UNIVERSITY OF CALIFORNIA
RIVERSIDE

Hashing, Caching, and Synchronization: Memory Techniques for Latency Masking Multithreaded Applications

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

Doctor of Philosophy

in

Computer Science

by

Skyler Arron Windh

September 2018

Dissertation Committee:

Dr. Walid A. Najjar, Chairperson
Dr. Vassilis Tsotras
Dr. Nael Abu-Ghazaleh
Dr. Daniel Wong

The Dissertation of Skyler Arron Windh is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Looking back on the last six years of my Ph.D., it has been a time of tremendous growth. Both personally and professionally. And that growth comes from both great accomplishments and deep, deep upsets. I am forever grateful to my advisor Dr. Walid Najjar for the man he has been in my life. He has been a constant source of wisdom both technical and personal. He has given my work direction and kept me steady when life gave me punches.

I would also like to thank Dr. Vassilis Tsotras for his assistance on my work the last few years. He has been very helpful in editing papers and asking questions that keep projects in scope. I also thank my other committee members, Dr. Nael Abu-Ghazaleh and Dr. Daniel Wong for their time serving on my committee and for asking good questions that lead to exploring different avenues in this research.

During graduate school, I learned as much about life and personal skills as I did technical. Thank you Steven Jacobs for jumping into grad school with me because why not. I'm grateful to all the friends I have made in the Embedded Systems lab and all the help I have received. Thanks to Robert Halstead who showed me the ropes with FPGAs and left a lot of fertile research for me to pursue. Prerna Budhkar and I have spent many hours hashing out details of this multithreaded model and how it would work in many different applications. Her patient, hard-working demeanor is inspiring and a goal I hope to reach one day. Jose Rodriguez has been a good friend and colleague, always willing to let me bounce ideas off of him and see what solutions fall out. And my good friend Jason Ott, who spent many hours with me discussing faith, physics, fatherhood, and so many other random topics. Thanks for the encouragement and friendship.

Finally, this thesis contains two of my previously published works from my research. The first explores HLS tools and was published in The Proceedings of the IEEE. I want to thank the reviewers and publishers for considering the work as well as my co-authors for the help in bringing it all together. It's content appears in Chapter 3.The full citation and author list is as follows:

Skyler Windh, Xiaoyin Ma, Robert J Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level Language Tools for Reconfigurable Computing. Proceedings of the IEEE, 103(3):390-408, 2015.

The other paper was published at ICCAD in 2015 and applied our multithreaded model to breadth first search. It's content is the basis for Chapter 4. The full citation is:

Skyler Windh, Prerna Budhkar, and Walid A. Najjar. CAMs as Synchronizing Caches for Multithreaded Irregular Applications on FPGAs. ICCAD'15, pages 331-336, 2015.

To my parents for all the years of support and encouragement.

To my amazing wife Melanie, thanks for the never-ending support through the hardest parts. I would not have finished without you and your constant selflessness.

To my children. May this always encourage you to work hard and persevere.

ABSTRACT OF THE DISSERTATION

Hashing, Caching, and Synchronization: Memory Techniques for Latency Masking Multithreaded Applications

by

Skyler Arron Windh

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Walid A. Najjar, Chairperson

The increase in size and decrease in cost of DRAMs has led to a rapid growth of in-memory solutions to data analytics. In this area, performance is often limited by the latency and bandwidth of the memory system. Furthermore, the move to multicore execution has put added pressure on the memory bandwidth and often results in additional latency.

Irregular applications, by their very nature, suffer from poor data locality. This often results in high miss rates for caches and many long waits to off-chip memory. Historically, long latencies have been dealt with in two ways: (1) latency mitigation using large cache hierarchies, or (2) latency masking where threads relinquish their control after issuing a memory request. Multithreaded CPUs are designed for a fixed maximum number of threads tailored for an average application. FPGAs, however, can be customized to specific applications. Their massive parallelism is well known, and ideally suited to dynamically manage hundreds, or thousands, of threads. Multithreading, in essence, trades memory bandwidth for latency.

This thesis describes the use of CAMs (Content Addressable Memories) as synchronizing caches for hardware multithreading. We demonstrate and evaluate this mechanism by implementing multithreaded datapaths for Breadth First Search, Hash-Join, and Group-By Aggregation. Synchronization between concurrent threads is typically implemented using expensive in-memory locks that are accessed via atomic operations. CAMs allow us to move the lock on chip, increase the multithreading, and achieve better performance.

Contents

List of Figures

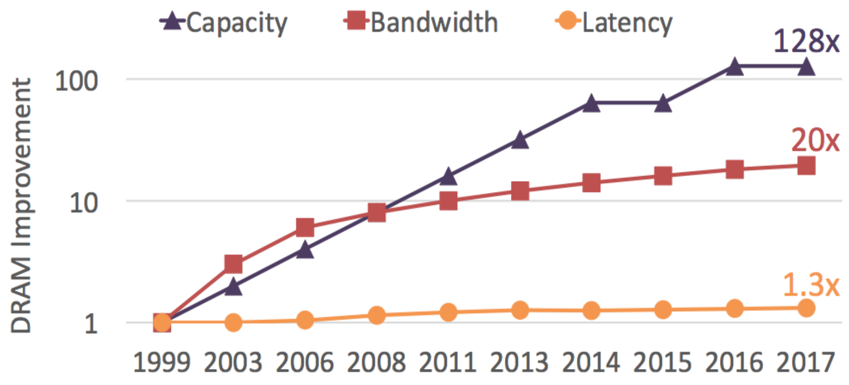
List of Tables

Chapter 1

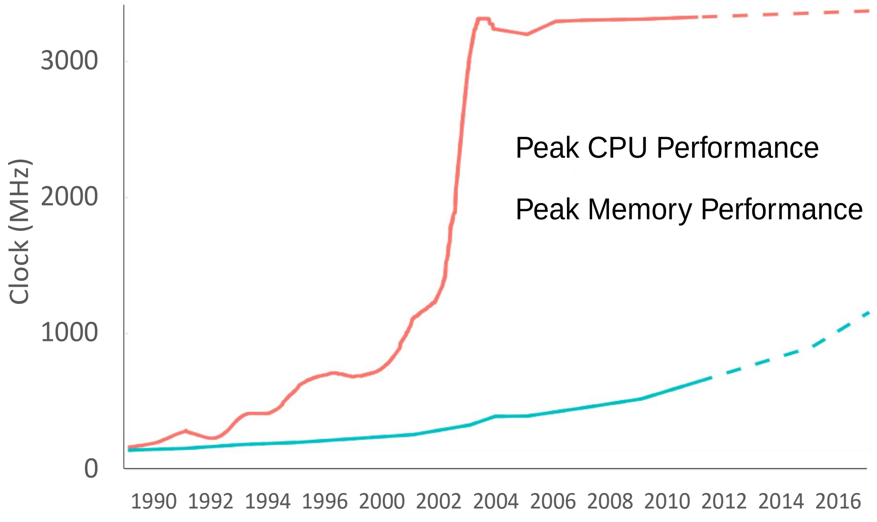
Introduction

Fast database analytics over large collections of customer data is key concern for any modern business, looking to get an edge over its competitors. In recent years many companies have sprung up offering their own in-memory database solutions. Oracle's Exadata [106], and Pivotal Software's Greenplum [60] have built custom machines for memory intensive workloads. On the other hand IBM's Netezza [75], and Teradata's Kickfire [81] approached the problem using off the shelf reprogrammable hardware, i.e. integrated FPGA boards.

The proliferation of these solutions is to try and solve the problem of the giant gulf between our data and compute. In the Big Data era, we are producing data at tremendous rates and our compute has gotten faster than it has ever been. Yet, we still cannot feed the compute cores fast enough for all algorithms. Despite the progress made in multi-core architectures, the major performance limitations come from the memory latency (known as the *memory wall* figure 1.1b), that restricts the scalability of such memory-bounded



(a) DRAM capacity and bandwidth improve at the cost of latency.



(b) The "Memory Wall" between CPU and memory performance.

Figure 1.1: The Memory Wall describes the increasing gap between CPU performance and memory performance. As this latency gap grows, any irregular application pays a higher and higher performance cost on modern processor architectures. Figures from [38]

algorithms. This performance bottleneck is evident in the historical trends in Figure 1.1a. Over the last 20 years, capacity and bandwidth have followed the scaling of Moore's law, improving 128x and 20x respectively. However, latency has only improved about 30% since 1999.

A memory access can take anywhere from 100 to 200 CPU cycles - equivalent to the execution of 100s of instructions. The most common solution to the memory latency problem is the use of extensive *cache hierarchies* that occupy up to 80% of a typical processor die area. This latency mitigation approach relies on data (spatial and temporal) and instruction localities. *Multithreaded execution* [85, 122] has been proposed as an alternative approach that relies on the masking of memory latency by switching to a ready but waiting thread when the currently executing thread encounters a long latency operation, such as a main memory access. Several different multithreading models (simultaneous, fine-grained temporal, coarse-grained temporal) have been proposed along these. They can be distinguished by how close in time instructions from different threads may be executed. In this thesis, the focus will be on coarse-grained temporal multithreading.

Despite wide acceptance of caching as a latency mitigation method of choice this thesis explores an alternative approach by supporting multiple outstanding memory requests from various independent threads. This multithreading architecture is implemented on FPGAs and optimized to process relational database workloads. The design is similar to the multithreading approach used in the SUN UltraSPARC architecture (for example, the UltraSPARC T5 [56] can support eight threads per core and 16 cores per chip). However, because our FPGA implementation is able to support deeper pipelining and custom threads with extremely small contexts, it can maintain thousands (instead of tens) of outstanding memory requests and hence drastically increases concurrency and therefore throughput. Furthermore, the multithreaded execution maximizes the utilization of the available memory bandwidth.

Adding to the complications of achieving performance is the proliferation of multi-core, many-core, and highly parallel hardwares. No longer is it good enough to simply write efficient single threaded CPU code. If programmers aren't targeting parallelization, they miss out on the resources that modern architectures are providing. The next generation CPU release will most likely have more parallelism advancements than any clock frequency gains. In the world of FPGAs, implementing a single compute engine is not enough for any real performance. Designs need to replicate and use as much area as possible. Parallelization is the way to win modern performance.

In this dissertation, we will look at all of aforementioned issues. We will start by looking at the growth of tools in the FPGA space that try to extract parallel solutions from High Level software code. And we'll also look at the limitations of these tools Then we show how to exploit this massive multithreading to break through the memory wall and provide performance for irregular applications. We also describe the use of CAMs (Content Addressable Memories) as synchronizing caches for hardware multithreading. We demonstrate and evaluate this mechanism by implementing multithreaded datapaths for Breadth First Search, Hash-Join, and Group-By Aggregation. Synchronization between concurrent threads is typically implemented using expensive in-memory locks that are accessed via atomic operations. CAMs allow us to move the lock on chip, increase the multithreading, and achieve better performance. Given their irregular memory nature, these algorithms incur poor spatial locality, thus traditional CPU approaches rely on vast caches to attempt to alleviate the latency penalty. Since an FPGA takes an alternative approach it requires massive parallelism to compete with the CPU's order of magnitude faster clock frequency.

Chapter 2

Background

This thesis focuses on irregular algorithms for data analytics. This section presents some general background needed to understand the algorithms that will be implemented. We first give a general overview of multithreading since it is such an overloaded word with several architectural meanings. After covering the models we discuss several architectures that implement each style. We then cover the MT-FPGA execution model and how we implement our hybrid of SMT and coarse-grained temporal multithreading. Finally we'll cover the growth of FPGA heterogeneous computing, the Convey HC-2EX architecture, implementing CAMs on FPGAs, and finally the use of FPGAs for database query processing.

2.1 Multithreaded Architectures

The goal of a processor is to fully utilize all hardware resources in any given cycle. However, many times instructions have data dependencies and they cannot execute until all dependencies are met. Compilers can do some work reordering instructions to

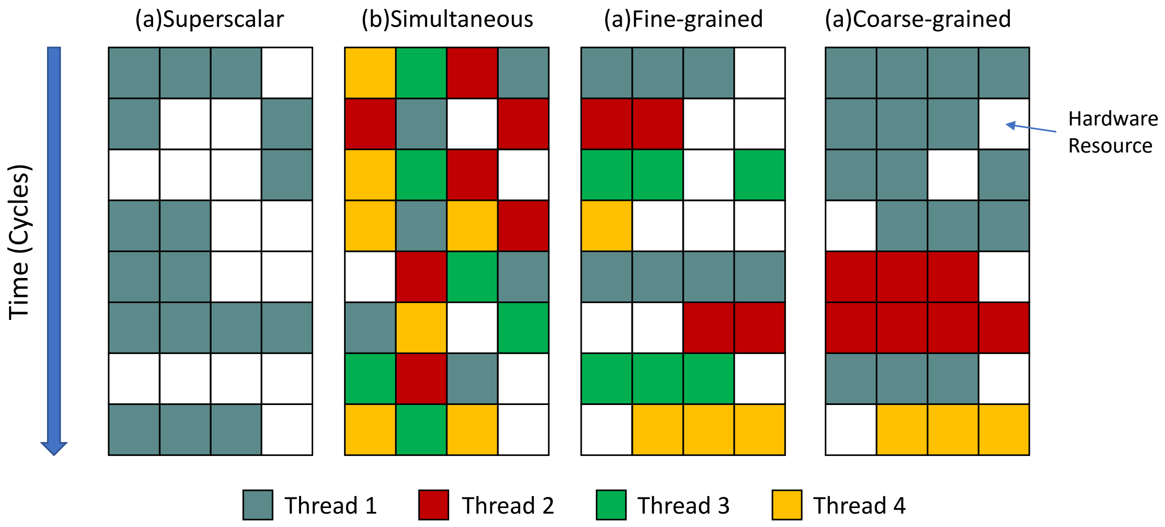


Figure 2.1: Various multithreading models

minimize delay slots in a stream of instructions. And the hardware in the processor can further expand on this by issuing instructions out of order and letting several instructions try to acquire as many dependencies concurrently as possible. However, even with all this advanced instruction scheduling there are often still delay slots to be found.

Multithreaded execution tries to get around this problem by filling slots with instructions from independent threads. Figure 2.1 shows schedules under different multi-threading techniques. In the traditional superscalar processor(figure 2.1a), several instructions in a single thread are scheduled to fill available issue slots. Some cycles will be able to use all resources, but sometimes pipeline bubbles are inevitable. Many modern processors with out-of-order superscalar architectures like the IBM Power and Intel Skylake feature support for simultaneous multithreading (SMT) (figure 2.1b). SMT maintains multiple independent threads active in a given core and will take instructions from each thread and issue them to execution units as available. Since each thread is maintained in an active state, there is the benefit that there is no context switching. However, it comes at the cost of increased complexity in the hardware and increased pressure on hardware resources. Each thread may be in a completely different address space and they may keep thrashing local cache and register resources. Extracting peak performance requires diligent effort on the programmer to make sure threads are grouped together wisely.

2.1.1 Temporal Multithreading

In contrast, the fine-grained threading model(figure2.1c) is rather simple. The processor is built around the concept that every thread will be swapped out in a fixed interval, e.g. every cycle. Then, you can hide delay slots by having as many threads as the longest stall in the system. If the current instruction causes a stall, it doesn't matter because the processor is swapping in a new thread and this thread won't be executing for n cycles and the data dependency will already be filled. This model provides high throughput since it is always rotating threads and also provides deterministic, easy to understand behavior. However, this comes at the expense of individual thread performance. Each thread is getting kicked out even if the current instruction won't cause a stall. In fine-grained multithreading with n threads, Compute heavy code with m stall free instructions will take $n * m$ cycles to execute instead of the m cycles it would take on a traditional pipeline.

Finally, the coarse-grained multithreading model (figure2.1d) tries to take the simpler model of fine-grained and provide better single threaded performance. Instead of swapping threads out in fixed intervals, coarse-grained multithreading only swaps threads out when an instruction triggers a long-latency operation (e.g. memory request). In this case, a thread can continue to make progress while it has instructions to execute. When it needs to wait for data, it can wait in buffers while other threads progress. Single-threaded performance is restored, however, it is at the cost of non-determinism and higher conflicts in shared resources. Coarse-grained multithreading must also maintain enough threads to hide the latency of the requests.

Both fine-grained and coarse-grained multithreading fall in a class called temporal multithreading because they have the idea of using multithreading to hide the time of long-running operations.Because of this, both have the property that they need significantly more threads than the SMT model. SMT threading needs a number of threads relative to the amount of execution units in the core. The other two models require a number of threads relative to the length of the memory latency. And as we saw in the introduction, the memory wall has been growing for decades, continually pushing up the latency. Between the two classes there is a trade-off of less threads and complicated cores for SMT, and order of magnitude more threads but simpler cores for temporal.

2.2 Latency Masking Multithreaded Architectures

In the early 90s the Tera Corporation, built the Tera MTA. The MTA design consisted of 256 processors sharing 64 GB of memory organized as a distributed NUMA architecture. Its interconnection network allowed better scaling to a larger number of processors. It also forced instruction requests through a shared cache lowering the network traffic. Custom processors supported the issuing of one memory request per thread per cycle. The maximum memory latency from any processor to any memory module was 128 cycles. Each processor could support up to 128 active threads. The MTA design [18] was later evolved into the Cray XMT. While the MTA had only 256 processors the XMT machine could support up to 8,192 processors, but the largest ones built had 512 processors. The shared memory was also increased from 1 TB to 128 TBs for the MTA, and the clock speed was improved from 220 MHz to 500 MHz.

2.3 MT-FPGA Execution Model

MT-FPGA (Multithreading on FPGAs) [66] is an execution model that combines the memory masking ability of multithreaded execution with a customized data path. This

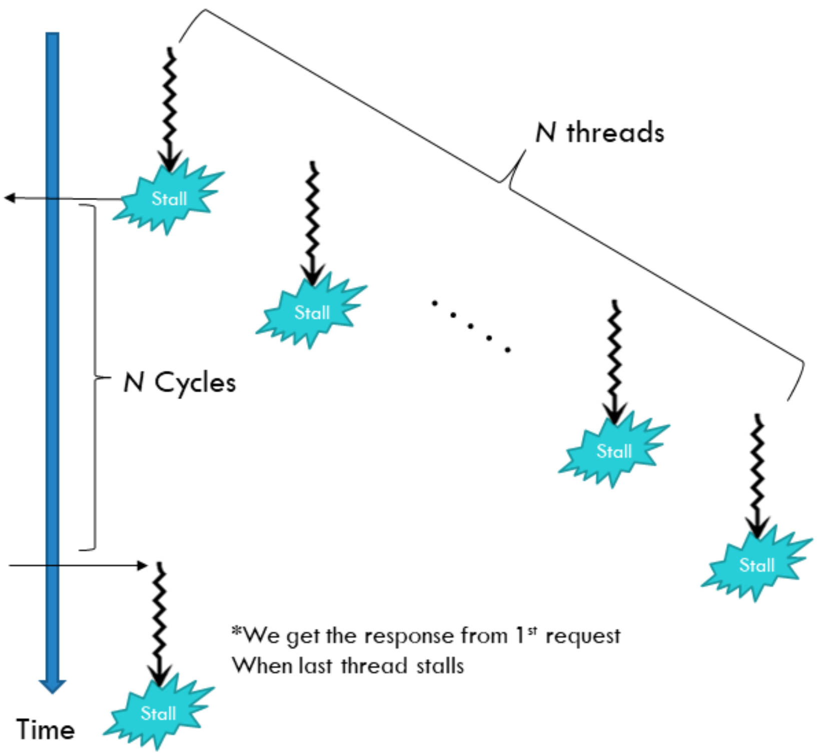


Figure 2.2: Hiding Memory Latency

execution model suspends the thread as soon as it performs a long latency read and a waiting ready thread is given the chance to execute. It performs decoupling by buffering the returned data in the order it was requested. This execution model exhibits following advantages:

1. Can support hundreds of outstanding memory requests, hence massive parallelism
2. Full utilization of the datapath
3. The state of the thread is extremely small, and can therefore cater to more number of pending threads, stored on a FIFO.

