle results in high fanout rates that inflate quickly as the CAM grows in size. Current FPGA chips offer little custom hardware support (i.e. DSP units, BRAMs, etc.) for CAMs, and therefore they can be area intensive. Our design was limited by CAM size. A Convey HC-2ex is used to implement the design as discussed in Section 6.3.1, and it needs more than 500 threads to fully mask the memory latency. However, each CAM only holds 128 unique elements, which was done to meet the 150 MHz timing constraint. Platforms with higher or lower clock frequencies can use smaller or larger CAMs.

Multiple aggregation engines can be placed on a single FPGA, or duplicated across multiple FPGAs. A single CAM could be shared among all these engines, but it would create many routing issues. In addition communication channels between FPGAs are typically limited. Therefore the engines in our design utilize their own distinct CAMs. It increases modularity, and makes them truly parallel. However, doing so requires each engine to work on its own hash table, and introduces an extra merging phase at the end of the computation. The merge step can be made streamable by forcing the aggregation engines to build sorted linked list. This introduces negligible overhead because linked lists are not sequential data structures. It can also reduce the search time to find matching key nodes.

6.3 Experimental Evaluation

The multithreaded aggregation approach is compared to a number of software implementations. Software designs are run on a single Intel Xeon E5-2643 CPU, and the hardware design is run on 2 Xilinx Virtex-7 LX760 FPGAs. Because aggregation is a memory bounded application we compare 2 FPGAs with 1 CPU so the memory bandwidth is comparable. However, at 38.4 GB/s the FPGA is still giving a 33% bandwidth advantage to the CPU, which has 51.2 GB/s.

6.3.1 FPGA & Software Implementations

All software and hardware experiments are performed on th same machine; a Convey HC-2ex. It is a heterogeneous platform that offers a shared global memory space between the software and hardware. While the memory can be directly accessed by any processor it is divided into regions connected through PCIe with portions closer to the CPU, and portions closer to the FPGAs. The hardware region has 4 runtime programmable Xilinx Virtex-6 LX760 FPGAs with fast access to 64 GBs of global memory. The software region has 2 Intel Xeon E5-2643 CPUs with fast access to 128 GBs of global memory.

Each of the 4 FPGAs has 8 memory controllers that run at 300 MHZ, but to simplify timing they are duplexed into 16 channels at 150 MHz for the FPGA logic. All channels have an 8-byte data bus that connects to Convey's custom crossbar. The engine proposed in Section 6.1 requires 4 memory channels. A single channel to stream in the tuples. Another channel to communicate with the in-memory hash table. Two channels for the in-memory linked lists. To better utilize the available bandwidth four engines are placed on a single FPGA. The hardware kernel has a total of 8 multithreaded aggregation engines running in parallel across 2 FPGAs. The peak theoretical throughput for the FPGA is therefore 1200 MTuples/s. However, as stated in Section 6.2 our implementation is limited by the CAM size, and we do not expect to sustain such a high throughput.

Multiple software approaches are compared against[1]. The algorithms are briefly outlined here:

- **Individual Tables** is similar to the algorithm implemented in hardware. The tuples are evenly split between individual threads (without partitioning), and each thread works with its own hash table. Once the aggregation is complete all tables must merge their results together.

[1]All software implementations were written and run by Ildar Absalyamov. Their description is included here for comparative purposes.

- **Shared Table(atomic)** splits the tuples evenly between threads, but all threads write to a single hash table. Intel specific locking hardware is used to synchronize the execution. No merge step is needed.

- **Hybrid Table** is a combination of *Individual Tables* and *Shared Table.* Each thread has its own small table based on the processor's L2 cache size. Values that match, or that fit are written to the small table. Values that do not are overflowed into a large shared table. Once aggregation is complete the small tables are merged into the large shared table.

- **Partitioning (Individual Output)** uses individual tables per thread, but before aggregation is performed the tuples are partitioned. Therefore, each table works on unique values and the final tables can be concatenated instead of merged.

- **PLAT (Partitioned Local Aggregation Tables)** is a combination of all the previous techniques. It first partitions the tuples into workloads with mutually exclusive keys. Each thread has its own small table based on the L2 cache size. Values that do not fit into the small table are written to a larger table. Once aggregation completes the small tables are merged with their large tables. Finally the large tables are concatenated together.

6.3.2 Dataset Description

Aggregation performance in software drops off significantly as the dataset's key cardinality increases. When the CPU's caches can hold the entire hash table the performance is high because few memory requests are needed. We compare the hardware and software design on 5 datasets with varying key distributions. Each dataset consists of 5 benchmarks with cardinalities ranging from $2^1$0 unique keys to $2^2$2 unique keys.