A recent tool called CHAT uses the MT-FPGA model. This compilation tool generates customized hardware support for multithreaded execution on FPGAs and claims to ease the hardware development effort for complex irregular kernels. Using just one accelerator FPGA, CHAT shows a speed-up of up to 50x over a single Intel Xeon on simple irregular kernels. Similarly, a database hash-join system demonstrated in [64] is also designed using a MT-FPGA model. Throughput results show a speedup between 2x and 3.4x over the best multi-core approaches with comparable memory bandwidths on uniform and skewed datasets.

Designing an engine that supports enough threads to hide latency like Figure 2.2 follows a simple model. The high level idea of the model can be seen in Figure 2.3. The compute can be viewed as the datapath block. A thread continues to execute until it needs memory. At this point, the thread will issue the request and move itself into the *Waiting threads* FIFO. Because the hardware guarantees in memory responses, we know that the next response must belong to the thread at the front of the FIFO. A memory response will add the data to the threads state and the thread will move to the *Ready threads* FIFO and

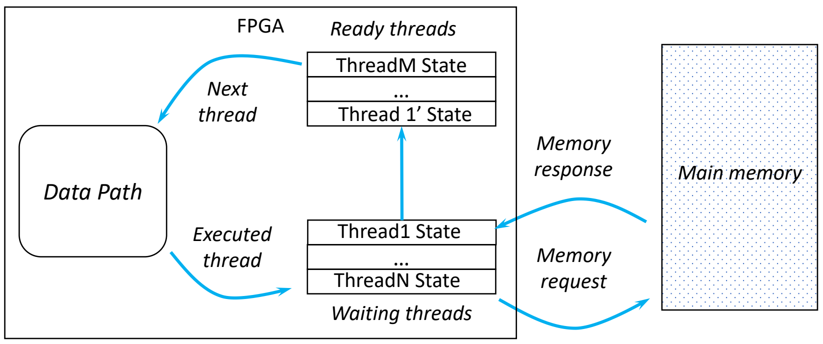
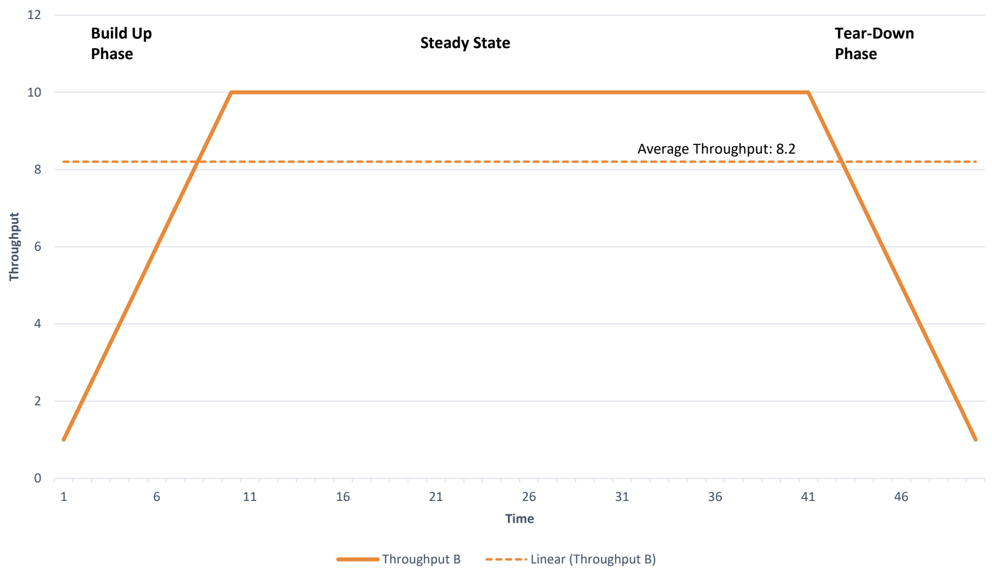


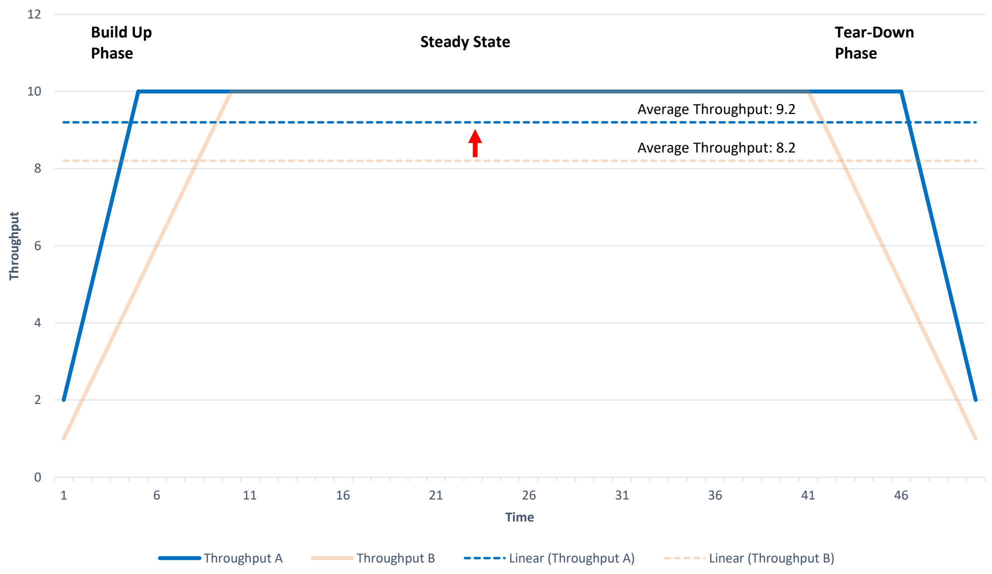
Figure 2.3: MT-FPGA Architecture Model

wait until the datapath reactivates the thread. Since these datapaths are pipelined, we can activate a thread every cycle as long as there are no memory threads. And, the pipelined datapath means we can design the compute only thinking about one thread moving through each state. We don't need to worry about synchronization unless there are shared resources. And in that case, we can use a CAM, as this thesis will show later.

The achievable throughput in this model is demonstrated in Figure 2.4. In Figure 2.4a shows the 3 states of the MT-FPGA model states: Build-up, Steady, and Tear Down. In the Build-up state, threads are getting started and begin issuing requests. Since we do not yet have enough threads making requests, the throughout will be less than the peak bandwidth. Once execution has fully started we transition to the steady state. In the steady state, threads issue requests every cycle and memory is responding with new data every cycle. In this state the throughput is exactly the peak bandwidth since memory is working every cycle. As execution begins to wind down, we transition to the tear down state. Threads finish their work and terminate and we no longer have enough threads to mask latency. In this state, throughput begins dropping as we see more latency. The overall throughput of the full execution is simply the average throughput of all three stages.



(a) Three stages of multithreading



(b) Average throughput increases with longer steady state

Figure 2.4: Execution in the MT-FPGA model has 3 states: Build-up, Steady, and Tear Down. In the build up phase, threads are just getting started and issuing requests so we are not yet masking latency. In the steady state, a thread issues a request every cycle and we receive a response every cycle. in the Tear-Down stage, we start losing masking as threads terminate. The overall throughput is the average of these three stages

Figure 2.4b shows why this simple model works so well. As we increase the time the system stays in the steady state, our average throughput will continually increase and approach the full system bandwidth. And the more data the system needs to process, the easier it is to maximize the time in the steady state. This model pairs well with the continued growth of the Big Data era where queries are now running on Terabytes of data. The other benefit of this model is that it works independent of regular or irregular memory access patterns. As long as the engine issues requests every cycle, the throughput will approach the peak bandwidth of the system.

2.4 FPGA Heterogeneous Computing

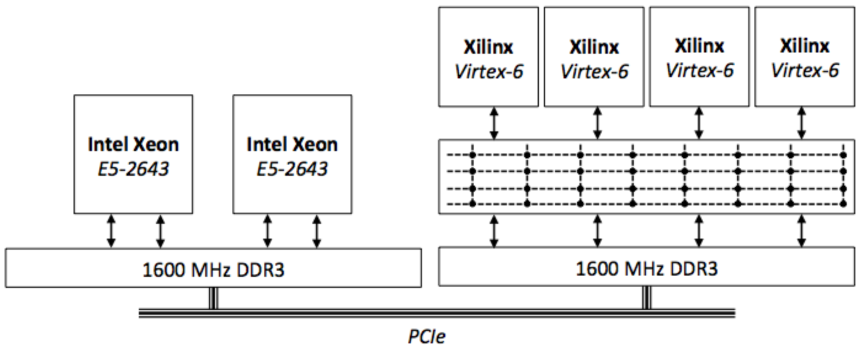
FPGA architectures have evolved from their early stages (supporting simple 8-bit logic operations), into large logic arrays capable of concurrently executing multiple complex instructions. They have historically been used as off-chip accelerators where CPUs can offload compute intensive workloads, and read back the results. However, in recent years the FPGA has been trending closer and closer to the CPU. Xilinx currently offers a Zynq [136] line of chips that couples the FPGA's reconfigurable fabric with an ARM processor, and Intel's recent acquisition of Altera suggests this trend will continue. However, it is currently still more common to see the FPGA connected with the CPU over a PCIe bus. Microsoft Research incorporated multiple Stratix-V FPGAs into a 48 node server that was used to accelerate the Bing search engine [110]. Alpha Data announced a CAPI environment, which allows Xilinx All Programmable devices to connect with IBM Power8 architectures [11]. Convey Computers, Maxeler Technologies, and Pico Computing are all companies currently offering FPGA platforms over PCIe, which have been actively used in the research community to show acceleration and power savings compared to CPUs and GPUs [64, 37].

Different FPGA architectures are optimized for various use cases: HPC computations, database workloads or packet processing. The designs proposed in this paper utilize only the off-chip memory interface, which makes them general enough to be ported between most currently available FPGA platforms. For simplicity we choose only one platform, the Convey-WX, to implement and run all our designs. The Convey architecture offers a shared global memory space between hardware and software, which eliminates any variability due to the memory architecture and allows us to do direct performance comparison.

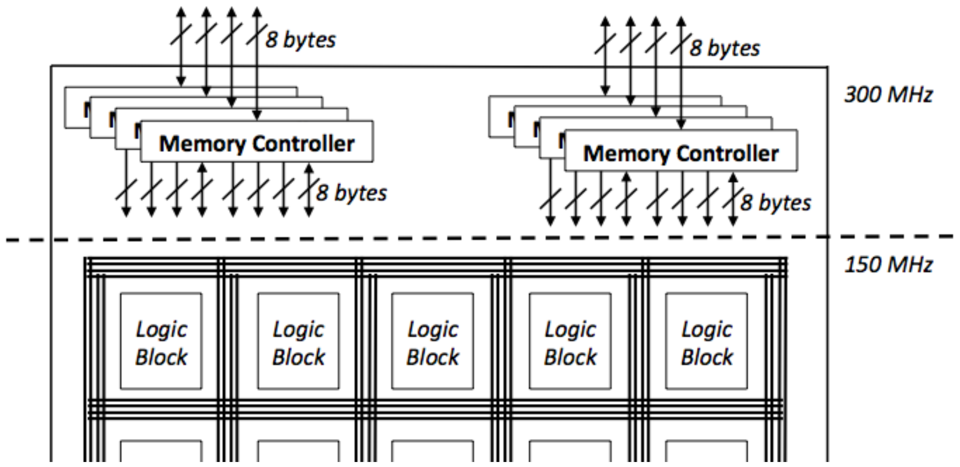
2.5 Convey HC-2ex Platform

The Convey HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions. As shown in Figure 2.5a the memory is divided into regions connected through PCIe with portions closer to the CPU, and portions closer to the FPGAs. The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache. In total the software region has 128 GB of 1600 MHz DDR3 memory. The system has a peak memory bandwidth of 51.2 GB/s.

The hardware region has 4 Xilinx Virtex6-760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300MHz (Figure 2.5b). 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. These



(a) The Convey HC-2ex software and hardware regions.



(b) Convey HC-2ex FPGA AE wrapper.

Figure 2.5: The Convey HC-2ex architecture. Separation into software and hardware regions in shown in (a). In hardware region each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b).

memory channels provide a highly parallel 8,192 simultaneous outstanding requests. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s.

2.6 CAMs on FPGA

A CAM (also known as an associative memory), is an array that can perform efficient entry-matching (i.e. answer membership queries). Its operation is the inverse of a Random Access Memory (RAM): when presented with a *search word* the CAM returns all the locations whose content matches that word. Each CAM bit consists of a flip-flop with a comparator matching it to the corresponding bit in the search word. The outputs of all the bit positions in a word are ANDed to generate the (mis)match for that word. The CAM's ability to perform a search in unit time comes at a high cost of area, energy and long clock cycle time (due to the long wires for the bit-wise AND and propagating the search word to all the entries)

As the number of entries in the CAM increases, the achievable clock frequency of the circuit drops. This limitation either restricts the size of the CAM or increases the number of cycles it takes to perform an update operation. Nonetheless, CAMs have proven to be very useful in domains such as networking (e.g. implementing an IP table in a network router). Recently we explored how CAMs can be used to accelerate the breadth first search algorithm [131].

In a streaming environment CAMs can maintain a cache of recently seen unique items and allow quick access to them without stalling the pipeline. This fast cache look-up mechanism can also be used as a fine-grained address-based synchronization primitive, which avoids long latency trips to main memory and does not require special hardware. Consider the case when a CAM is assigned to guard a particular memory partition. It can be configured to hold the addresses of the values that need synchronized access. If all memory requests within a partition are first submitted to the CAM, before being routed to the memory, the accesses to identical addresses are serialized locally in the CAM. In this case a CAM entry serves as an exclusive lock, which gets released (flushed from the CAM) after the request(s) completion. In [8] we discuss how to use this approach for synchronization in the multithreading group-by aggregation algorithm.

To the best of our knowledge all previous FPGA implementations relied on specialized platform features to provide synchronization primitives. In our previous work [64] we used atomic operations provided by the now discontinued Convey MX architecture [45]. Each word in memory maintains a locking bit that can be set by a memory instruction. The Convey development kit provides test and set instructions that can be executed from the FPGA to lock the memory location while the operation is executed. Leveraging CAMs for synchronization increases the portability of our design. This moves all synchronization operations internal to the FPGA, and can be done on any architecture. In addition, this design provides more selective fine-grained synchronization primitives in comparison to the Convey-MX, which places a lock on all FPGA-memory communication channels.

It was shown that implementing fully-associative matching logic for CAMs on both Altera and Xilinx FPGAs introduces a 60x overhead compared to regular BRAMs [139]. This drawback makes implementing large CAMs on reconfigurable fabrics notoriously hard. Dhawan et al. [51] explored various designs of CAMs and introduced a trade-off between CAM size and update time.

2.7 Query Processing on FPGAs[1]

Many research works in the early 1980s were dedicated to the design and architecture of *database machines* - specialized hardware, designed solely for the purpose of storing and processing large amounts of data. These architectures were utilizing parallel data processing by tightly coupling processing units with disk-based storage. The stagnated growth of disk bandwidth coupled with the continuous increase of storage density implied that data management systems were mostly IO bound. At the same time the rapid performance advances of off-the-shelf processors (due to Moore's law) made the database machine very cost-ineffective [27]. This allowed a handful of processors to operate on a large number of parallel disk I/O operations thus avoiding the rigid pairing of storage and compute units. The interest gradually shifted from intra-node database machine-style parallelism to shared-nothing systems, providing effective easy to scale inter-node parallelism [50, 49]. The depletion of the processing frequency growth finally discontinued the "free ride" on performance scaling. Abundance of cheap main memory diminished the role of I/O-related overhead as a main bottleneck. Nevertheless, the growing gap between memory access latency and the processor's computational capabilities ("memory wall") brings up the data access overhead, but on a different level ("memory is the new I/O"). At the same time, the limited bandwidth of current network technology has restricted the scaling potential of the shared-nothing systems. The aforementioned hardware trends as well as The availability of new generation of data processing hardware (GPUs, FPGAs, ASICs) revived the interest in specialized hardware-accelerated database systems. Recently several research projects

[1]This section worked on collaboratively with colleague Prerna Budhkar

proposed building hybrid CPU-GPU systems [141, 29, 61, 126, 58]. These systems are deployed on a traditional CPU architecture, but use the GPUs as a co-processor to accelerate easy-to-parallelize parts of the query processing.

Several academic projects have worked towards simplifying the use of custom hardware for query processing. For instance, the Glacier library [101] implements a component library that generates query-specific FPGA circuits for various streaming queries. This approach is suitable for scenarios with few queries that are known in advance. Queries that fall under typical stream processing applications run longer which justifies invoking a time-consuming synthesis process for every new query. 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 [115]. The Q100 [134] 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 [75] 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 [119]. 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 [81], uses FPGA hardware accelerators connected through either PCIe or hyper transport. It defines various database operations as HARP logic that consists of a hardware circuit and a large memory systems. All queries are analyzed by Teradatas 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 [28, 71, 94]

Chapter 3

High Level Language Tools for Reconfigurable Computing

This first chapter looks at the flexibility of High Level Synthesis tools. The goal of these tools is to take an algorithm description in a high level language like C/C++ and automatically generate a synthesizable hardware circuit. They provided automated testing facilities for both software and hardware models to try and improve the speed of iterating on hardware design. The hardest, most expensive part of FPGA development tends to be the engineering hours contributed to making and testing the design. Any progress in bringing down development time is extremely beneficial. An insightful webcomic (figure 3.1) on this idea is called "Compiling" by Randall Munroe [137]. As true as waiting for compilation is in the software world, it is painfully more true with hardware. With several projects over the years that took more than 24 hours to build for the FPGA, sometimes creativity wins over productivity.

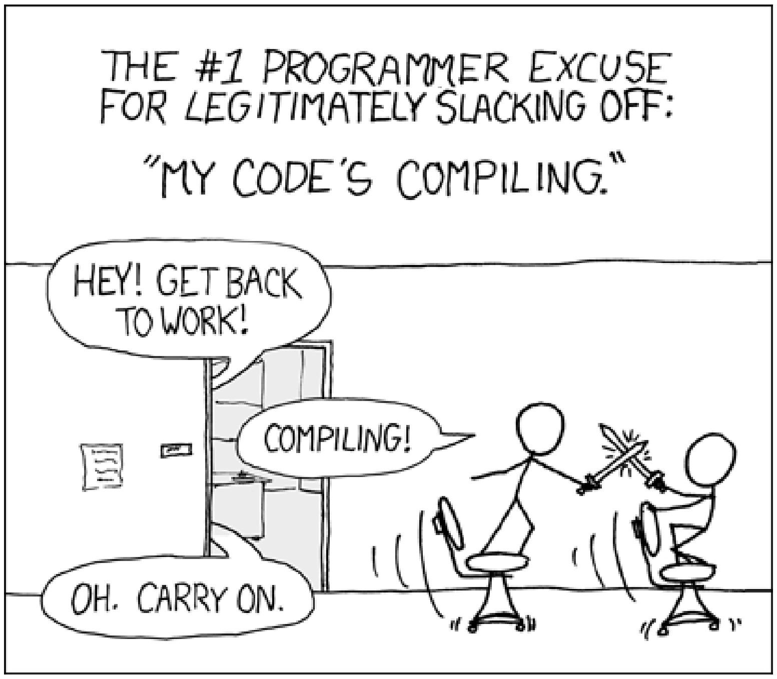


Figure 3.1: xkcd: Compiling - sometimes there is no other option than jousting in the lab.[137]

While the tools are making tremendous progress, they are not a silver bullet. They generally only work on a subset of the high level language, at the time of this research they only focused on regular compute cores, and they did not support dynamic memory. In this project[132], we look at the tools from the programmer's perspective. How easy it is to describe a design, how easy it is to explore different hardware configurations, and how much of the process to tool automates.

3.1 High Level Synthesis and FPGAs

In recent years we have witnessed a tremendous growth in size and speed of FPGAs accompanied by an ever widening spectrum of application domains where they are extensively used. Furthermore, a large number of specialized functional units are being added to their architectures such as DSP units, multi-ported on-chip memories and CPUs. Modern FPGAs are used as platforms for configurable computing that combine the flexibility and re-programability of CPUs with the efficiency of ASICs. Commercial as well as research projects using FPGA accelerators on a wide variety of applications have reported speed-up, over both CPUs and GPUs, ranging from one to three orders of magnitude as well as reduced energy consumption per result ranging from one to two orders of magnitude. Application domains have included signal, image and video processing, encryption/decryption, decompression (text, integer data, images etc), data bases [98] [99], dense and sparse linear algebra, graph algorithm, data mining, information processing and text analysis, packet processing, intrusion detection, bioinformatics, financial analysis, seismic data analysis, etc.

FPGAs are programmed using Hardware Description Languages (HDLs) such as VHDL, Verilog, SystemC and SystemVerilog that are used for digital circuit design and implementation. In these languages the circuit to be mapped on an FPGA is designed at a fairly low level: the data paths and state machine controllers are built from the bottom up, timing primitives are used to provide synchronization between signals, the registering of data values is explicitly stated, parallelism is implicit and sequential ordering of events must be explicitly enforced via the state machine. Traditionally trained software developers are not familiar with such programming paradigms. Beyond the program development state, the tool chains are language and vendor specific and consist of a synthesis phase where the HDL code is translated to a netlist, mapping where logic expressions are translated into hardware primitives specific to a device, place and route where hardware logic blocks are placed on the device and wires routed to connect them. This last phase attempts to solve an NP-complete problem using heuristics, such as simulated annealing, and may take hours or days to complete depending on the size of the circuit relative to the device size as well as the timing constraints imposed by the user. The steepness of the learning curve for such languages and tools makes their use a daunting and expensive proposition for most projects.

This paper provides a qualitative survey of the currently available tools, both research and commercial ones, for programming FPGAs as hardware accelerators. We start with a historical prospective on the use of FPGA-based hardware accelerators (Section 3.1.1) showing that the role of FPGAs as accelerators emerged very shorty after their introduction, as glue-logic replacements, in the 1980s. In Section 3.1.2 we discuss the efficiency of the hardware computing model over the stored program model and review the challenges posed by using High-Level Languages (HLLs) as programming tools to generate hardware structures on FPGAs. Related works and five High-Level Synthesis (HLS) tools are described in Section 3, three commercial tools: Xilinx Vivado, Altera OpenCL, Bluespec BSV, and two university research tools: ROCCC and LegUp. We use a simple image filter, dilation, and AES encryption routines to describe the style of programming these tools and explore their capabilities in implementing compiler-based transformations that enhance the throughput of the generated structure (Section 3.3 and Section 3.4). Area, performance and power results for both benchmarks are compared in Section 3.5 and, finally, concluding remarks are presented in Section 3.6. Note: in this paper we report results and compare tools only to the extent allowed by the terms of the user license agreements.

3.1.1 A Historical Perspective

The use of FPGAs as hardware accelerators is not a new concept. Very shortly after the introduction of the first SRAM-based FPGA device (Xilinx, 1985) the PAM (Programmable Active Memory) [22][127] was conceived and built at the DEC Paris Research Lab (PReL). The PAM P_0 consists of a 5x5 array of Xilinx XC3020 FPGAs (Figure 3.2) connected to various memory modules as well as to a host workstation via a VME bus. It had a maximum frequency of 25MHz, 0.5 MB of RAM, and communicated on a host bus of 8 MB/s. The PAM P_1 was built using slightly newer FPGA, the Xilinx XC3090. It operated with a maximum frequency of 40MHz, 4 MB of RAM, and used a 100 MB/s host bus. It was described as "universal hardware co-processor closely coupled to a standard host computer"[23]. It was evaluated using ten benchmark codes [23] consisting of: long multiplication, RSA cryptography, Ziv-Lempel compression, edit distance calculations, heat and Laplace equations, N-body calculations, binary 2D convolution, Boltzman machine model, 3D graphics (including translation, rotation, clipping and perspective projection) and discrete cosine transform. It is interesting to note that most of these benchmarks are still today subjects of research and development efforts in hardware acceleration. Berlin et al. in [23] conclude that PAM delivered a performance comparable to that of ASIC chips or supercomputers, of the time, and was one to two orders of magnitude faster than software. They also state that because of the PAM's large off-chip I/O bandwidth (6.4 Gb/s) it was ideally suited for "on-the-fly data acquisition and filtering," which is exactly the computational model, streaming data, adopted by most of the hardware acceleration projects that rely on FPGA platforms.

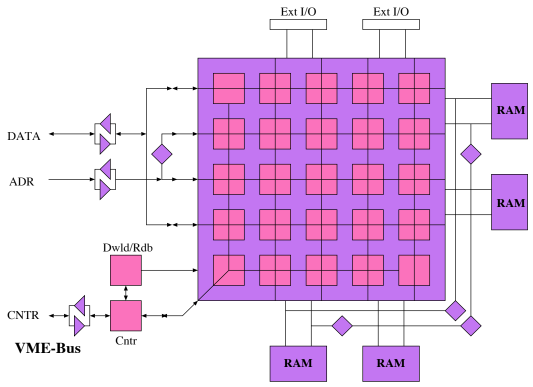


Figure 3.2: Architecture of the DEC PReL PAM P_0

This first reconfigurable platform was rapidly followed by the SPLASH 1 (1989) and SPLASH 2 (1992) [112, 30, 111, 73, 109] projects at the Supercomputer Research Center. Each were linear arrays of FPGAs with local memory modules. They were designed for accelerating string-based operations and computations such as edit distance calculations. The SPLASH 2 was reported to achieve four orders of magnitude speedup, over a SUN SPARC 10, on edit distance computation using dynamic programming.

Table 3.1: Architecture parameters of the SPLASH 1 and SPLASH 2 accelerators

	SPLASH 1	SPLASH 2
Year	1989	1992
FPGA	XC3090	XC4010
4-LUT/board	10,240	13,600
Max. bandwidth	1 MB/s	100 MB/s
Memory/FPGA	128 KB	512 KB
Interconnect	Linear array	Linear array, broadcast

The PAM and SPLASH projects put the foundation of reconfigurable computing by using FPGA-based hardware accelerators. In the past two decades the density and speed of FPGAs have grown tremendously: the density by several orders of magnitude, the clock speed by just over one order of magnitude. Both of these projects could each be easily implemented on single moderately sized modern FPGA device. However, the main challenge to FPGAs as hardware accelerators, namely the abstraction gap between application development and FPGA programming, not only remains unchanged but has probably gotten worse due to increase in complexity of the applications enabled by the larger device sizes. FPGA hardware accelerators are still beyond the reach of traditionally trained application code developers.

3.1.2 Hardware and Software Computing Models

In this section we discuss two issues that define the complexity of compiling HLLs to hardware circuits: (1) the semantic gap between the sequential stored-program execution model implicit in these languages and (2) the effects of abstractions, or lack thereof, on the complexity of the compiler.

Efficiency and Universality

The stored program model is a universal computing model: it is equivalent to a Turing machine with the limitations on the size of the tape imposed by the virtual address space. It can therefore be programmed to execute any computable function. Hardware execution, on the other hand, is not universal unless it has an *attached* microprocessor. It is, however, extremely efficient. Consider an image filter applied on a 3×3 pixel window over a frame: the *forall* loop implemented in hardware can be both pipelined (let d be the pipeline depth) and unrolled as to compute multiple windows concurrently, let the unroll factor be k. In the steady state $d \times k$ operations are being executed concurrently producing k output results per cycle. On a CPU, the innermost loop of a typical image filter requires 20 to 30 machine instructions per loop body including nine load instructions. Assuming an average instruction level parallelism (ILP) of two, each result takes 10 to 15 machine cycles - which is the ratio of the respective clock speeds of CPUs and GPUs to FPGAs. However, that same loop can be replicated many times on the FPGA achieving a much higher throughput (at least an order of magnitude). Furthermore, the ability to configure local customized storage on the FPGA makes it possible to reduce the number of memory accesses, mostly reads, by reusing already fetched data resulting in a more efficient use of the memory bandwidth and lower energy consumption per task [90]. Hence the higher speedup or throughput observed on a very wide range of applications using FPGA accelerators over multi-cores (CPUs and GPUs). Further details on CPU efficiency for image filters are discussed in Section 3.3.1.

Semantics of the Execution Models

CPUs and GPUs are inherently stored-program (or von Neumann) machines and so are the programming languages used on these. Most of the languages in use today reflect the stored program paradigm. As such they are bound by its sequential consistency, both at the language and machine levels. While CPU and GPU architectures exploit various forms of parallelism, such as instruction, data and thread-level parallelisms, they do so circumventing the *sequential consistency* implied in the source code internally (branch prediction, out-of-order execution, SIMD parallelism, etc.), while preserving the appearance of a sequentially consistent execution externally (reorder buffers, precise interrupts etc.). The compiling of a HLL code to a CPU or GPU is therefore the translation from one stored program machine model, the HLL, to another, the machine's Instruction Set Architecture (ISA). In the stored program paradigm the compiler can generate a parallel execution only when doing so is provably safe. In other words when the record of execution can be *proved*, by the compiler, to be either identical or equivalent to the sequential execution. For example, in a single level *forall* loop, any interleaving of the iterations produces a correct result. Also, in a single threaded CPU execution the producer/consumer relationship is not a parallel construct since the semantics imply that all the data must be produced before any datum can be consumed. Hence all the data is stored in memory by the producer loop before the consumer loop starts execution.

A digital circuit, on the other hand, is inherently parallel, spatial, with distributed storage and timed behavior. HDLs (e.g. VHDL, Verilog, SystemC and Bluespec) are arguably the most commonly used parallel languages. In a digital circuit the producer/consumer relation is a parallel structure: the data produced is temporarily stored in a local buffer the size of which is determined by the relative rates of production and consumption. Furthermore, any implementation would be expected to include back pressure and synchronization mechanisms to (1) halt the production before the buffer is full and (2) stall the consumption when the buffer is empty to achieve a correct implementation. Buffering the data is not necessary when compiling individual kernels (e.g. stand-alone filters). However, it becomes a necessity, and often a major challenge, when compiling larger systems. Consider data streaming through a series of filters: buffers and back-pressure are necessary to

Table 3.2: Features and characteristics of Stored Program and Spatial computation models

	Stored Program	**Spatial Computing**
Storage & data access	Central, large, virtual address space. Multi-level caches	Distributed, small, physical. Streaming data. Limited caching. No virtual memory
Parallelism	Dynamic - separate ILP, DLP, TLP	Static - integrated ILP, DLP, TLP
Sequencing	Central, static, sequentially consistent	Data-flow, asynchronous
Data-Path	Pre-designed, one size fits all. Dynamic data dependencies	Customized, very deep pipelines. No dynamic data dependencies

hold the data between filters. Automatically inferring efficient buffering schemes without user assistance in the forms of pragmas or annotations is a major challenge.

Edwards [53] makes the case that C-based languages are not well suited for HLS. The major challenges described in the paper for C-based languages apply to most HLLs. These challenges are the lack of: (1) timing information in the code, (2) size-based data types (or variable bit length data types), (3) built-in concurrency model(s), (4) local memories separated from the abstraction of one large shared memory. While all these points are valid, the main attraction of C-based languages is familiarity. Most HLS tools using C-based languages provide workarounds for one or more of these obstacles as described in [53].

The abstraction and semantic gaps between the hardware and software computing models are summarized in Table 3.2. Translating a HLL to a circuit requires a transformation of the sequential to a spatial/parallel, with the creation of custom sequencing, timed synchronizations, distributed storage, pipelining, etc. The storage in the von Neumann model is abstracted in a single large virtual address space with *uniform* access time (in theory). The spatial model is better served with multiple local small memories. The parallelism in the von Neumann model can be dynamic: threads are created and complete relinquishing resources. In hardware every thread must be provisioned with resources statically. The software model relies on an implicit sequential consistency where all instructions execute in program order and no instruction starts before all the previous instructions have completed execution. The hardware execution is data flow driven.

Raising the abstraction level of FPGA programming to that of CPU or GPU programming is a daunting task that is yet to be fully completed. It is of critical importance in the programming of accelerators as opposed to the high-level design of arbitrary digital circuit, which is the focus of high-level synthesis. Hardware accelerators differ from general purpose logic design in one important way: the starting point of logic design is a device whose behavior is specified by a hardware description code implemented in a HDL such as VHDL, Verilog, SystemC, SystemVerilog, or Bluespec. The starting point of a hardware accelerator is an existing software application a subset of which, being frequently executed, is translated into hardware. That subset is, quasi by definition, a loop nest. Hopefully that loop nest is parallelizable and can therefore exploit the FPGA resources. By focusing on loop nests, the task of compiling HLLs to FPGAs is simplified and opportunities for loop transformations and optimizations abound. The ROCCC compiler takes this approach and is described later in this paper.

3.2 Related Work

As the number of tools supporting HLS for FPGAs has increased so has the number of surveys comparing and contrasting such tools. However, the rapidly shifting landscape of HLS tools for reconfigurable computing makes most endeavors obsolete within a few years. A description of the historical evolution of HLS tools, starting with the pioneering work in the 1970s can be found in [92]. The authors offer an interesting analysis of the reasons behind the successes and failures of the various generations of HLS tools. While the survey is not focused on HLS tools for FPGAs, it does mention several FPGA-specific tools, such as Handel-C, as well as general HLS tools that could be used for FPGAs.

The major research efforts in compiling high-level languages to reconfigurable computing are surveyed in [36]. The paper offers an in-depth analysis of the tools available at that time. AutoESL is described in [44]. The paper also provides an extensive survey of HLS in general and of tools specifically for FPGA programming. In [54] the authors reviewed six high level languages/tools based on programming productivity and generated hardware performance (frequency, area). User experience of using the targeted languages is recorded and normalized as a measure of productivity in this study. However, most of the tools evaluated in this work are no longer supported by their developers.

An extensive evaluation of 12 HLS tools in terms of capabilities, usability and quality of results is presented in [93]. The authors use Sobel edge detection to evaluate the tools along eight specific criteria: documentation, learning curve, ease of implementation, abstraction level, data types, exploration, verification and quality of the results. Daoud et al. [48] survey past and current HLS tools.

3.2.1 Xilinx Vivado HLS

Vivado High-Level Synthesis is a complete HLS environment from Xilinx. It has been in development for the last several years following Xilinx's acquisition of AutoESL[57] [7] [142]. Vivado HLS is available as a component of Xilinx's larger Vivado Design Suite or as a standalone tool. Like most HLS tools, Vivado HLS is mostly oriented towards core generation over full system design. It is possible to create hybrid designs with portions of code running on a soft-core processor communicating with custom hardware accelerators. However, the recommended work-flow [103] requires exporting the IP core from HLS and importing into the full Vivado Design Suite. As a Xilinx tool, there is significant support for different boards of multiple families of Xilinx FPGAs (7-series Virtex, Artix, Zynq, etc.). Depending on requirements, the hardware accelerator can be exported as one of several different Xilinx specific core formats for simple integration into other products, or just the HDL specification.

The Vivado HLS tool is built using LLVM [89][133] compiler framework. As such it has access to many software optimizations (e.g. loop-unrolling, loop-rotation, dead-code elimination, etc). However, hardware and software programing paradigms are inherently different so we cannot expect all of LLVM's optimizations to work seamlessly for HLS. Several studies using Vivado HLS to generate FPGA accelerators have been demonstrated, including Dynamic Data Structures[133], Sobel Filter [93], Control Algorithms for Power Converters [102], and real-time embedded system vision [72].

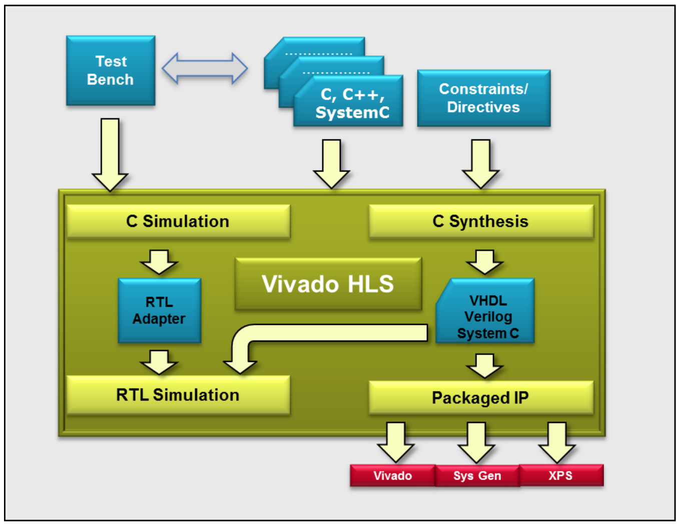


Figure 3.3: Vivado HLS Workflow

User Experience

Typical design flow (Figure 3.3) starts with C code compiled to a pure software implementation and a self-validating testbench to verify correctness. The user must specify the top function in the code that they wish to synthesize to hardware. The GUI provides the user a list of code regions (targeted at loops, function bodies, and other bracketed regions) that can be optimized using synthesis directives to guide the RTL generation.

At the function level, directives include inline, instantiate (local optimization), dataflow (improve concurrency), pipeline (improve throughput), and interface (function defines an interface), among others. At the loop level, dataflow pipelining, and the common optimizations of loop-unrolling, loop-merging, loop-rotation, dead-code elimination, etc. are also available. The interface directive higlights an important aspect of the flexibility in Vivado HLS – the ability to generate user specified I/O protocols. Documentation highlights the convenience of using Xilinx's AXI4 interfaces in terms of compatibility with their IP catalog, however, the tool does not prevent custom protocols, and thus increases portability. The tool also provides the ability to set custom bit-widths for all variables in a design, leading to more efficient use of area.