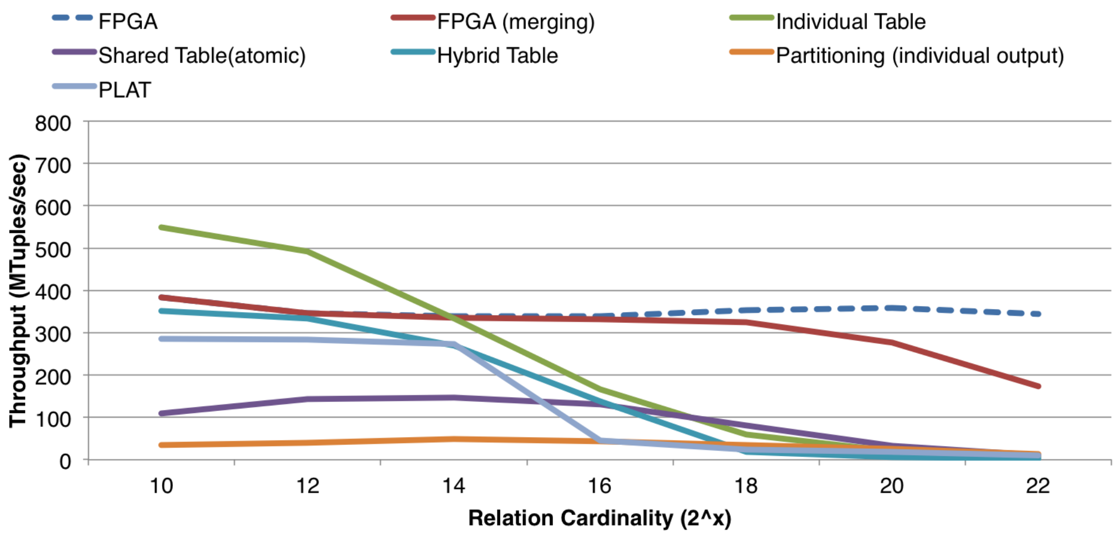


Figure 6.3: Aggregation throughput for hardware and software approaches on the uniform dataset.

- In the **Uniform** dataset all key values are randomly generated. They are unordered and evenly distributed throughout the relation.
- In the **Heavy Hitter** dataset a single key value is shared by 50% of the tuples. The remaining key values are evenly distributed throughout the rest of the relation.
- In the **Moving Cluster** dataset key values are grouped into clusters. Lower key values are more likely to appear at the beginning of the relation, and higher key values are more likely to appear at the end of the relation.
- In the **Self Similar** dataset a single key value is shared by 20% of the tuples. Of the remaining 80% of tuples 20% of those share a single key value. This pattern is repeated recursively to generate the relation. Tuples are randomly shuffled. The algorithm is described in [26].
- In the **Zipf** dataset key values fall the Zipf distribution with a coefficient factor of 0.5. The algorithm is described in [26].

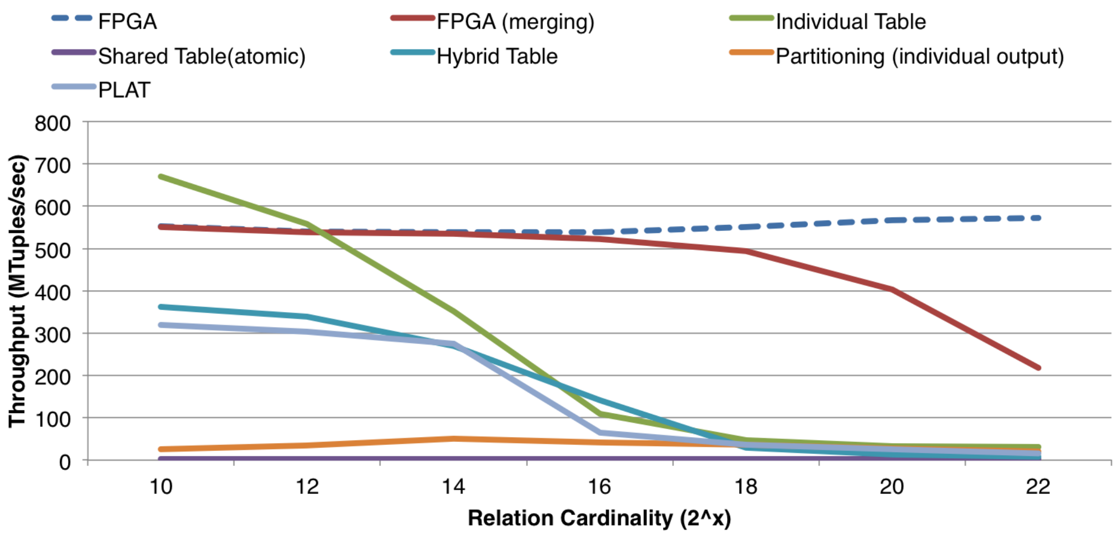


Figure 6.4: Aggregation throughput for hardware and software approaches on the heavy hitter dataset.

6.3.3 Throughput Evaluation

Throughput is considered across 2 FPGA, and 5 software (2 partitioned, and 3 non-partitioned) approaches. All software results were run on a single Xeon E5-2643 CPU with 8 hardware threads. The hardware results are run on 2 Xilinx Virtex-6 LX760 FPGAs. 2 FPGAs are compared with 1 CPU to balance out the software's memory bandwidth advantage. A single CPU has 51.2 GB/s of memory bandwidth, but the 2 FPGAs combined only have 38.4 GB/s. The CPU still has the advantage with 30% more bandwidth, but it is not as severe.

Key cardinality is varied for each of the dataset's benchmarks, and the throughput performance results are shown in Figures 6.3 to 6.7. Software performance is inversely related to the dataset cardinality. Benchmark's with few unique keys create few hash table nodes, which better fit in the software's caches. However, as the number of unique keys increases caches become less effective. This trend is shown across all 5 benchmarks. Partitioned algorithms are outperformed by non-partitioned algorithms,

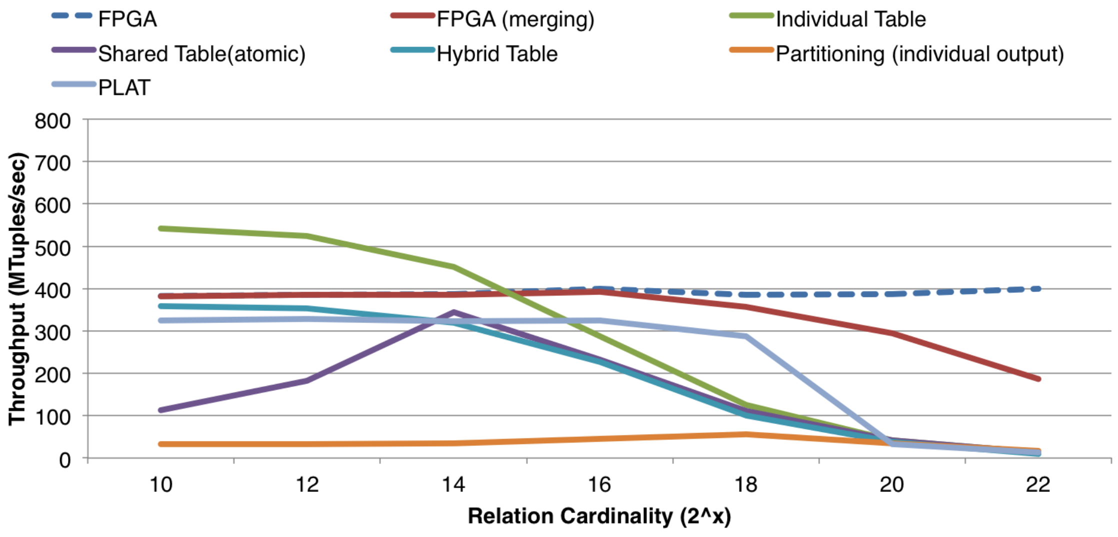


Figure 6.5: Aggregation throughput for hardware and software approaches on the moving cluster dataset.