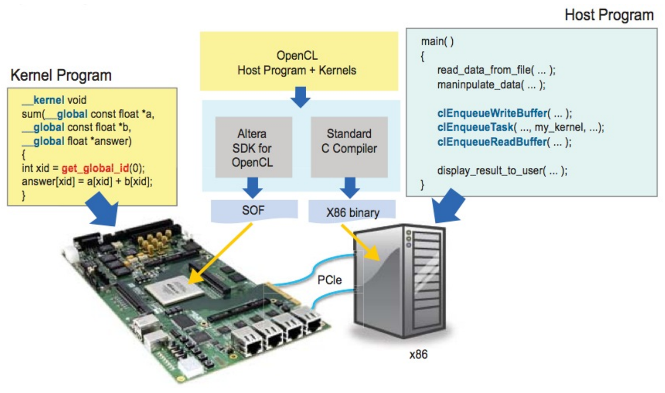
The directives guiding the optimizations can be defined directly in the source code using *#pragmas*, similar to OpenCL code. They can also be defined separately in a directives.tcl file that the synthesis tools will apply before RTL generation. The second option creates the flexibility of maintaining multiple solutions testing different optimizations using the same source code. As design space exploration is a generally iterative and time consuming process,[14] this can reduce the development time.

3.2.2 Altera OpenCL

Open Computing Language (OpenCL) is a programming language originally proposed by Apple Inc. and maintained by the Khronos Group [80]. The OpenCL specification provides a framework for programming parallel applications on a wide variety of platforms including CPUs, GPUs, DSPs, and FPGAs [124]. Moreover, OpenCL is a royalty-free, cross-platform, cross-vendor standard that targets supercomputers, embedded systems and mobile devices. OpenCL allows the programmers to use a single programming language to target a combination of different parallel computing platforms. Parallel computation is achieved through both task-level and data-level parallelism.

The OpenCL framework provides an extension of C (based on C99) with parallel computing capabilities and the OpenCL APIs, which is an open standard for different devices. In the OpenCL programming model, a host is connected to one or more accelerator devices running OpenCL kernels. Device vendors provide OpenCL compilers and runtime libraries necessary to run the kernels. The host program is written in standard C to query, select, and initialize compute devices. Communication between the host program and accelerators is established through a set of abstract OpenCL library routines. Each accelerator device is a collection of compute units with one or more processing elements. Each processing element executes code as SIMD or SPMD.

In the FPGA industry, both Altera and Xilinx have announced support of OpenCL HLS in their FPGA development tools. Altera released an OpenCL SDK in 2013 that supports a subset of the OpenCL 1.0 specifications. Xilinx started to support OpenCL in their Vivado HLS tool in April 2014. In this paper we focus on the Altera OpenCL SDK.



```
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```

```
main( )
{
    read_data_from_file( ... );
    maninpulate_data( ... );

    clEnqueueWriteBuffer( ... );
    clEnqueueTask( ..., my_kernel, ...);
    clEnqueueReadBuffer( ... );

    display_result_to_user( ... );
}
```

Figure 3.4: The OpenCL system overview, image from [118].

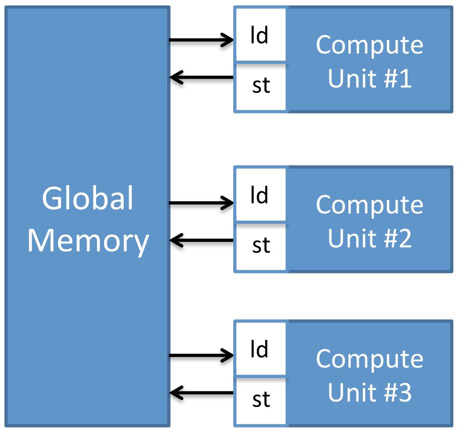
The Altera OpenCL SDK provides software programmers an environment based on a multi-core programming model that abstracts away the underlying hardware details while maintaining efficient use of FPGA resources. The Altera OpenCL compiler (AOC) is an offline compiler that translates OpenCL to Verilog and runtime libraries for the host application API and hardware abstractions. The OpenCL system overview is shown in Figure 3.4. Unlike the OpenCL compiler for CPUs and GPUs, where parallel threads are executed on different cores, AOC transforms kernel functions into deeply pipelined hardware circuits to achieve parallelism. AOC uses a CLANG front-end to parse OpenCL extensions and intrinsics to produce unoptimized LLVM IR [39]. The middle-end performs optimization with about 150 compiler passes such as loop fusion, auto vectorization, and branch elimination. On the back-end, the compiler instantiates Verilog IP and manages control flow circuitry of loops, memory stalls, and branching. Finally the generated kernel is loaded onto an Altera FPGA using an OpenCL compatible hardware image. Various applications using OpenCL to program FPGA accelerators have been demonstrated, such as information filtering [39], Monte Carlo simulation [118], finite difference [47], particle simulations [47], and video compression [40].

User Experience

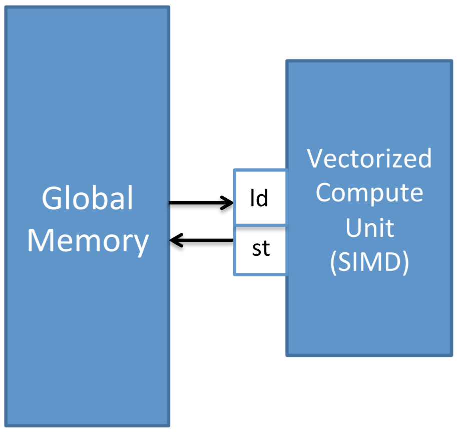
The AOC is designed for software programmers to construct parallel FPGA applications. It has a similar command line interface to the GCC compiler. All OpenCL codes must be included in a single text file before passing to the compiler. The AOC will generate transformations, create Quartus II project files and perform synthesis, place and route, and bitfile generation for FPGA execution. There are several compiler optimizations that can be applied to OpenCL code: kernel vectorization, static memory coalescing, generating multiple compute units, and loop unrolling. Optimizations, when invoked by the user, are applied automatically by the compiler on the whole code to improve processing efficiency. In addition, the programmer can specify attributes, such as *num_compute_units* and *num_simd_work_items*, in the source code to manually control the degree of kernel vectorization and parallel compute units respectively, as shown in Figure 3.5. When setting the *num_simd_work_items* attribute, the data path within a compute unit is replicated to increase throughput and can also lead to memory coalescing. On the other hand, the *num_compute_units* attribute will also duplicate all the control logic which may increase the number of global memory access. Figure 3.6 shows the difference between these two optimizations. Both techniques can be combined to further increase the parallelism at a cost of higher memory bandwidth usage and more FPGA resource occupation. Furthermore, AOC allows users to specify the loop unrolling factor by setting the pre-processor directives (#pragma unroll). In Figure 3.5, the loop iteration is unrolled 16 times. If the amount of unrolling is not specified, AOC will fully unroll the loop. Moreover, the AOC can perform resource-driven optimizations that analyze various combinations of compute unit number, work group size, loop unrolling factor and number of shared resources under the constraints of available hardware resources and memory bandwidth to determine the optimal choice of these values. The resource-driven optimizer is invoked when the programmer sets the *-O3* switch in the compiler.

```
 __attribute__ ((num_simd_work_items(1)))
 __attribute__ ((num_compute_units(3)))
 __kernel
void SampleKernel(__global int * restrict a_in,
                  __global int * restrict a_out)
{
        int i;
        #pragma unroll 16
        for(i = 0; i < 128; ++i)
                ....
}
```

Figure 3.5: Sample OpenCL kernel function with programmer set attributes.



(a) FPGA kernel with multiple compute units.



(b) FPGA kernel of one compute unite with kernel vectorization.

Figure 3.6: OpenCL compiler optimization techniques to increase parallelism.

To test the OpenCL code functionality, AOC provides an OpenCL emulator to simulate the behaviour of the OpenCL kernel program. The emulated kernel is used as a dynamically linked C++ library that can be called from a host program. To compile the OpenCL code for emulation, the *-march=emulator* option should be included in the compilation. Programmers can write a host program to verify if the OpenCL kernel works as designed. In addition, basic C functions like *printf* can be used inside the OpenCL kernel with the emulator to check intermediate values. When the emulator is used, no compiler optimizations can be applied. At present, there is no RTL simulations for hardware programmers to test the generated Verilog kernel.

3.2.3 Bluespec System Verilog

Bluespec System Verilog (BSV) [105] [104] is a high level hardware description language built upon the synthesizeable subset of SystemVerilog. BSV's behavioral model is based on *Atomic Rules and Interfaces*. The rules ensure parallelism, which is well suited for complex hardware designs. BSV also provides powerful abstraction mechanisms that were previously believed to be suited only for software applications. BSV derives most of these abstractions from System Verilog, such as module and module hierarchies, loops, and user-defined data types (enum, struct, tagged union). BSV provides architectural transparency - meaning the programmer, not the tool, expresses the architecture of a the design. This transparency places the tool in an area between HLS and HDL; it abstracts away some of the complexities of working at the hardware level, yet it loses some of the automation provided by many HLS tools. BSV has strong type-checking which ensures all objects are compatible and conversion functions are valid [5]. BSV also provides strong static checking by preventing movement of values to/from currently unused modules - ensuring a synchronizing mechanism must be used to cross clock boundaries. All of these features help eliminate timing errors.

Language Semantics

In conventional Verilog, values are typically kept synchronized by using an *always* block. In BSV, *Rules* are used to describe how data is moved from one state to another. The rules are atomic in nature, meaning the execution of each rule should be considered independently. A Rule is triggered when all of its preconditions are met. Preconditions are boolean expressions, which are purely combinational logic and do not have any side effects. All actions within a rule occur simultaneously, implying all rules should be composed of independent actions. Rules help in achieving higher concurrency and avoiding race conditions. The independent nature of rules allows hardware to execute multiple rules concurrently.

```
interface  Product_Interface  ;
  method int readResult  ;
  method Action setValues (int  p,  int  q,  int  r)  ;
endinterface

(* synthesize *)
module product (Product_Interface)  ;

 Reg#(int) x <- mkReg (0) ;
 Reg#(int) y <- mkReg (0) ;
 Reg#(int) z <- mkReg (0) ;
 Reg#(int) result  <- mkRegU ;

 Reg#(Bool) b <- mkReg (False) ;

 rule  toggle  ;
          b <= !b ;
 endrule

 rule  r1  (b)  ;
         result  <= x * y ;
 endrule

 rule  r2  (!b)  ;
         result  <= x * z ;
 endrule

 method readResult = result  ;

 method Action setValues (int  p,  int  q,  int  r)  ;
         x <= p ;
         y <= q ;
         z <= r ;
 endmethod

endmodule
```

Figure 3.7: Multiplication example in BSV.

A BSV module's interface consists of methods instead of a ports list. A method is similar in concept to a function, it takes in arguments and returns a result. However, they differ in that a method also carries with it a set of implicit conditions. Each method has an associated *Ready* signal (output port) and an *Enable* signal (input port) if it is an *Action* method. Because BSV does not accept standard C like the other HLS tools presented in this paper, we show a simple BSV code to multiply two integer values for syntax clarification (Figure 3.7). The module named *product* provides *Product_Interface*. *Product_Interface* has two methods. The readResult is a *Value Method*, which forms the output port of the module. The second method *setValues* takes three arguments and forms the input port of the module. The attribute *synthesize* tells the BSV compiler to generate a separate hardware implementation (Verilog) of the following module. We then define four registers of type *int* and instantiate them with a BSV defined module *mkReg*. Finally we apply different rules to compute the product.

BSV uses a dynamic scheduler which allows multiple rules to be executed in each clock cycle. The compiler, guided by the scheduler, performs a detailed analysis of all the rules and their interactions with each other and maps the design into the clocked, synchronous hardware. This mapping permits multiple rules to be executed in each clock cycle, however, it may also give rise to a situation where two or more rules conflict (e.g. limited resources). In the absence of user guidance, the compiler arbitrarily chooses which rule to prioritize and issues a warning. As shown in Figure 3.8, the BSV compiler makes sure that all the control logic is dictated solely by the applicable rules, thus functional correctness is achieved.

User Experience

Bluespec compilation can be controlled from the Bluespec GUI or from the command line interface. It is very important for the programmer to understand how Bluespec

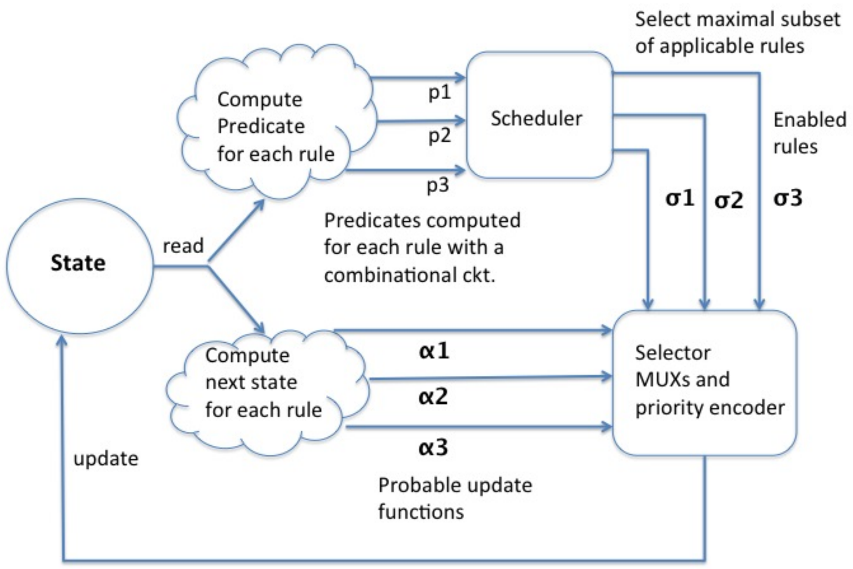

Select maximal subset of applicable rules
Compute Predicate for each rule
p1
p2
p3
Scheduler
Enabled rules
σ1
σ2
σ3
State
read
Predicates computed for each rule with a combinational ckt.
Compute next state for each rule
α1
α2
α3
Selector MUXs and priority encoder
update
Probable update functions

Figure 3.8: Scheduling of Rules in BSV

language semantics get converted to RTL. For example, to interact with the module we need ports which are formed by Bluespec *interfaces* [93].

The RTL code is generated by scanning each line of the Bluespec code. All the Bluespec design entity names are preserved in the RTL code generated by the compiler. For example, in Bluespec, methods are used to bring data into the module and send out of the module. Methods eventually form port signals of the same name in the RTL, which makes the RTL code more readable and traceable [93].

The benefit of BSC over C-based synthesis tools is that BSC does not invent the architecture for us. With C-based synthesis, the designer writes an (often sequential) algorithm, and the C-based tool figures out an architecture to implement that algorithm. With BSC, the designer chooses the architecture, writes BSV to express the desired hardware implementation, and BSC generates that hardware. However the designer may not have many choices for design exploration since the design entry is at the hardware level.

3.2.4 LegUp 3.0 (U. Toronto)

LegUp [33][34] is an integrated HSL environment that is developed and maintained at the University of Toronto. The project's goal is to take existing C applications in their entirety, compile and run them on an FPGA. This differs from most HLS tools that focus on compiling only specific regions of code to hardware. Currently, LegUp targets two Altera platforms – a Cyclone II on the DE2 board, and a Stratix IV on the DE4 board. Design choices like the soft-core processor and the communication bus are tightly coupled to these specific boards, and porting them to other platforms would be a non-trivial task. However, the hardware accelerator cores themselves have only a few Altera specific components, and can be ported with little effort.

LegUp supports three types of compilation: pure software, pure hardware, and a hybrid approach. In a pure software compilation, LegUp only generates assembly code that can be run on a Tiger MIPS [123] soft-core processor, enabling the FPGA to handle almost all of the C language standard. However, it limits the performance and energy efficiency. In a pure hardware compilation, LegUp generates a custom circuit for the entire application. However, only a subset of the C language is supported for hardware acceleration. For example, dynamic memory and recursive functions do not make sense as a circuit. It should be noted that these limitations are ubiquitous among all HLS tools. In a hybrid compilation, LegUp generates a custom circuit for only part of the application, and assembly code is generated for the rest. LegUp keeps any hardware calls from software transparent to the user by automatically inserting them into the assembly code. Support exists for both blocking and non-blocking hardware calls. Blocking calls improve the energy efficiency, but non-blocking calls will improve the performance.

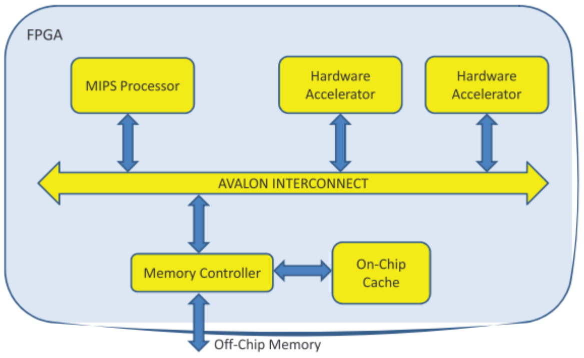


Figure 3.9: LegUp 3.0 Target Architecture

Figure 3.9 shows the complete hybrid architecture that LegUp generates. It has a single Tiger MIPS processor and can have multiple hardware accelerators depending on the application. All memory data is stored in an off-chip memory, but an on-chip cache is used to improve performance. All communication is carried across Altera's Avalon bus [3]. Targeting a single type of architecture has its trade-offs. A shared global memory space prevents costly data offloading, and assuming only one type of bus simplifies the communication protocols. However, it results in very specialized hardware accelerators. Their performance is limited by the Avalon bus's bandwidth, and cannot be extended to higher bandwidth architectures. LegUp is currently a very specific tool, but extensions could make it more general.

Typical design flow for an application starts with the C code compiled into a pure software implementation. LegUp provides a built in profiler to help identify computation intensive code regions that are strong candidates for hardware acceleration. This stage is not automated: the user must mark functions for hardware compilation. LegUp then generates the necessary accelerators in Verilog and the application is recompiled to insert the necessary hardware calls. The entire design can be simulated to verify correctness and then synthesized for the target FPGA. These steps are repeated iteratively until the designer is satisfied with the performance.

Similar to several other tools, LegUp is based on the LLVM compiler framework. The impact of various LLVM optimisations on the performance of the generated hardware structures is explored in [35]. Extra passes are added to LLVM for HLS and work in three phases: allocation, scheduling, and binding. The allocation stage determines the available hardware based on the target architecture and manages the application's constraints like clock speed and power consumption. Scheduling orders the operations. Currently, only as-soon-as-possible scheduling is supported, but because of LLVM's modular design this pass can be easily changed. Binding reduces resource utilization for complex instructions (i.e. multiply or divide) by multiplexing the data path through a single component. A weighted bipartite matching heuristic is used to handle the binding problem.

User Experience

Being a research tool, LegUp is not primarily concerned with usability. Installation can be difficult because it requires specific versions of standard tools. The LegUp website [88] does offer an Ubuntu virtual machine, for VirtualBox, with the tools already setup.

The user interfaces with the compiler through command line, but comprehensive makefiles and many example are provided to help get new users started. Most operations can be handled through a provided makefile, from compiling and simulating to automatic project creation and synthesis. Optimizations are applied in two locations: For hardware specific optimizations (e.g. pipelining) the user needs to create a .tcl script. Common software optimizations (e.g. loop-unrolling) can be passed directly to the LLVM compiler through the makefile.

3.2.5 ROCCC 2.0 (UC Riverside)

The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [125] is a C to VHDL compiler built using SUIF [130] and LLVM [87]. ROCCC was initially developed at the University of California Riverside. ROCCC 2.0 was developed by Jacquard Computing Inc. with funding from the AFRL under an SBIR contract. ROCCC 2.0 is freely available on GitHub. The SUIF toolset is used to implement the high-level transformations, such as loop and array transformations, and generates an intermediate representation, CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics) [62], a readable text file. The CIRRF code is then passed to LLVM for further low-level optimizations and VHDL generation. Hardware specific optimisations, such as pipelining, expression tree balancing etc., are implemented in the second pass of LLVM. The ROCCC design puts a special emphasis on user driven transformations and optimisations whose objective is to maximise throughput by exploiting the inherent parallelism and reducing the area footprint of the generated code [31]. Another objective is reducing the number of memory accesses per output result by automatically reusing already fetched data [63].

The designs in ROCCC are divided between two abstractions: systems and modules. Systems perform the main computations that are being accelerated for the application. They are based around *for-loops* that allow them to process streams of data and access memory through arrays. The system generated code is the only code that can read and write external data. Modules are designed to implement a specific task and are used as function calls from systems or other modules. Both modules and systems have a variable number of parameters that represent I/O ports in the underlying hardware core. These abstractions provide reusability and ease of development for large systems with multiple levels of complexity. ROCCC also allows the addition and use of external cores in projects as if they were another function call.

ROCCC provides a number of high-level optimizations. They include: division elimination, multiply elimination, loop-unrolling, module inlining, redundancy, systolic array generation[32], loop-fusion, and Temporal Common Subexpression Elimination (TCSE) [69]. Additionally, ROCCC provides some low-level optimizations. They include: maximize precision, copy reduction, fanout tree generation, and arithmetic balancing. System code has access to all these optimizations whereas modules do not have the systolic array generation, loop-fusion, and TCSE optimizations. All loops in modules are fully unrolled because modules do not stream data. ROCCC is designed to create platform independent hardware structures. However, by default larger FIFOs use vendor-specific IP cores to improve timing. They are wrapped in an *InferredBRAM* component to allow easy portability. Similar to most C-to-FPGA tools, ROCCC is only able to compile a subset of C that works well with FPGAs. Features like dynamic memory allocation and recursion are not supported.

User Experience

ROCCC development is accomplished through the Eclipse IDE with custom plugins for the GUI. Since the idea behind HLS tools is to make hardware development easier, using Eclipse provides a small learning curve to ROCCC. Short and straightforward tutorials are available online [6]. Those comfortable with the Eclipse IDE will be able to produce systems at the same rate that they would produce a software project. Adding modules and other IP cores to projects is done through their respective menu screens after choosing to import them on the ROCCC drop-down menu. After writing the C code for either a module or system, clicking the build button brings up a menu process. The menu process involves selecting optimizations, setting a pipeline preference, and managing I/O streams. All of which have their benefits to a circuit described in the tutorial. A deep understanding of how the hardware works in order to test it is not required since ROCCC provides a test bench generation process. It involves including .txt files with sequences of inputs and expected outputs. The user can verify the circuit from the console output of the simulator.

3.3 Dilation Kernel Example

In this section, we use a simple image processing operation, dilation, to demonstrate the use of various HLS tools. Dilation sets the value of the center pixel of a window (in our case 3x3) to the maximum of all values in that window. In the case of a greyscale image, it will make white regions brighter and reduce dark spots as shown in Figure 3.11. We start with a black and white image with the white letters "UCR" crisscrossed with black

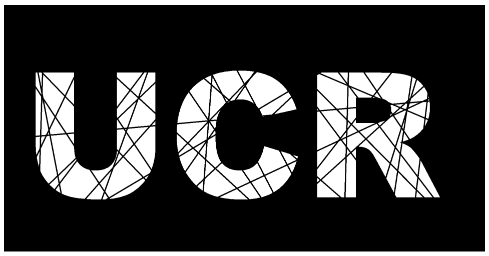
```
//slide window across image
for(i = 0; i < HEIGHT; ++i) {
  for (j = 0; j < WIDTH; ++j) {

    maxRow1 = MAX(img[i   ][j],
                  img[i   ][j+1],
                  img[i   ][j+2]);

    maxRow2 = MAX(img[i+1][j],
                  img[i+1][j+1],
                  img[i+1][j+2]);

    maxRow3 = MAX(img[i+2][j],
                  img[i+2][j+1],
                  img[i+2][j+2]);

    f_img[i][j] = MAX(maxRow1,
                      maxRow2,
                      maxRow3);

  }
}
```

(a) Dilation code in C. Same code was used for the ROCCC compilation

```
for(i = 0; i < HEIGHT; ++i){
  maxCol1 = MAX(img[ i ][0],
                img[i+1][0],
                img[i+2][0]);

  maxCol2 = MAX(img[ i ][1],
                img[i+1][1],
                img[i+2][1]);

  for (j = 0; j < WIDTH; ++j)  {
    maxCol3 = MAX(img[ i ][j+2],
                  img[i+1][j+2],
                  img[i+2][j+2]);

    f_img[i][j] = MAX(maxCol1,
                      maxCol2,
                      maxCol3);
    //shift maxes for reuse
    maxCol1 = maxCol2;
    maxCol2 = maxCol3;
  }
}
```

(b) Optimized Dilation code in C. These optimizations, plus data reuse, are applied by ROCCC to reduce memory reads (Table 3.5)

Figure 3.10: Assume WIDTH and HEIGHT are defined in terms of filter size. A simple change to access pattern and order of calculations makes a significant difference for CPU optimization opportunities


(a) Crisscrossed UCR Logo

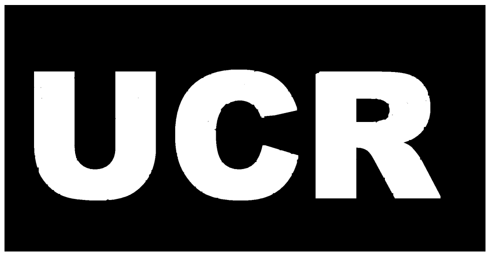
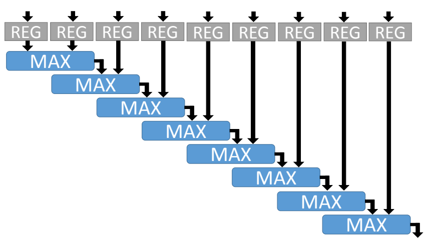
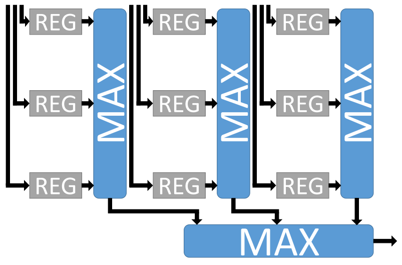

(b) Clean UCR Logo following Dilation

Figure 3.11: Dilation Example

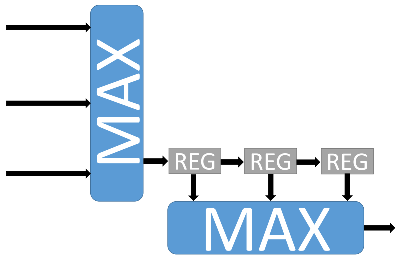
lines. After several passes of a dilation, the cross marks are removed and a clean UCR logo shows through. Dilation is rather amenable to comparisons because it uses simple,



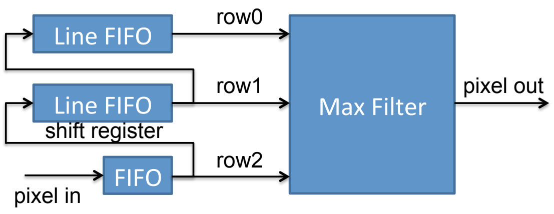
(a) Zero optimization - Lack of tree balancing causes wasted memory access and increased latency.



(b) Tree balancing helps latency, but still wastes memory accesses.



(c) Example Dilation Circuit, using TCSE optimization



(d) Example Dilation Circuit, multi-line buffer

Figure 3.12: Examples of Naïve implementations(Top) vs. Smart optimizations(Bottom)

straight-forward C code, an efficient circuit is not too challenging to generate, yet compilers optimizations can result in very large performance increases.

The simplest way to implement this filter that follows directly from the sequential description from the C code can be seen in Figure 3.12a. In this circuit, we simply register the nine pixels within the window and then compare pairs of two pixels to find the max of the nine pixels. While simple to generate from the source, it has two major drawbacks - latency and wasted memory requests. Without a mechanism to save values from window to window, we waste 66% effort re-fetching the same data as seen in Figure 3.13.

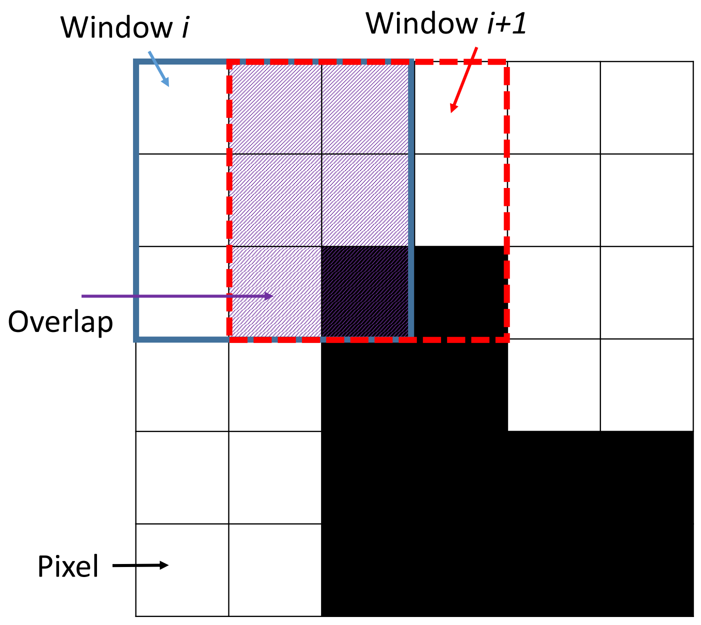


Figure 3.13: Overlapping region of adjacent Dilation windows can lead to smart optimization or redundant memory requests

A slightly improved version can be seen in Figure 3.12b. Balancing the arithmetic between the max operations helps to reduce latency, but does nothing to help with the memory accesses. Unintelligent use of data can cause excessive uses of the memory subsystem, and really hinder performance - especially throughput.

Figure 3.12c demonstrates the type of circuit we would like to see produced by the HLS tools. By using temporal common sub-expression elimination, we have data reuse between windows, maximizing the utilization of the memory system. This layout allows for the streaming of 3 rows of the image from memory, and we take the max of the current column. The max is registered and shifted through a buffer for use in the next cycle. After the initial three cycle latency for the buffer to be filled, this circuit can produce one pixel per cycle. We can further increase performance by duplicating this design on the FPGA by unrolling the inner loop. Each kernel on the device would process overlapped streams of the image, creating data reuse vertically as well as horizontally. This design yields itself to low latency, good data management, and the ability to scale as the resources on the chip allow.

A final possible optimization, represented in Figure 3.12d, uses a line buffer (supported by AOC): an entire row of the image is shifted onto the FPGA and a window of three pixels is used to generate the max. Great performance with this technique is possible as long as there are enough resources on the FPGA to accommodate all the necessary line buffers. It may not be feasible when using large images.

3.3.1 CPU Implementation

In the introduction, we proposed an argument about the efficiency of FPGAs in relation to CPUS, and how FPGA codes can achieve their speed-up relative to traditional software. Before presenting the how the various HLS tools implement the following benchmarks, we feel it is prudent to have a baseline software implementation (compiled with GCC 4.6.3, -O3) for comparison.

Figure 3.10 shows two implementations of the dilation filter: in Figure 3.10a, we have a simple, direct interpretation of dilation (taking care not to do a completely serial implementation like Figure 3.12a) and in Figure 3.10b a programmer optimized version to take advantage of TCSE. Since a major focus of this paper is how much the tool can accomplish for the programmer, we started with the direct implementation to determine the level of efficiency GCC could achieve.

In the direct case, the compiled assembly for x86 has 32 instructions for the inner loop, meaning 32 machine instructions executed for every output pixel generated. Another issue with the assembly is the lack of register reuse. Register allocation is a particularly hard problem with compilers, and in this case reusing registers for the next iteration of the loop was not identified - all nine pixels used in a window are loaded with move instructions from memory. Depending on the memory hierarchy and caching structure, this could also be a detriment to performance.

Unrolling only provides a minor benefit to the simple software implementation and yields a total of sixty instructions. In this case, all the compiler can accomplish is duplicating the body of the inner loop once, without any register reuse between the unrolled loop bodies as evidenced by the 18 load requests. Overall, the code executes 30 instructions per output pixel.

In generating the optimized version of the code, we wrote it specifically to force value reuse between iterations. We implemented four different versions, including cache blocked memory accesses to determine the best performing implementation - row based access and non-memory blocking. The dilation filter by default reuses a lot of values that will be brought to cache as it passes over the rows, so blocking the accesses only adds overhead and reduces performance. For value reuse, we simply pre-compute the two maxes of the initial columns of the current window before iterating over the rows of the image. This minor re-organization simplifies the inner loop to only calculate one new max from the input, write the output, and then shift two of the current maxes to reuse on the following iteration.

The compiler performs more extensive optimisations on this implementation. During initialization in the outer loop, the code executes forty-six instructions and stores two results in this stage with 12 loads, for a total of twenty-three Instructions per output pixel. Loop unrolling is able to unroll the inner loop six iterations, for a much better stride over the majority of the data. During this stage, the code does 18 loads and six stores and executes 76 instructions, for 12.67 instructions per output pixel.

With the best compiler optimizations applied to an already hand-optimized version of the dilation code, the peak performance we saw was 13 instructions per output pixel. CPUs run at an order of magnitude faster clock frequencies, but still has to execute at least an order of magnitude more instructions and thus, clock cycles. And that ignores any many-cycle-stalls from memory misses. The key point we want to highlight is that in order to achieve that performance, a programmer had to be knowledgeable about the possible shortcomings of the platform and spend development time altering the source to generate the desired assembly performance - the tools could not achieve that independently. Contrast this to some of the following examples where naïve code and a compiler flag are enough to generate well optimized hardware implementations.

3.3.2 Vivado HLS

To implement the dilation project in VivadoHLS, we started with the sample C code provided in the beginning of the section, except for a minor difference in the parameter list. To have VivadoHLS process the input as a stream, and thus pass the input as a pointer, a protocol must be created to interface between the stream and the circuit. Since the information we are interested in is how the tool compiles the kernel and not the data passing, we elected to use an input array of fixed size to avoid the extra overhead.

With that minor change, VivadoHLS was able to compile the given source to hardware. The tool also uses the programmer-provided software test harness to generate a hardware testbench to validate correctness. It is possible to go from C to VHDL without ever looking at a waveform - a benefit to software developers without experience in hardware development. By default, VivadoHLS does not apply any optimizations. It is easy to get a working RTL solution, but the tool will only do exactly as the programmer directs. In this example, the first solution generated had similar performance characteristics to Figure 3.12a. Adding simple directives to the inner loop like pipelining and unrolling and the resulting RTL resembles Figure 3.12c. VivadoHLS is capable of creating some of the best solutions, but it does require the knowledge of what optimizations to apply when and where.

3.3.3 OpenCL

We have implemented dilation in OpenCL by using a single loop structure, as in AOC, nested loops should be avoided for performance considerations [2, 1]. Note that, for a single loop structure, the output image has the same size as the input image. Also, the edge values may not be correct since all loop iterations should have the same computation. However, this is a minor issue for image processing as edges are typically not the region of interest. The OpenCL implementation code is shown in Figure 3.14. The code was compiled with and without optimizations(-O3), however, the generated hardware description is not human-readable and it is not possible to observe the effects of the optimizations on the generated circuits. We did observe significant changes in the resource utilizations and the clock frequency.

```
#define R  1080
#define C  1920
 __kernel
void  max_filter ( __global  unsigned *  restrict  p_i ,
                   __global  unsigned *  restrict  p_o)
{
        unsigned rows[2*C+3];  // line  buffer
        const unsigned  iterations  = R*C;
        unsigned count = 0,  wimMax= 0;
        while  (count !=  iterations )
        {   // infer  shift  registers  as  line  buffer
                #pragma unroll
                for  (unsigned i  = C*2 + 2; i > 0; --i)
                {
                        rows[i]  = rows[i  - 1];
                        rows[0]  = p_i[count];
                }
                winMax = 0;
                #pragma unroll
                for  (unsigned i  = 0;  i  < 3; ++i)
                {
                        #pragma unroll
                        for  (unsigned j  = 0;  j  < 3; ++j)
                        {
                                unsigned p = rows[i*C+j];
                                 if (max_pix < p)
                                        winMax = p;
                        }
                }
                p_o[count++] = max_pix;
        }
}
```

Figure 3.14: OpenCL implementation of Dilation using a single loop structure.

3.3.4 Bluespec System Verilog

The Bluespec *dilation* module reads three rows of an image matrix at a time. It forms the pipeline of incoming data and as soon as the window of 3x3 is received, the module outputs the maximum value in a window. The Bluespec module is built similar to Figure 3.12c. The interface to dilation in BSV is declared as depicted in Figure 3.15.

```
typedef Int#(16) size;
interface Incomingdata;
 method Action input_in (size a,size b, size c);
 method Bool isPipeFlush();
 method size out_data();
 method Bool done();
endinterface
```

Figure 3.15: BSV Interface to dilation module.

The interface has three methods. An *Action* method usually causes some state change. In this example *input_in* is an *Action* method and brings in three inputs *a,b,c* into a module. The *out_data* method is a *Value* method which outputs data from the module. In our case we return 16 bit wide *maximum value of 3x3 window* from the module, which the Bluespec compiler names the same as method name, *out_data*. The other two methods act as the control signals to the outside modules . The *done* method terminates the execution of a module and *isPipeflush* method let the other module know that pipeline is being flushed. The dilation module is shown in Figure 3.16. The module begins by importing packages and is instatiated with two parameters: height and width of an image. As we can see in Figure 3.16, rule *incr* writes to *wCount* and rule *check* reads the value of *wCount*, the BSC attribute *descendency_urgency* guides the compiler to schedule the more urgent rules first in case both of these rules fire in the same cycle. We then apply rules to compute the maximum value of window. Each of the rules have some conditions specified under which they fire. The pseudo-code can be seen in Figure 3.16.