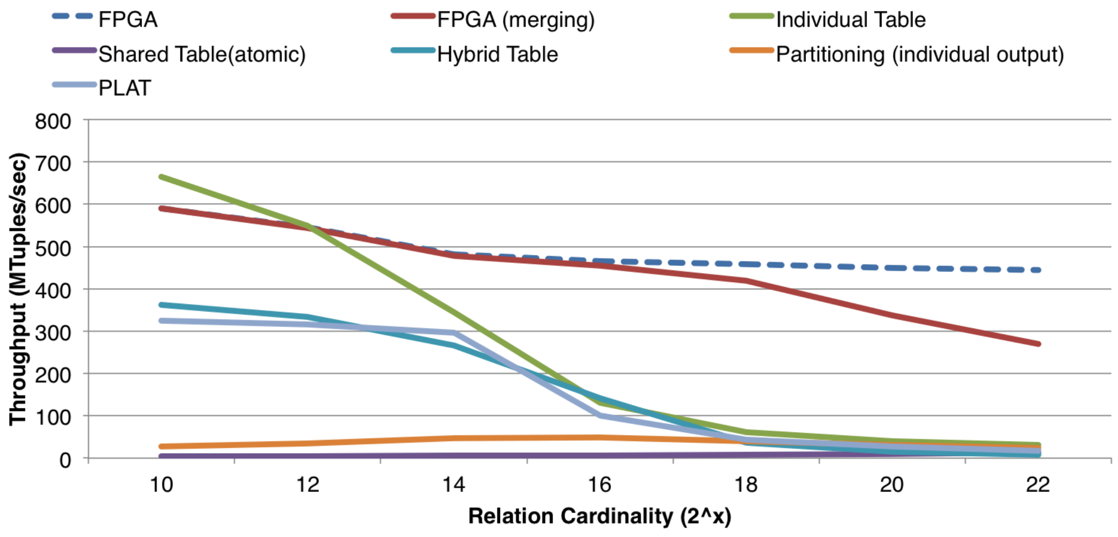


Figure 6.6: Aggregation throughput for hardware and software approaches on the self similar dataset.

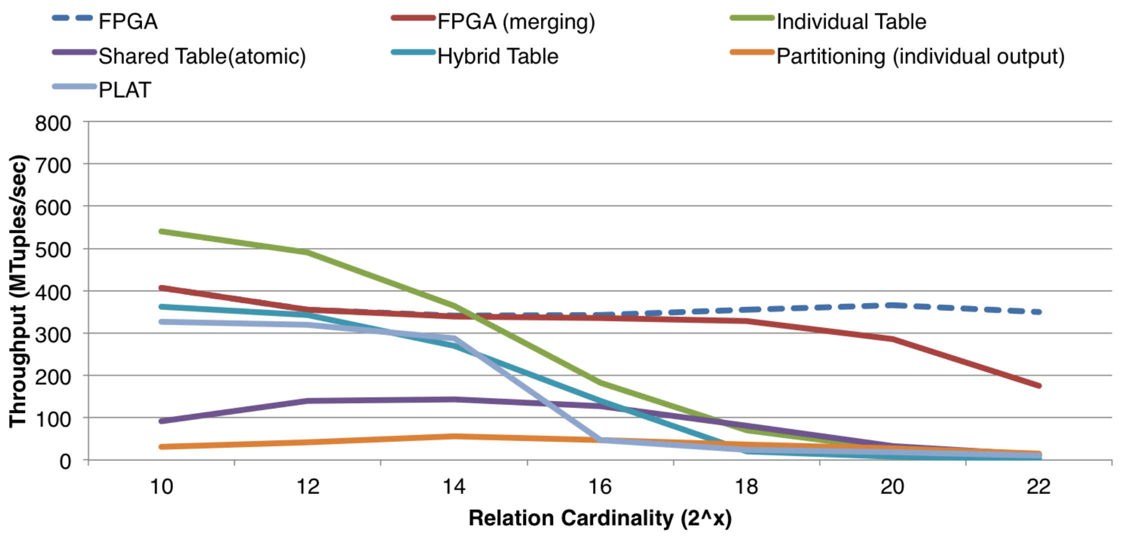


Figure 6.7: Aggregation throughput for hardware and software approaches on the Zipf 0.5 dataset.

but the difference is only noticeable with low cardinality. Throughput over 500 MTuples/sec is possible on benchmarks with around 1K unique keys. However, throughput drops below 100 MTuples/sec for all algorithms, on all datasets when the benhmarks have more than 1M unique keys. Software performs best on the *Moving Cluster* dataset; Figure 6.5. This is because the key values are grouped together within the benchmarks, which allows the CPU to better utilize its caches.

The mulithreaded hardware approach has 8 custom engines that build 8 separate hash table. Two sets of results are reported in Figures 6.3 to 6.7. The *FPGA* approach only builds the hash tables, and once completed the results are split across 8 hash tables. The *FPGA(merging)* approach adds an extra step where all nodes are merged together into a single table. With 8 engines running at 150 MHz the peak performance is 1200 MTuples/sec, but our design is limited to 128 outstanding jobs by the CAM size. The FPGA has consistent performance across all benchmarks, when not considering the merge step, in 4 of the 5 datasets. *Uniform*, *Moving Cluster*, and *Zipf*

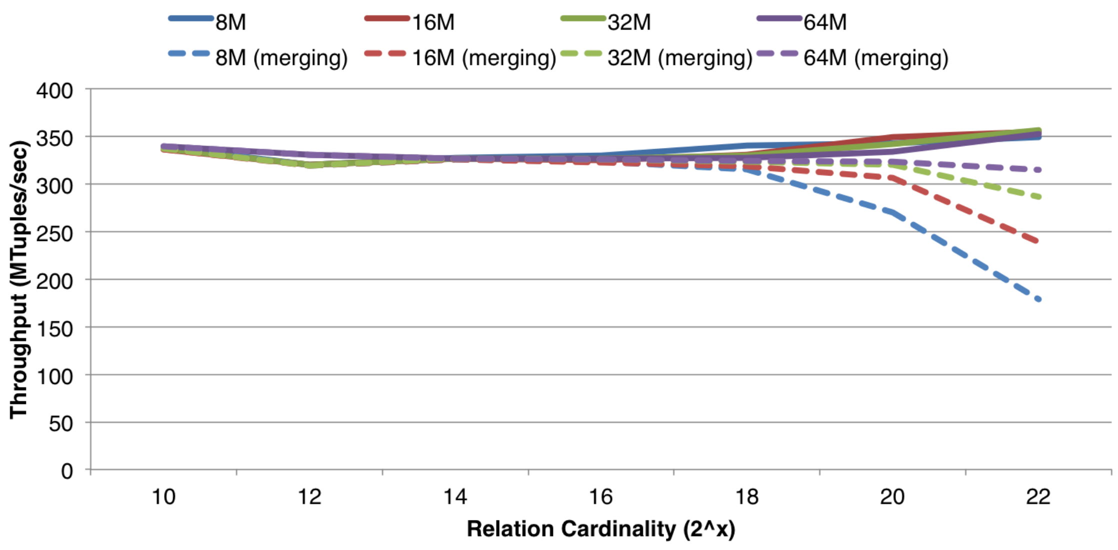


Figure 6.8: Aggregation throughput for the FPGA on multiple relation sizes with uniform key distribution. Results are shown without (solid line), and with (dashed line) the merge operation.

all sustain throughput between 300 MTuple/sec and 400 MTuples/sec. *Heavy Hitter* has higher performance ranging between 500 and 600 MTuples/sec because 50% of the tuples share a key value. The duplicate keys are merged together into a single outstanding request, and merging jobs does not stall the datapath. *Self Similar* is the only dataset where performance drops as the cardinality increases. At low cardinality many tuples share the same keys, but the distribution disburses as the cardinality increases. At low cardinalities it performs similar to *Heavy Hitter*, and at high cardinalities it performs similar to the other datasets.

The merge step has a fixed cost that depends on the number of unique keys. Aggregation creates a histogram of results so all benchmarks, no matter the size, will reduce into a fixed number of nodes. Results show that the merge step is insignificant to performance below 128 million keys, but the performance begins to drop at higher cardinality. The performance is still an order of magnitude higher than software.