```
import Incomingdata :: *;
import Vector :: *;

module mkTb #(parameter size height, parameter size width)
           (Incomingdata);

function size maxOf3(size x, size y, size z);
        let temp = max(x,y);
        return max(temp,z);
endfunction

(* descending_urgency = "check,incr" *)
rule shift_to_stage1( !pipeflush && valid0 );
        stage1[0] <= stage0[0];
        stage1[1] <= stage0[1];
        stage1[2] <= stage0[2];
        valid1 <= valid0;
endrule

rule incr(!pipeflush && valid0 );
        wCount <= wCount + 1;
endrule

rule check( !pipeflush && wCount == width-1 );
        pipeflush <= True;
endrule

rule find_max( valid2 );
        let temp_max0  = maxOf3(stage0[0], stage0[1], stage0[2]);
        let temp_max1  = maxOf3(stage1[0], stage1[1], stage1[2]);
        let temp_max2  = maxOf3(stage2[0], stage2[1], stage2[2]);
        let temp_final = maxOf3(temp_max0, temp_max1, temp_max2);

        max_of_window <= temp_final;
        max_of_all_stage <= True;
endrule

method Action input_in(size a, size b, size c)
        if (!pipeflush);
        stage0[0] <= a;
        stage0[1] <= b;
        stage0[2] <= c;
        valid0 <= True;
endmethod
endmodule
```

Figure 3.16: Dilation implementation in BSV.

As a part of verification, testbenches are manually written in Bluespec. In our case, we instantiate a dilation module in our testbench module. We keep sliding our window one column at a time and as soon we reach the width of an image we slide our window by one row. We repeat the steps until the last window can be formed. In this testbench, the top-level interface is "Empty", which means when we see the synthesized verilog, we can see only clock and reset lines. Bluespec provides a test driver module for modules with "Empty" interfaces which applies reset and then drives the clock indefinitely.

3.3.5 LegUp 3.0

As we previously stated LegUp targets a very specific architecture, and this limits any generated kernel's bandwidth to 2 memory channels. Even though these kernels cannot be optimized to run with higher bandwidth they can be optimized to improve the bandwidth utilization. Table 3.5 shows the performance results as we apply loop unrolling to the dilation kernel. An important note we want to point out is that for every test, the total number of write memory accesses is exactly the same because LegUp only duplicates the hardware engines, but does not merge their computations.

However, duplicating the engines does increase the number of available memory requests. LegUp's FSM can then schedule the requests closer together to improve bandwidth utilization. This also improves the number of runtime cycles. Clock frequency is affected as the kernel is unrolled due to larger resource utilization on the FPGA. Overall runtime is also affected as shown in Table 3.5.

3.3.6 ROCCC 2.0

The ROCCC implementation of dilation starts with the C code shown in Figure 3.10a. The same code was used in [125] to demonstrate the TCSE optimization. The sample code uses a modular approach which replaces each section of if-statements with a call to a MAX module. The current version of ROCCC would not unroll either the inner loop or the outer loop more than twice with the modular approach. In order for ROCCC to unroll either loop further, the code had to be written with the max functions inlined by hand.

To take advantage of the loop unrolling, input and output streams were added to the circuit through the ROCCC GUI. Multiple input streams divide up the number of memory accesses, further reducing the execution time. Each extra window gained from loop unrolling shares the bottom two rows with the window above it, meaning that an extra input stream is required for the bottom row of the new window. With no unrolling, three inputs (one input for each row of the 3x3 window) and one output were used. The amount of outputs is equal to the amount of times the loop is unrolled and the number of inputs is two more than the number of times the loop was unrolled. For example, two unrolls uses four inputs and two outputs.

The circuit that ROCCC generates processes input similar to the circuit depicted by Figure 3.12c. The MAX component represents the if-statements sequence inside the nested loop. Every cycle an element from each row of the 3x3 window is pulled in through a FIFO. Once all three elements of the row are in, they get compared and the max from each row goes to the next comparison block to produce the maximum number of the 3x3 window. This specific example is a dilation circuit with three input streams. With only one input stream there would be just one FIFO for all three MAX components. Loop unrolling and adding the appropriate amount of input streams creates multiple compute units.

3.4 AES Encryption Kernel Example

The Advanced Encryption Standard (AES) [4] is a symmetric block cipher that can be used to encrypt and decrypt information to protect electronic data. The AES algorithm has become the default choice for various security services in numerous applications.[68]. The encryption process converts plaintext into an unintelligible form known as cipher-text, and decryption is the inverse of this process. AES processes the data in 128-bit input blocks using a key size of 128, 192, or 256 bits. With respect to those key sizes, the algorithm executes 10, 12, or 14 iteration rounds of transformations. The five core operations of the algorithm are KeyExpansion, AddRoundKey, SubBytes, ShiftRows, and MixColumns. A visual flow-graph of the algorithm can be seen in Figure 3.17. The input block, called the state array, is constructed as a 4x4 matrix of bytes. The state array goes through the appropriate amount of processing rounds where the subBytes, ShiftRows, MixColumns, and AddRoundKey steps are performed on the state array. After the final round, the state array contains the encrypted data.

For this paper, we focused on the KeyExpansion and MixColumns operations of the AES encryption algorithm with a 128-bit key. Both steps of the algorithm are more computationally intensive than the rest, meaning they do more than a single operation on each element of the state array. The KeyExpansion algorithm takes the four 32-bit words of the input key and expands the key into 44 words. The AddRoundKey operation XORs four

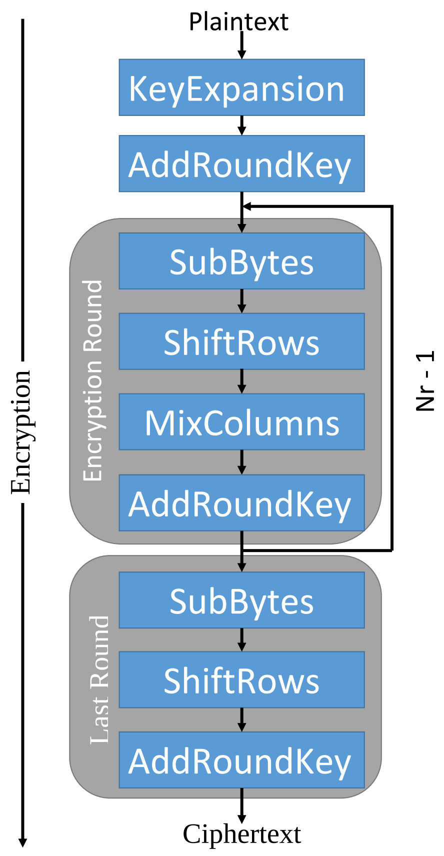


Figure 3.17: AES Flow Graph

of the key words at a time, which requires four words for each round and an additional four during the initial round before the main encryption rounds begin. The SubBytes algorithm makes use of a pre-computed array of round constants and a substitution box to generate the next four words based on the key words of the previous iteration or the initial four key words. ShiftRows is a transposition step where the last three rows of the state array a rotated cyclically. Finally, the MixColumns algorithm is simply a matrix multiplication of the state array with a predefined array composed of the values 0x01, 0x02, and 0x03. The arithmetic for the matrix multiplication in this step is done in the Galois field GF(2^8) in which addition becomes XOR and multiplication becomes bit shifting and XORing.

3.4.1 Implementing AES

When implementing AES in hardware, the HLS tools divided themselves into two groups: those capable of compiling the entire source into functioning hardware code and those that had to break each sub-component of the algorithm into individual components to be combined by hand.

VivadoHLS and LegUp were able to compile the C source with little or no modification, and provided good design space exploration. LegUp also allowed the programmer to specify how much of the AES code should be executed in hardware or software. To compile with the Altera OpenCL Compiler, we simply adapt the original C code to have one input and one output stream as the plain text and encrypted text respectively, and add appropriate OpenCL grammars (function qualifiers, attributes, and et al). Then the code can be directly compiled by AOC and an executable file is generated. However, as noted previously, there are no human-readable Verilog files and corresponding RTL simulation methods provided by AOC, which prevents examining the generated architecture.

ROCCC and Bluespec System Verilog, however, required more direct effort to handle AES. ROCCC is currently unable to compile source with multiple loops at the same nesting level within the same system. Input streams must be accessed within a for-loop and all inputs will be accessed at every iteration of the loop. In ROCCC each system contains only one top-level loop. Multiple systems can be configured in producer/consumer relationships. In order to implement AES in ROCCC, each component of AES had to be compiled as its own system and then combined together. Since BSV is closer to the HDL

Table 3.3: Area utilization and timing results for the pass-through filter.

Tool	Unroll #	LUTs	Registers	BRAM	DSP	Clock Freq.
ROCCC	1	243	298	1	0	338 MHz
	8	645	1479	9	0	337 MHz
	16	1008	2746	16	0	260 MHz
	32	1675	5298	30	0	279 MHz
LegUp	8	282	271	0	0	376 MHz

level than HLS, each component had to be built by hand with significant effort on the programmer.

3.5 Synthesis Results

In this section we report on the FPGA area and performance results where applicable. Note that LegUP and ROCCC were designed with different goals in mind. LegUp focuses on compiling a large subset of the C language. ROCCC focuses on streaming applications, which require a smaller subset of C, with an emphasis on extensive user-directed compile-time transformations and optimizations. We first start with a simple pass-through filter to establish the baseline data for both tools. Results are reported for a Xilinx Virtex-7 (XC7VX690T) using ISE 13.4.

3.5.1 Pass-Through Filter

We compiled a pass-through Filter using ROCCC and LegUP to obtain a baseline for the resource utilization in each compiler. Using different unrolling factors we report the results in Table 3.3. Resource utilization is reported as the total occupied slices (LUTs and registers), BRAM, and DSP elements. We also report the clock frequency after placement and routing. However, our designs did not use pin assignments, or did they specify a target clock frequency. Faster results are likely possible with higher effort in the synthesis tools.

Unrolling is handled very differently in ROCCC than in LegUp. Since ROCCC does not support the use of arbitrary aliases (pointers) within loop bodies, ROCCC can detect the independence of loop iterations and generate parallel and separate loop bodies, one for each unrolled iteration. ROCCC does not make any assumptions regarding the interface to the outside world, e.g. memory, therefore unrolling eight folds would require that eight data elements can be fetched each cycle. As expected the resource utilization scales linearly with the unroll factor. Pass-through is a minimalist design, and most of its resources cannot be shared as the design is unrolled.

LegUp supports the whole C language including pointers. The LegUp architecture assumes two memory channels for all memory requests. Unrolling, by hand, can increase the number of parallel instructions within a loop body, which in turn allows for better utilization of the two memory channels. The compiler does not do any code analysis to identify loop-level parallelism and exploit unrolling. In Table 3.3 we report an unroll factor of eight because it is the smallest value possible in LegUp. Parallelism could easily be added to the LegUp design by modifying the C code.

It is worth mentioning that, due to architectural assumptions made by each compiler, LegUp does not utilize BRAMs in its design, while ROCCC utilizes multiple BRAMs. ROCCC generates a general-purpose kernel for any architecture, which includes architectures having high bandwidth and large memory latencies that often support many outstanding requests. If the datapath should stall for some reason, the outstanding requests

Table 3.4: Dilation area utilization

LegUp

unroll	LUTs	reg.	FIFO18E1	FIFO36E1	DSP48
1	3142	4883	0	0	4
8	3142	4883	0	0	4
16	3292	4981	0	0	4

ROCCC

unroll	LUTs	reg.	FIFO18E1	FIFO36E1	DSP48
1	1065	1426	4	4	12
8	2903	4092	18	21	33
16	5486	8133	33	38	102

will be buffered in the BRAMs. In contrast, LegUp assumes a local cache will handle all memory requests. Therefore, the request latencies are always short and the design will have few outstanding requests.

3.5.2 Area Utilization

In Table 3.4 we report the number of LUTs registers, FIFOs and DSPs used by the LegUp and ROCCC implementation of dilation. For BRAMs, Xilinx Virtex-7 uses FIFO36E1 blocks, which are true dual-port 36Kb BRAMs.

As the loop is unrolled, the area utilized by the ROCCC implementation grows linearly (from 1,065 to 10,763 LUTs) while that of LegUp stays relatively constant. The unrolling in LegUp affects mainly the scheduling of by FSM and does not increase the area used by the logic. In fact, in Table 3.5, the number of memory reads performed by the LegUp code stays constant while the one by the ROCCC code decreases.

Table 3.5: Dilation runtime performance

LegUp

			Runtime	
unroll	Freq(MHz)	Memory Reads	cycles	ms
1	167	20676040	99453175	595.53
8	167	20676040	99453175	595.53
16	167	20676040	61103567	365.89

ROCCC

			Runtime	
unroll	Freq(MHz)	Memory Reads	cycles	ms
1	155	2071440	99453175	13.39
8	147	259200	99453175	1.76
16	145	129600	61103567	0.90

3.5.3 Runtime Performance

Table 3.5 shows LegUp and ROCCC's runtime performance on the dilation as the kernel is unrolled. How each tool targets streaming applications is immediately evident in how both tools unroll their designs. As the kernel is unrolled, LegUp will duplicate some of the hardware, but the kernel is always limited to two memory channels. The main performance benefits come from the FSM's memory scheduling. Higher unrolling means more memory request are available sooner, which yields better bandwidth utilization. In fact, unrolling less than eight is not possible in LegUp as evidenced in the table. No unrolling unroll and eight unrolls performs the same because there is no difference in the code. Unrolling is more like a software optimization that improves scheduling than pure hardware replication. On the other hand, ROCCC's focus on streaming applications allows it to fully utilize any memory channels it has available. Therefore, unrolling affects the hardware, input channels, and output channels together. Going from no unroll to 16 unrolls

Table 3.6: AES Encryption Steps - Area, Frequency, and Power

LegUp

AES step	reg.	LUTs	FIFOs	Freq(MHz)	Power (W)
MixColumns	606	536	0	470	21.59
KeyExpansion	784	827	0	320	15.73

ROCCC

AES step	reg.	LUTs	FIFOs	Freq(MHz)	Power (W)
MixColumns	813	450	6	265	13.59
KeyExpansion	2231	1001	9	370	17.67

shows a 14.9X speedup compared to LegUp's 2.5X speedup. Comparing the 32 unrolled versions of each, where ROCCC can assume 34 streams, ROCCC shows a 536X speedup.

3.5.4 AES Performance and Power

Table 3.6 shows LegUp and ROCCC's performance and power results for the MixColumns and KeyExpansion algorithms in AES. Power estimates were generated using Xilinx's Power Estimator (XPE) version 2014.2 and using the reported clock rate. It is interesting to note that if the power estimation is kept to the default of 250MHz, both tools achieve about 13 Watts - meaning very similar designs architecturally. The main difference comes from the different clock frequencies achieved by each tool. ROCCC was able to perform strongest on KeyExpansion since it is purely compute while LegUp performed well on MixColumns, which benefits from the memory scheduling of LegUps FSM.

3.6 Conclusion

In this paper we have addressed one of the main obstacle to a wider adoption of FPGA-based reconfigurable hardware accelerators, namely the programmability of FPGA devices. We have shown that, historically, the use of FPGA accelerators has immediately followed their introduction in the mid 1980s. We have discussed the challenges that must be overcome to raise the abstraction level of FPGA programming to levels similar to those in software. We have identified five high-level programming tools currently available, both research and commercially, that attempt to raise the abstraction level and make it easier for traditionally trained software developers to write applications for FPGA accelerators. These tools are Xilinx Vivado HLS, Altera OpenCL Compiler, Bluespec BSV, LegUp (University of Toronto) and ROCCC (University of California Riverside). We describe the user interaction in using these tools and, using image dilation filter and AES encryption algorithm, we report on how the code is expressed and compiled and discuss the resulting utilization. A summary of the main features of these HLS tools is in Table 3.7.

Table 3.7: Summary of tools features

	VivadoHLS	**OpenCL**	**BSV**	**LegUp 3.0**	**ROCCC 2.0**
Developed By	Xilinx Inc.	Altera	Bluespec	University of Toronto	UC Riverside