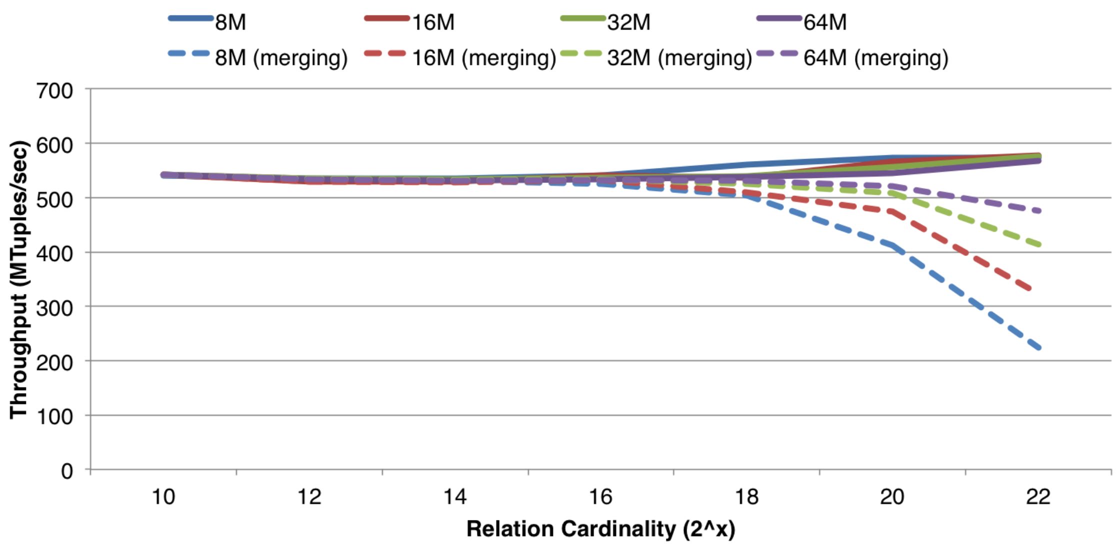


Figure 6.9: Aggregation throughput for the FPGA on multiple relation sizes with heavy hitter key distribution. Results are shown without (solid line), and with (dashed line) the merge operation.

6.3.4 Effects of the Merge Operation

To easily interface the FPGA aggregation with existing database management systems the final result should have all matching keys allocated to one node in a hash table. Managing memory locks locally on the FPGA improved portability, but it also required each engine to work on individual hash tables. In our case this means a single key can be spread across 8 hash table. To complete the operation an extra merging step must be done. As discussed in Section 6.3.1 we build sorted linked-list chains, which allows the merging to be done in a streaming fashion.

The merge step requires finite time dependent on the key cardinality. It is independent of the relation's size, or key distribution. The merge step will have a larger effect on overall performance for high cardinality benchmarks, but the effect will be less pronounced as the relation size increases.

Figures 6.8 and 6.9 show the throughput performance for multiple relation sizes. The solid-lines show performance when only the aggregation step is performed.

The dashed-lines show performance when the merge step is included. We report results for two datasets (*uniform* and *heavy hitter*) which are representative of all five datasets. As expected the merging operation's effects are only noticeable on high cardinality benchmarks, and the effect is lessened as the relation size increases from 8 million tuples to 64 million tuples.

6.4 Conclusion

In this chapter we have presented a multithreaded FPGA implementation for the aggregation operation. The design is easily portable between FPGA architectures because is uses CAMs to enforce synchronization. All data structures are stored in main memory, which allows the DBMS to seamlessly transition between software and hardware execution. Results show that throughput performance is consistent and predictable regardless of a relation's size and cardinality. However, the final merge step does affect performance when the relation size and cardinality are similar (i.e. few tuples are aggregated together). Performance ranges between 300 and 600 MTuples/sec depending on the key distribution. Results show that the multithreaded FPGA approach can significantly outperform many software approaches on high cardinality benchmarks.

Chapter 7

Conclusion

Many interesting problems that require sparse matrices, hash tables, or any irregular data structures are going to have trouble with cache centric hardware. Regardless, it seems unlikely that the major chip manufactures are going to move away from their cache model. Luckily there has been a push by the hardware industry toward heterogeneous platforms. Companies are actively developing FPGA accelerators that easily connect through PCIe, and providing APIs to easily integrate them with existing software. These advances allow researchers to quickly develop and prototype algorithms that do not rely on caching to cope with long memory latencies. Since the 1980s processor performance has outpaced memory performance, causing memory bandwidth to be the bottleneck for many applications.

Latency masking multithreaded architectures have existed since the early 1990s. They perform well on applications with sufficient parallelism, but identifying parallelism is a non-trivial task. Some chip manufactures like Oracle, with its UltraSPARC T series, have partially adopted this platform into their CPU designs. However, The emerging heterogeneous platforms can offer the best of both worlds. Caches work well for regular applications, but the irregular applications can be offloaded to accelerators with custom multithreaded kernels.

Programmability is the first issue addressed by this thesis. Designing custom hardware is notoriously difficult. The startup costs often discourage software developers from building/using hardware circuits. We present the CHAT tool, which is a C to VHDL compiler that can generate custom memory masking mulithreaded hardware circuits. It is designed to be portable with standard FIFO interfaces. Developers only need to know enough hardware logic to connect the FIFOs to their memory architecture. Results from Chapter 3 showed that the multithreaded model can accelerate memory bounded kernels with irregular access patterns by 2x over software implementations.

In Chapter 4 the CHAT tool is used to generate a custom SpMV kernel. Many important problems in high performance computing and scientific computing rely on sparse matrices. They have been used to represent economic models, hardware circuits, graphs, and simulation runs. Using benchmarks chosen by other research groups we showed that a multithreaded FPGA approach can perform as well as CPU and GPU platforms. However, these benchmarks exhibited regular memory access patterns that are well suited to the CPU's caches, and the GPU's memory coalescing. Performance when using irregular sparse matrices showed that the FPGA's performance was higher than the GPU in terms of throughput. This even though the GPU had 2.7x the memory bandwidth of the FPGA; 208 GB/s vs. 76.8 GB/s.