Targets	Xilinx FPGAs	Generic(inc. GPUs)	Generic	Mostly Generic	Generic
Origin	Commercial	Commercial	Commercial	Academic	Academic
Learning Curve	Small	Moderate	Large	Small	Small
Interface & Optimizations	GUI development. Local, user driven optimizations	Command line interface. Basic global optimizations	GUI development hand optimized.	Command line interface. Basic global optimizations (.tcl scripts necessary for hardware optimizations)	GUI development. Local, user driven optimizations

Chapter 4

CAMs as Synchronizing Caches for Multithreaded Workloads

Great progress has been made in the area of High Level synthesis. However, from the last chapter we were able to see that the tools were not yet perfect in extracting performance out of highly regular and deterministic code. If the tools are not ready for the regular case, extracting performance from irregular applications is a non-starter. Inspired by work from my colleague Dr. Robert Halstead [67], we decided to look at further applying the multithreaded FPGA model to other application domains[131]. In particular, we decided to look at the Breadth First Search algorithm to test the utility of CAMs as synchronizing caches and multithreading over irregular problems. As data continues to grow at significant rates and our computation infrastructure gets more and more distributed, our algorithms become more and more irregular. Graph representations are particularly well suited to representing many of these problems, and Breadth First Search is often an important low-level operation that sits at the base of many graph algorithms. Improving the performance of BFS can have a direct impact on the overall performance of an algorithm.

4.1 Accelerating Breadth First Search

Graphs are the most natural and efficient way to represent social and biological systems, where nodes represent entities such as people, web sites and genes whereas edges represent the interactions (relationship, communication and regulations). As these problems grow in scale, parallel computing resources are required to meet their computational and memory requirements. This has motivated a substantial amount of work that deals with the design and optimization of graph exploration algorithms, in particular BFS designs, either for commodity processors[96, 97, 140, 70, 116] or for dedicated hardware [128, 13, 24, 16, 74, 143, 79].

A multi-threaded graph engine was developed by [96] which implements a semantic graph database on commodity clusters. They have addressed the issue of irregular memory accesses by using lightweight software multi-threading and data aggregation. This implementation was able to maintain constant query throughput with the scaling of dataset size.

A parallelized BFS algorithm was described by [9] on multi-core architectures. They used a bitmap to keep track of visitation status of a node and demonstrated speedup over previous work. A significant speedup was shown by [20] on distributed memory machines. GPUs have also been chosen by some to speedup computations in variety of applications, including graph processing. A level synchronous BFS kernel for GPUs was proposed in [74] and showed improved performance over previous implementations. A slightly different approach was used in [95] to elevate the graph processing performance. This method used a prefix sum approach for cooperative allocation. In [143], the authors have presented a GPU programming framework for improving GPU-based graph processing algorithms. Another graph processing framework was proposed by [79]. This method uses G-shards and concatenated window representation to store graphs in GPU global memory and provide better performance over other state-of-art implementations.

With the advent of heterogeneous machines, such as Convey HC-1/HC-2 [16], which support cache coherent shared virtual memory accesses from both the software (CPU execution) and the hardware (FPGA execution), application acceleration has become much more feasible . For example, the Convey HC-2 has four Virtex-6 LX760 FPGAs, further allowing multiple sections of an application to be written to a FPGA without need of reconfiguration at runtime. A reconfigurable architecture for parallel BFS is presented in [13]. This method worked well for graphs with out-degree up to 32, but does not scale to outperform the approach in [24] for higher degrees. It uses level-synchronous BFS algorithm and uses a customized CSR representation to achieve a high throughput. The design approach proposed in [24] is based upon serializing execution and processing of data within an engine and parallelizing access to off-chip memory. The processing engine sequentially issues multiple requests to memory and use on-chip RAM to store data from memory.

There are multiple reference implementations of the BFS implementation. In the top-down approach (Figure 4.1a), each parent vertex in the current queue visits all its children and adds them to a next queue then next becomes current. In the bottom-up

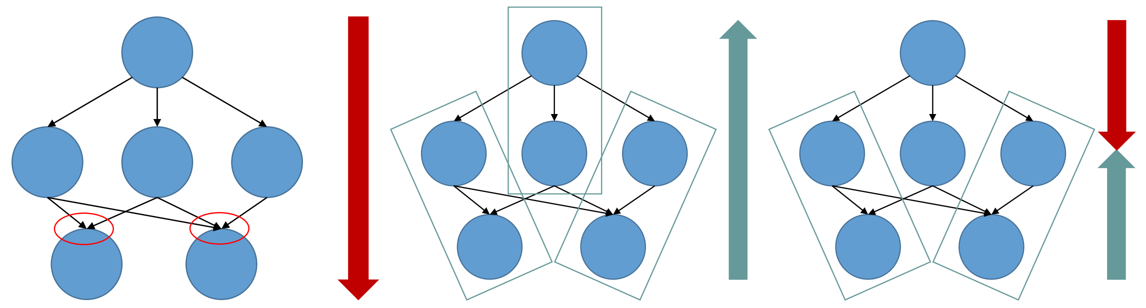


Figure 4.1: The three main approaches to the Breadth First Search algorithm. (a)Top Down starts at the root and visits each child per level. The highlighted intersections show the redundant work that top down creates and drops the efficiency of the algorithm. (b)Bottom up divides the graph into partitions and different threads continually process the same set of nodes checking if any parent was visited. (c)Hybrid Starts top down to generate a large frontier of nodes, then switches to bottom up to finish. Hybrid is generally the most efficient implementation of BFS.

approach (Figure 4.1b), the processor continually checks each unvisited vertex to see if it has a neighbor that was visited during the previous level. The work presented in [21] emphasized that both top-down and bottom-up approaches are beneficial when applied to different parts of the graph. This hybrid method (Figure 4.1c) starts with top-down traversal on the host side and switches to bottom-up on the coprocessor after a specific cut-off level. Based on the similar observation [128] came up with a hybrid approach to perform breadth first search with concurrent processing on both the host and the coprocessor and achieved significant performance. This method initially starts with the top-down approach on the host, copies the result in the coprocessor memory, which is later used by BFS. For larger frontiers, bottom-up BFS is said to be more efficient [21] because once the vertex has found a parent it need not check rest of its neighbors. Recently, this work[144] did an extensive comparison of each BFS approach on CPU-FPGA heterogeneous platforms.

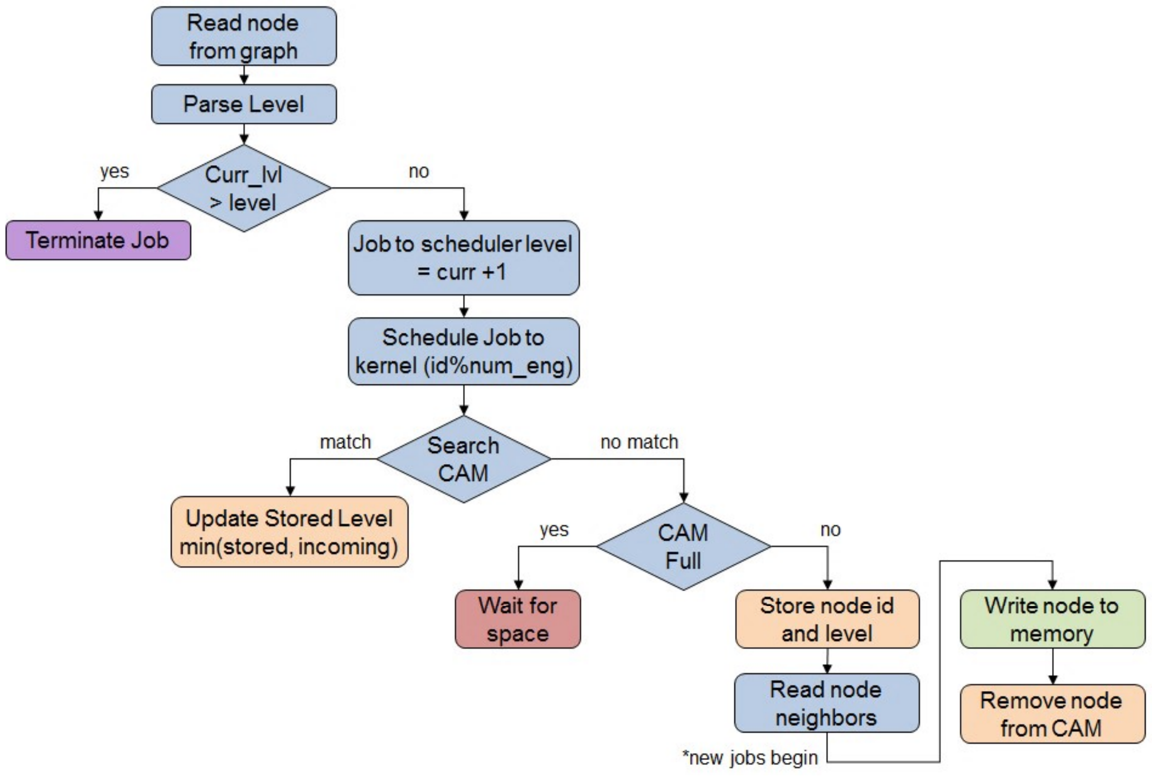


Figure 4.2: BFS Thread Flow

4.2 Hardware Multithreaded Breadth First Search

We designed the BFS kernel for large-scale graphs that would be too large to store locally on the FPGA. Since memory requests incur long latencies, we use the MT-FPGA approach to mask latency and utilize the available bandwidth. The implementation also uses custom CAMs as on chip synchronizing caches to reduce the number of memory requests and redundant jobs (Figure 4.1a). We accomplish this by allowing the kernels to merge any requests to the same node into a single job. This is an important optimization for handling nodes the have multiple shared children.

Figure 4.2 shows the flow chart of one job through the BFS engine. Each edge in the graph is a unique job and assigned a thread on the FPGA. We start by initializing the engine with the start node in the graph. Following the example graph in Figure 4.3 that would be node 0. The software hand-off to the FPGA initializes the start id to 0 and sets the level to 0. The scheduler assigns the node to its designated kernel (0 in this case) by doing the modulus of the number of engines. Since this is in binary and the number of engines is a power of two, modulus is simply a bitwise-and.

The kernel starts by adding an entry into its CAM for (id:0, level:1). It then requests the node data from memory. This will bring the stored level, count of neighbors, and the pointer to the neighbor list. Once all requests have been made, we set the level for the arriving nodes (node 0s level + 1) in a queue and the thread can write node 0s level to memory and terminate.

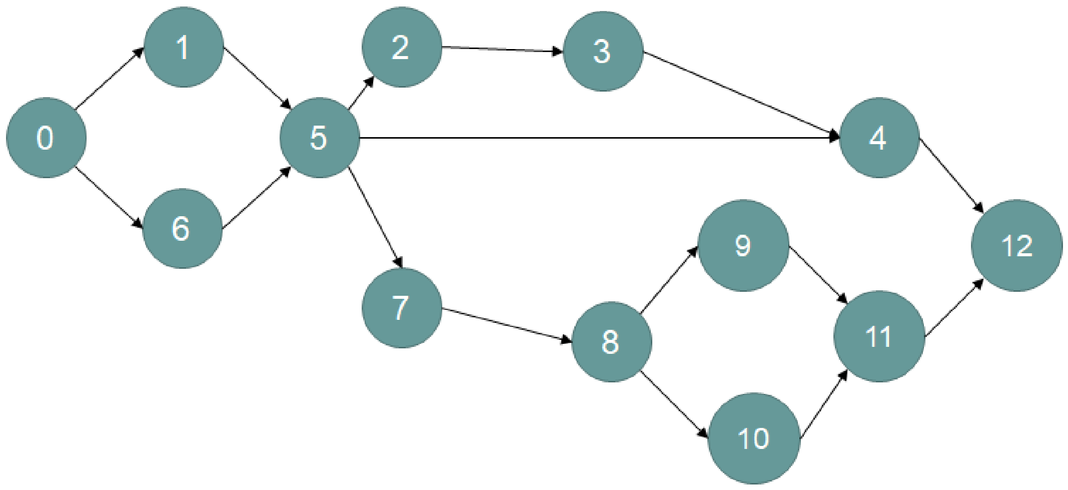


Figure 4.3: BFS Example Graph

Following the graph, the requests that node 0 made to memory would return nodes 1 and 6. Those flow through the ring network with the updated level information from node 0 and are scheduled to their respective engines. The same process happens for those nodes in parallel, and both request node 5. Eventually, both requests for node 5 are scheduled into kernel 5, where the jobs will merge in the internal CAM.

One of the limitations of a top-down approach to BFS is the enumeration of multiple redundant edges as nodes can have multiple common siblings. Using CAMs in this way can compress each of those possible jobs into a single job. This provides a mechanism to synchronize these threads on the FPGA instead of having to block on each thread finishing a level and writing out to memory. As long latency requests to memory are costly, so is synchronizing in memory.

4.3 Implementation

The left half of Figure 4.4 shows the in-memory graph representation. Much of the previous work on BFS relies on using a CSR representation for increased memory density. However, we argue that this representation is limiting to the representational power of graphs. This is evident with the recent growth in node-based graph databases and models like the Property Graph Model [12]. Our graphs our formed with the property graph model in mind, thus we use an adjacency-list style representation. It would only require a minor modification of the kernel to fetch additional key-value pairs from within the node object and support filtered graph searches.

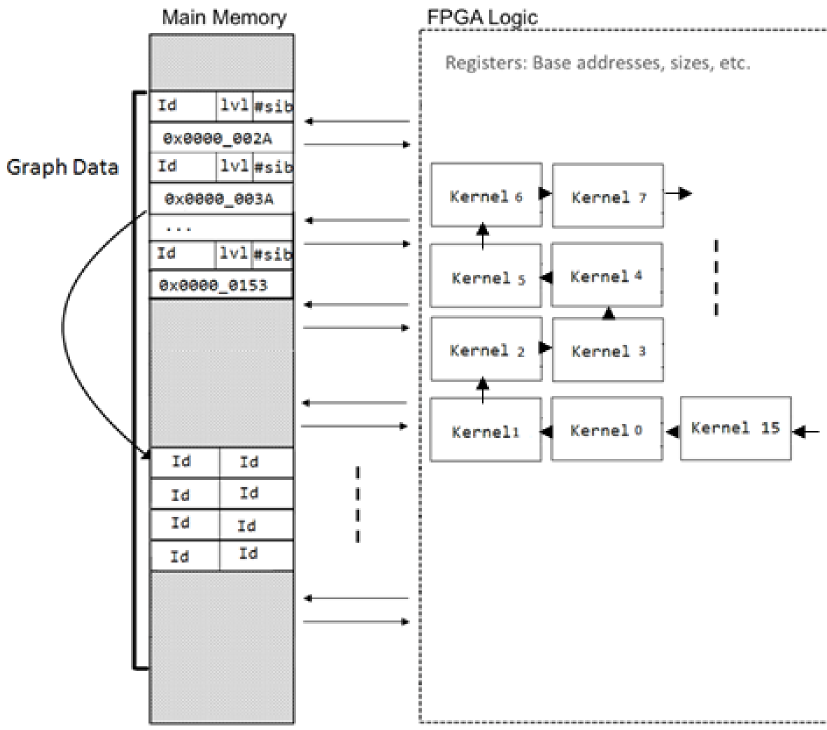


Figure 4.4: BFS Engine Layout

In this paper, we implemented the FPGA designs for the Convey HC-2ex platform; however, the designs are platform independent and only require in-order responses to memory requests to port to other platforms. The HC-2ex provides 4 Virtex 6 FPGAs, each with 16 independent memory channels connected through a full cross-bar with a theoretical peak bandwidth of 19.2 GB/s. Peak performance is dependent on the number of kernels used and the clock frequency. Since the kernel runs at a 150 MHz system clock, assuming no stalls, taking into account our 2 word node representation would provide a throughput of 75 Million Traversed Edges per second (MTEPS) per kernel. Since each kernel occupies only one memory channel, this design can scale to 16 independent kernels per FPGA, giving a theoretical peak of 1.2 GTEPS per FPGA. The right portion of Figure 4.4 shows the layout

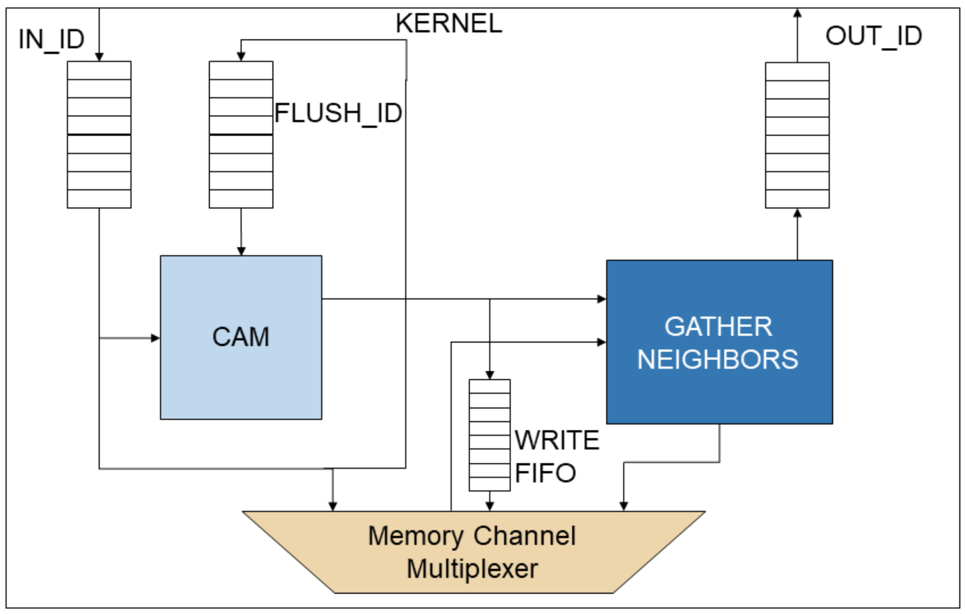


Figure 4.5: BFS Kernel Layout

and memory channels of the BFS engine and the use of a ring network to schedule the work of each kernel. This allows the kernels to work independently and not individually coordinate with each other. The kernel design is comprised of several FIFOs for coordination, a CAM, a BRAM, and a custom unit we call GATHER NEIGHBORS. A job enters the kernel and first checks the CAM if it can merge and terminate. Otherwise, it continues in the kernel fetching node information from memory. Once the memory responds, the kernel flushes the job from the CAM, checks if the node has already been visited, and either terminates the job or sends the neighbor list address to GATHER NEIGHBORS and writes the nodes level value to memory. GATHER NEIGHBORS, issues requests for all the IDs in the neighbor list and sends the responses back to the scheduler with the previous nodes level plus one. The layout can be seen in Figure 4.5.

4.4 Experimental Evaluation

Throughput and CAM Utilization

For this project, we experimented on the usefulness of the CAM by running our system on random and R-MAT graph data using GT-graph, a suite of synthetic graph generators[15]. R-MAT graphs exhibit the power law property, which makes them useful for testing since many real-world graph relationships also follow a power law.

Graph Details					Merge Threads			
	Nodes	Edges	Max In-flight Requests	MTEPS	Update	Same Level	Stale	% Merged
RMAT	32	128	68	2.5	15	41	29	66%
	64	512	198	9.1	2	247	67	62%
	1k	32k	1476	17.5	5563	12067	7947	80%
Random	1K	8k	1312	31.97	257	4328	738	65%
	2k	32k	1558	35.6	2421	10579	2909	48.5%
	4k	64k	1798	37	1520	38337	4113	67%

Table 4.1: Throughput and merged jobs for BFS on various graphs with 8 FPGA kernels

Table 4.1 shows the graph details, throughput in MTEPS, and the usefulness of the CAM for a handful of smaller graphs. First note the Max In-flight requests column. This is directly tied to the multithreading model, and we need to be able to maintain significant outstanding requests to mask the latency. For a system with an average of 150 cycle response latency, for 8 engines we need at least $8 * 150 = 1200$ outstanding requests to fully mask latency. As our graph size grows, we are able to sustain at least that many requests. As this number grows, we also see the throughput(MTEPS) grow as well, showing that our throughput increases as we stay in the steady state longer and amortize the build-up and tear-down phases. Finally, The far right of the table shows the counts of jobs that were able to be terminated early in the CAM. There are three cases: (a) update jobs that come into the engine with a lower level than is currently stored.

FPGA Area Utilization

# Kernels	Registers	LUTs	BRAMs
1	6,595 (0.7%)	12,790 (2%)	3 (0.1%)
4	27,011 (2%)	56,673 (11%)	10 (1%)
16	104,776 (11%)	216,215 (45%)	40 (5%)

Table 4.2: FPGA Resource utilization.

Table 4.2 shows the resource utilization (registers, LUTs and BRAMs used) for the different FPGA designs with scaling number of engines. The use of resources scales linear with the number of engines, which is a positive sign for routability. It is also worth noting that device usage is reasonably small, with 16 engines taking less than 50% of the device LUTs. It would be possible to further explore expanding this system by adding more engines and multiplexing requests over memory channels. This would increase the number of concurrent threads, increase the number of outstanding requests, and ensure that the system could spend more time in the steady state fully masking memory latency.

4.5 Conclusion

In this chapter, we have motivated the use of CAMs as synchronizing caches for irregular applications. In conjunction with using a highly multithreaded datapath to support hundreds of threads masking memory latency, the CAMs can take advantage of this long latency to merge jobs and reduce the number of memory requests overall. We demonstrated this design using a breadth first search through a graph, using a graph representation that could easily expand to support the property graph model and richly annotated graphs. This work showed the resource usage and estimated performance of a few engines on a single FPGA, and showed that it could scale to using all the FPGAs on the Convey HC-2ex to completely utilize all available memory bandwidth and provide good throughput for graph-based algorithms. Future work could look at how we could expand this technique to improve the throughput of a bottom up approach to BFS.

Chapter 5

Processor-Side Locking for FPGA Multithreading of In-Memory Hash-based Operators

Joins and group-by aggregations are two memory intensive operators affecting the performance of Relational databases. Their efficient implementation becomes even more important as we enter the Big Data era, where systems need to deal with increasingly larger datasets. Recent paradigm shifts in multi-core processor architectures have reinvigorated research into how the join and aggregation algorithms can leverage these advances. The FPGA community has also been developing new architectures with the potential to push performance even further. While hashing is a common approach used to implement both operations, its poor spatial locality can hinder performance on multi-core processor architectures which rely on mitigating the latency by using large cache hierarchies. Multithreaded architectures can better cope with poor spatial locality by masking memory/cache latencies with many outstanding requests. However, the number of parallel threads even in the most advanced multithreaded processors (UltraSPARC) is not enough to fully cover the main memory access latency. Instead, the hardware reconfigurability of FPGAs enables deeper execution pipelines that maintain thousands (instead of tens) of outstanding memory requests - drastically increasing concurrency and throughput.