In Chapter 5 we present the first FPGA based in-memory hash join algorithm for relational databases. Hash join is handled in two phases. The build phase which creates a hash table based on one relation, and the probe phase that read and compares against the hash table for another relation. The custom kernels rely on multithreading to cope with the long memory latency that can take upwards of 500 cycles to complete.

An end-to-end implementation requires atomic operations to prevent race conditions during the build phase. Our design utilized custom memory locks provided by the Convey MX platform, but the same functionality could be had by incorporating CAMs into the FPGA kernel. The overhead for atomic operations limited performance, but the multithreaded approach still outperformed software (which used caches) on uniform and slightly skewed datasets. Heavily skewed datasets resulted in long linked list chains that made performance unpredictable.

In Chapter 6 we present a portable FPGA based in-memory aggregation algorithm for relational databases. The design uses custom CAM logic to enforce memory locking, and ensure synchronization while creating the hash table. The implementation is more challenging than hash join because each tuple must read through the linked list chains searching for a match before it can write to memory. Multithreading is again used to cope with long memory latencies, but the design is ultimately limited by the CAM size. Regardless results show the FPGA approach can sustain between 300 and 600 MTuples/sec on a variety of key distributions.

Bibliography

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[2] Altera. *Altera SDK for OpenCL*, 2014.

[3] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th International Conference on Supercomputing*, pages 188–197, 1992.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, 1990.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 362–373, 2013.

[7] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[8] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA, 2008.

[9] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–48, 2011.

[10] J. Boisseau, L. Carter, A. Snavely, D. Callahan, J. Feo, S. Kahan, and Z. Wu. Cray t90 vs. tera mta: The old champ faces a new challenger. 1998.

[11] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65, 1999.

[12] J. Branscome, M. Corwin, L. Yang, J. Shau, R. Krishnamurthy, and J. I. Chamdani. Processing elements of a hardware accelerated reconfigurable processor for accelerating database operations and queries. Patent: US 20080189251 A1, 2008.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–36, 2011.

[14] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, pages 151–160, 2014.

[15] G. Chin, A. Marquez, S. Choudhury, and K. Maschhoff. Implementing and evaluating multithreaded triad census algorithms on the cray xmt. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–9, 2009.

[16] Convey Computers. http://www.conveycomputer.com/.

[17] D. Crookes, K. Benkrid, A. Bouridane, K. Alotaibi, and A. Benkrid. Design and implementation of a high level programming environment for fpga-based image processing. *IEEE Proceedings Vision, Image and Signal Processing*, 147(4):377–384, 2000.

[18] CUSP-library. http://cusplibrary.github.io/.

[19] M. DeLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for fpgas. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 75–85, 2005.

[20] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 28–34, 2005.

[21] E. Fernandez, W. Najjar, E. Harris, and S. Lonardi. Exploration of short reads genome mapping in hardware. In *International Conference on Field Programmable Logic and Applications*, pages 360–363, 2010.

[22] E. Fernandez, W. Najjar, and S. Lonardi. String matching in hardware using the fm-index. In *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 218–225, 2011.

[23] P. Francisco et al. The netezza data appliance architecture: a platform for high performance data warehousing and analytics. *IBM Redbooks*, 2011.

[24] M. Gerards, J. Kuper, A. Kokkeler, and B. Molenkamp. Streaming reduction circuit. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 287 –292, 2009.

[25] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo. Hashing strategies for the cray xmt. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1–8, 2010.

[26] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 243–252, 1994.

[27] Z. Guo, A. Buyukkurt, J. Cortes, A. Mitra, and W. Najjart. A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming*, 36(5):493–520, 2008.

[28] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with fpgas. In *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20, 2013.

[29] F. D. Hinshaw, D. L. Meyers, and B. M. Zane. Programmable streaming data processor for database appliance having multiple processing unit groups. Patent: US 7577667 B2, 2009.

[30] R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the denelcor hep. *Parallel Computing*, 1(3):197–206, 1984.

[31] IBM Netezza. http://www.ibm.com/software/data/netezza/.

[32] M. Ionescu and K. Schauser. Optimizing parallel bitonic sort. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 303–309, 1997.

[33] S. Jin, J. Cho, X. Dai Pham, K. M. Lee, S.-K. Park, M. Kim, and J. W. Jeon. Fpga design and implementation of a real-time stereo vision system. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(1):15–26, 2010.

[34] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[35] R. Krishnamurthy, C. Ku, J. Shau, C. Zhang, K. Surlaker, J. Branscome, M. Corwin, and J. Chamdani. Methods and systems for generating query plans that are compatible for execution in hardware. US Patent App. 12/168,821, 2010.

[36] J. Kuehnand and B. Smith. The horizon supercomputing system: architecture and software. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 28–34, 1988.

[37] M. Kumar and D. Hirschberg. An efficient implementation of batcher's odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Transactions on Computers*, 100(3):254–264, 1983.

[38] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[39] LegUp. http://legup.eecg.utoronto.ca/.

[40] K. Li and W. Yang. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transaction on Parallel and Distributed Systems*, 2014.

[41] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[42] Maxeler Technologies. http://www.maxeler.com/.

[43] K. Meiyyappan, L. Yang, J. Branscome, M. Corwin, R. Krishnamurthy, K. Surlaker, J. Shau, and J. I. Chamdani. Accessing data in a column store database based on hardware compatible indexing and replicated reordered columns. Patent: US 20090254516 A1, 2009.

[44] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras. Boosting xml filtering with a scalable fpga-based architecture. *Conference on Innovative Data Systems Research*, 2009.

[45] D. Mizell and K. Maschhoff. Early experiences with large-scale cray xmt systems. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–9, 2009.

[46] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers*, pages 111–125. 2010.

[47] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Accelerating xml query matching through custom stack generation on fpgas. In *High Performance Embedded Architectures and Compilers*, pages 141–155. 2010.

[48] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: A query compiler for fpgas. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[49] K. Nagar and J. Bakos. A sparse matrix personality for the convey hc-1. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 1 –8, 2011.

[50] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga. An implementation of handshake join on fpga. In *Second International Conference on Networking and Computing*, pages 95–104, 2011.

[51] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga. A fast handshake join implementation on fpga with adaptive merging network. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 44:1–44:4, 2013.

[52] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez. Optimization of sparse matrix–vector multiplication using reordering techniques on gpus. *Microprocessors and Microsystems*, 36(2):65–77, 2012.

[53] Pico Computing. http://picocomputing.com/.

[54] Riverside Optimizing Compiler for Configurable Computing. http://roccc.cs.ucr.edu.

[55] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on fpgas. In *Proceedings of the 2012 IEEE International Conference on Data Engineering*, pages 1229–1232, 2012.

[56] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment*, 3(1-2):1525–1528, 2010.

[57] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul. Accelerating dynamic time warping subsequence search with gpus and fpgas. In *IEEE 10th International Conference on Data Mining*, pages 1001–1006, 2010.

[58] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. FPGA and GPU implementation of large scale SpMV. In *IEEE 8th Symposium on Application Specific Processors*, pages 64 –70, 2010.

[59] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, 1994.

[60] B. Smith. The architecture of hep. In *On Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 41–55, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology.

[61] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multiprocessor performance on the tera mta. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–8, 1998.

[62] B.-Y. Su and K. Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 353–364, 2012.

[63] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, 2012.

[64] J. Sun, G. Peterson, and O. Storaasli. Sparse matrix-vector multiplication design on fpgas. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 349–352, 2007.

[65] Teradata Kickfire. http://www.teradata.com/.

[66] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 625–636, 2011.

[67] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[68] M. R. Thistle and B. J. Smith. A processor architecture for horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, pages 35–41, 1988.

[69] I. S. Uzun, A. Amira, and A. Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. In *IEEE Proceedings Vision, Image and Signal Processing*, volume 152, pages 283–296, 2005.

[70] F. Vazquez, E. Garzon, J. Martinez, and J. Fernandez. The sparse matrix vector product on gpus. In *Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering*, volume 2, pages 1081–1092, 2009.

[71] O. Villa, D. Chavarria-Miranda, and K. Maschhoff. Input-independent, scalable and fast string matching on the cray xmt. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[72] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127 –134, 2010.

[73] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009.

[74] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[75] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 255–268, 2014.

[76] Xilinx. *Virtex-6 Family Overview*, 2012.

[77] Xilinx. *Vivado Design Suite User Guide High-Level Synthesis*, 2012.

[78] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos. Fpga vs. gpu for sparse matrix vector multiply. In *International Conference on Field-Programmable Technology*, pages 255 –262, 2009.

[79] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based esl synthesis system. *High-Level Synthesis: from Algorithm to Digital Circuit*, 2008.

[80] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 63–74, 2005.