In this chapter we present extensions to our first end-to-end in-memory FPGA hash join implementation. The FGPA uses massive multithreading during the build and probe phases to mask long memory delays, while concurrently managing hundreds of thread states locally. When considering "skinny" relations throughput results show a speedup between 2x and 3.4x over the best multi-core approaches with comparable memory bandwidths on uniform and skewed datasets; however, this advantage diminishes for extremely skewed datasets. We also provide an FPGA hash join implementation that can handle larger key sizes and arbitrarily wide tuple sizes. Using the TPC-H benchmarks the FPGA implementation shows around 1.5x speedup over the best multi-core implementations.

We further present extensions to our first end-to-end in-memory FPGA group-by implementation. In this context we explore how Content Addressable Memories (CAMs) can be intermixed within our multithreaded designs to act as a synchronizing cache, which enforces locks and merges jobs together before they are written to memory. Even though CAMs limit the number of active jobs to a few hundred, leveraging them within our aggregation implementation achieves speedups up to 10x in terms of throughput over CPU implementations across 5 types of data distributions.

5.1 Related Work

5.1.1 Hash-Join

Many recent works consider the in-memory implementation of join and aggregation relational operators (hash- or sort-based). The seminal paper by Manegold et al. [91] emphasized the importance of TLB misses for in-memory database operations and proposed a *radix clustering* algorithm to keep the partitions cache resident. Later performance of hash joins were studied [25] primarily by comparing simple hardware-oblivious algorithms and *hardware-conscious* approaches (since the radix clustering algorithm is tightly tailored to the underlying hardware architecture). The experimental results showed that the simple implementations surpass approaches based on radix clustering. However recently, Balkesen et al. [17] applied a number of optimizations and found that hardware-conscious solutions are in most cases prevalent over the hardware-oblivious in terms of throughput.

Sort-merge joins on modern CPUs were initially considered by Kim et al. [82]. This implementation explored the use of SIMD operations and hypothesized that sort-merge join performance will surpass the hash-based algorithms, given wider SIMD registers. Subsequent work [10] 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, Balkesen et al.[18] 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. Kocberber et al.[84] demonstrated a latency masking multithreaded version of hash-join on the Sparc T4 under the name of "Asynchronous Memory Access Chaining".

5.1.2 Aggregation

In addition to joins, group-by aggregation operator, relying on multi-threaded architectures to boost its performance, was also extensively researched. One of the earliest works [42] explores different aggregation implementations on chip multiprocessors (CMPs) and concludes that performance largely depend on input characteristics like key cardinality, thus opting for adaptive strategy based on sampling. However, this work focuses mainly on non-partitioned versions of algorithms. Follow up work from the same authors [43] specifically explores the partitioning step of hash aggregation in the same CMP environment and, in line with [91], emphasizes the thread coordination as a key component of this step. The work by Ye et al. [138] considers both partitioning-based and non-partitioned aggregation implementations and proposes several hybrid approaches, which outperform previous implementations on Intel Nehalem architecture. Finally, Wang et al. [129] describes novel NUMA-aware partitioned in-memory hash aggregation algorithm, which avoids cache coherency misses and minimizes locking costs.

While the software community has examined both hash and sort-merge for join and aggregation operators the FPGA community has concentrated on sort-merge approaches. The reasons for this are twofold. Firstly, sorting and merging implementations are easily parallelized on FPGA architectures. For example, sorting networks like bitonic-merge [76] and odd-even sort [86] are well established designs for FPGAs; Casper et al.[37] developed a multi-FPGA sort-merge algorithm, while other works [114, 135] used sort-merge as part of a hardware database processing system. Secondly, building an in-memory hash table efficiently is non-trivial task because of the required synchronization.

Commercial platforms like IBM's Netezza[75] and Teradata's Kickfire[81] 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 [28, 71, 94], 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 join and aggregation, which has been historically the most time intensive operations in analytical query workloads.

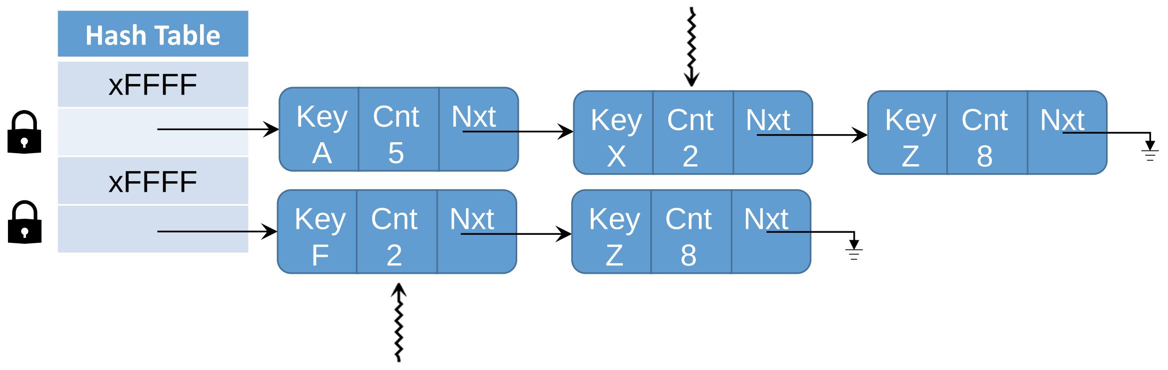
An FPGA-accelerated implementation of group-by aggregation was first considered by Mueller et al. [100]. This work also utilized CAMs in the implementation of the aggregation operator, but in a very narrow scope, i.e. using CAMs to match an incoming tuple with the appropriate group. Hence the work continued long tradition of using CAMs to answering set-membership queries (previously explored in applications like click-fraud, online intrusion detection [19]). Our design also uses CAMs, but is different from previous approaches in two ways: (i) in addition to the key we store and update the aggregate value locally in the CAM, and (ii) we use CAMs as a synchronization primitive to resolve conflicts during updates. Further details and related work on CAMs can be found in Section 2.6.

5.2 Group-By Aggregation on FPGA

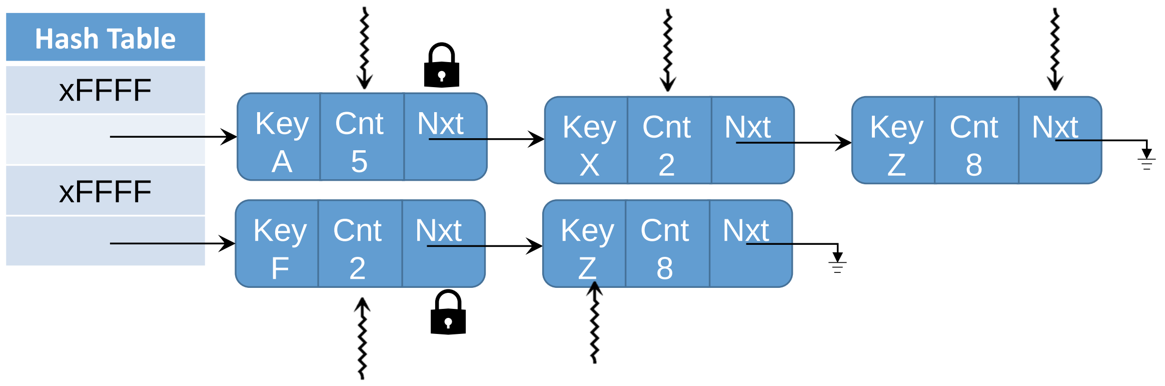
In our design we assume the input relation fits in main memory but is too large to fit locally on the FPGA's memory. The mixed read-write nature of aggregation in conjunction with multiple outstanding requests requires us to use explicit synchronization to ensure correctness. Using atomic operations is one option, but this approach severely impacts the performance. Moreover, unlike the join operator, aggregated tuples exhibit temporal locality. We propose a novel multithreaded aggregation implementation based on CAMs. The design leverages explicit synchronization combined with the cache-like properties of the CAM. This fits perfectly in the context of group-by aggregation: firstly, the latency of a single aggregation job is hundreds of cycles, which means many interleaved jobs can have identical keys. With a CAM we can merge these jobs pre-aggregating the result locally on the FPGA and reduce the number of outstanding memory requests. This merging is achieved by leveraging cache properties of the CAM (allowing us to hold the aggregate value for a particular key). It also allows up to alleviate skewed data distributions, where a subset of values appears as duplicate more often than the rest. Secondly, CAMs allow the FPGA to enforce locking on specific memory channels, therefore decrease granularity of the locks and boost the performance.

5.2.1 Fine Grained Locking

In the original aggregation design[8], our locks were implemented at the granularity of hash table buckets (figure 5.1a). This guaranteed that only one thread was working on a list in the hash table at a time and it was free to modify the list as needed. With exclusive



(a) Coarse grain locks stop threads at the bucket level and prevent concurrent searching through the linked list.



(b) Fine grain threads lock at the node level, increasing thread concurrency and only synchronize structural changes.

Figure 5.1: Multithreaded Architecture details

access to the list, the threads can perform node inserts in sorted order to improve the merge phase. However, such coarse-grained locking has a big impact on the parallelism the system is able to achieve. This is especially noticeable on skewed datasets where a majority of keys might map to the same bucket. All of those threads must stall and wait for the previous thread to finish. And the wait for each thread increases hundreds of cycles for each node added to the list. Each thread must pay this penalty even if it is only going to increment the count in a node and not modify the list structure.

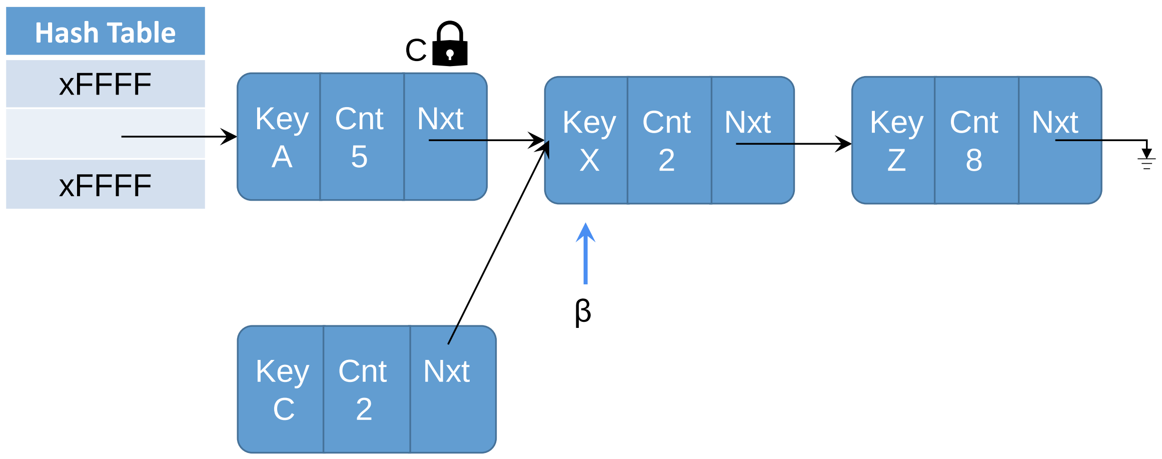


Figure 5.2: Lock free reads. Thread β is reading past the lock C has on node 1.

The first insight motivating the fine-grained locking comes from the benefits of the top level filter CAM. All new tuples that enter the aggregation start at the filter CAM. If the key already exists, it is incremented in the CAM and the thread terminates. If the key does not exist in the filter CAM and there is space, the key is added and a thread starts the hash table search. This construction guarantees that all threads searching the hash table are unique - they will never try to update the count in the same node because they all have different keys. We can take advantage of this design and move the lock lower to node pointers (figure 5.1b). This enables synchronizing where it matters, when a thread wants to insert in the list and make a structural change.

Lock Free Reads

If we only lock for structural changes (inserting a node), that means only writes are protected. That begs the question - is it safe for threads to read past a lock? Consider the possible situations of a thread progressing down a list (figure 5.2). There are only three possible outcomes:

1. $\boldsymbol{\beta > x}$: C's lock on node 1 is irrelevant and it is safe for β to proceed. It will either find the key or need to insert later in the list.

2. $\boldsymbol{\beta = x}$: In this case, β has found its node and can increment the count without synchronization.

3. $\boldsymbol{\beta < x}$: Thread β has progressed too far and must insert. However, insert is gated by locking the next node of the previous pointer. It will safely synchronize on C's lock and will try again after the lock is free.

There are several benefits to using this design where reads are not locked. First, at no point does a thread need to stall until it needs to insert. If reads had to wait for the lock, these fine grain locks could deteriorate to behavior like the coarse grain lock - a thread blocks the start of the list and all work stops. However, the more important benefit comes from the behavior of the aggregation algorithm. For any given aggregation, the cardinality of the key can be much smaller than the size of the data table that is being scanned. Therefore, there will be a slightly contentious period at the beginning of the execution where keys are getting inserted. But once all keys have been seen once and inserted in their respective locations in the table, the rest of the execution will proceed without any locks.

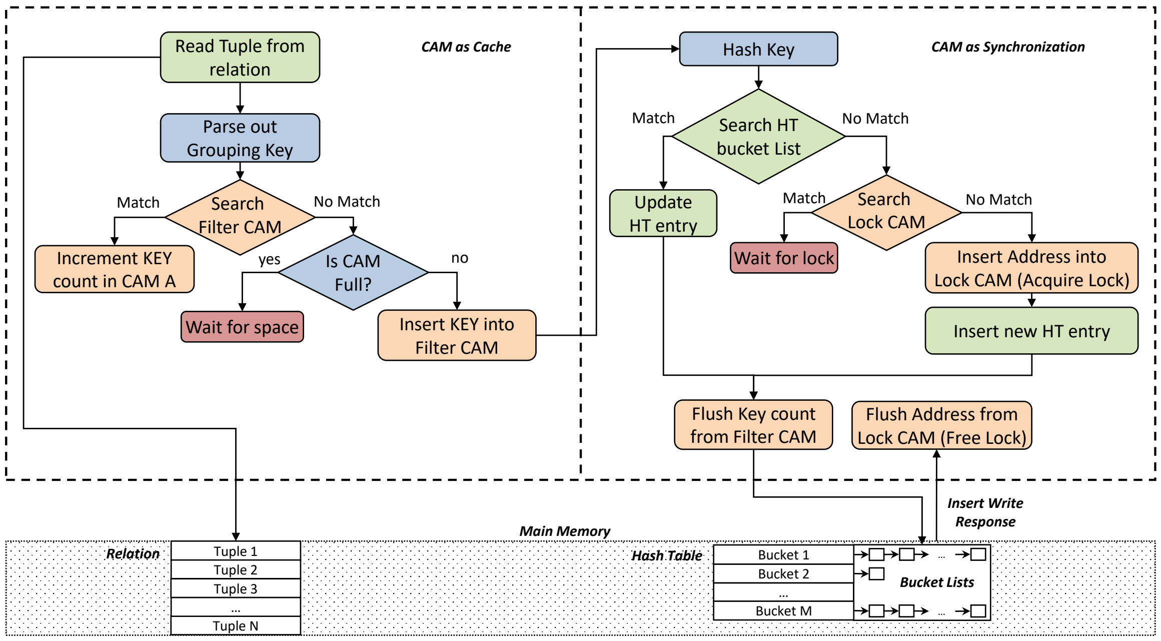


Figure 5.3: A state diagram for jobs in the aggregation engine.

5.2.2 Aggregation Engine Workflow

Our design of an aggregation operation uses a custom hardware datapath called *aggregation engine*. Initially each tuple from the relation is streamed from memory, gets assigned to a separate FPGA thread (job) and starts its pipelined execution. Figure 5.3 shows the state diagram for a single thread inside the aggregation engine. The *Filter CAM* is used to merge jobs with identical keys, hence reduces the memory request contention and minimizes the synchronization overhead. However due to hash collisions the synchronization cannot be avoided completely; thus the *Lock CAM* is used to acquire locks on the hash table, ensuring its integrity.

Table 5.1 shows an example of events and contents of *Filter* CAM, *Lock* CAM and main memory HashTable, while the input stream consists of 5 tuples with the following

Cycle	Key	Filter CAM	Lock CAM	HashTable	Comments
1	A	*Miss, Insert (A,1)* {(A,1)}	{}	{}	Request to search key A in HT is sent
2	C	*Miss, Insert (C,1)* {(A,1),(C,1)}	{}	{}	Request to search key C in HT is sent
3	A	*Hit, Update A* {(A,2),(C,1)}	*Miss, Job 1 locks hash(A)* {hash(A)}	{}	*Job 3 is discarded* Key A not found in HT, $Bucket_{hash(A)}$ is locked
4		*Job 1 clears key A* {(C,1)}	*Hit, hash(A)=hash(C)* {}	{(A,2)}	Create new entry (A,2) in HT Key C not found in HT, *Job 2* waits for lock
5	B	*Miss, Insert (B,1)* {(C,1),(B,1)}	*Job 1 frees lock on hash(A)* {}	{(A,2)}	Request to hash(C) in HT is sent *Job 2 restarts at previous address*
6	A	*Miss, Insert (A,1)* {(C,1),(B,1),(A,1)}	{}	{(A,2)}	Request to search key B in HT is sent *Job 2 reaches end of list*
7		{(C,1),(B,1),(A, 1)}	*Job 2 locks node(A).next* {*node(A).next*}	{(A,2)}	Request to search key A in HT is sent *node(A).next* is locked
8		*Job 2 clears key C* {(B,1),(A,1)}	*Job 2 frees lock for key C* {}	{(A,2),(C,1)}	Create new entry (C1,1) in HT Key B not found in HT, CAM busy
9		*Job 5 clears key A* {(B,1)}	*Job 4 locks hash(B)* {hash(B)}	{(A,3),(C,1)}	Key A found Update key A in HT to (A,3)
10		*Job 4 clears key B* {}	*Job 4 frees lock for key B* {}	{(A,3),(C,1),(B,1)}	Create new entry (B,1) in HT

Table 5.1: Contents of the *Filter CAM*, *Lock CAM* and HashTable (HT) and *modifications* altering all of them, while relation with the following keys is processed: *A*, *C*, *A*, *B*, *A*. Assume hash(A)=hash(C). Initially both CAMs are empty. *Filter CAM* maintains the occurrence of duplicate keys, while *Lock CAM* locks linked-list next pointers

keys: A, C, A, B, A. The design assumes the COUNT aggregation function, thus the *Filter CAM* maintains an occurrence count of duplicate keys. However, other functions could be potentially applied. Note that operations updating the CAMs are performed immediately, whereas main memory HashTable accesses (e.g., search, entry update, entry insert) take several cycles to finish. For example, *Job 1* sends a request to search value A in a hash table and gets response only at $Cycle_3$. *Lock CAM* maintains the locks for all addresses which are currently being modified. Notice that all jobs only acquire locks after searching memory. Locks are only needed when creating a new node. Even though both *Job 1* and *Job 2* need to search the same bucket, they won't synchronize until after finding the list is empty and trying to add a new node. *Job 1* finishes first and is able to get the lock and *Job 2* finds it must wait in the next cycle. Once a job completes, it invalidates the record in both CAMs and frees up resources for other jobs. Jobs, waiting for a place in a CAM, will continually cycle through a FIFO until the resource is available. Whenever there is a hit in the *Lock CAM* the job waits until the lock is released, e.g. *Job 2* resumes its work only at $Cycle_5$. *Job 3* provides an example of early termination, because its value was locally aggregated in *Filter CAM* in $Cycle_3$. After $Cycle_5$ we see more concurrency as 3 threads are searching the list. *Job 2* provides an example of fine-grained locking in $Cycle_7$. The thread gets to the end of the list and locks the next pointer of node A. At the same time, *Job 5* is able to find node A in the list and update its count without any locks. Finally, *Job 4* is able to finish in $Cycle_{10}$ after a long memory request and waiting for a free cycle in the *Lock CAM*.

5.2.3 FPGA Design Optimizations & Trade-offs

The main factor limiting performance of this memory-bounded problem the is efficient use of available memory bandwidth. In this paper we use a Convey-HC-2ex machine, but our designs are platform independent. In the Convey the communication between the FPGA and main memory relies on the abstraction called *channel*. Each channel supports independent and concurrent read/write accesses to memory. The original design of our aggregation engine required 4 memory channels: one for streaming the input tuples, one for accessing the in-memory hash table, and finally two channels for the bucket lists read/write operations. Since the Convey-HC-2ex has 16 memory channels, we replicated 4 engines ($\frac{16}{4}$) on a single FPGA thus leveraging inter-engine parallelism. Our original experiments showed that some memory channels were idle for almost 70% of the total execution time. Since the channels within an engine are statically assigned to perform different functions of the pipeline, back pressure from some components (e.g. job recycling through CAM synchronization) introduces stalls and decreases the effective throughput.

In order to increase memory utilization we then *multiplexed* a pair of engines on the same set of memory channels, thus allowing the same channel to be used by two different engines. This means that the following engine operations (e.g. send and receive tuple request and response, read and write respective values to the hash table, read and write entries into respective bucket list) can run concurrently on two different engines. The multiplexed design increases the number of CAMs that could be placed on the FPGA, leading to further improvement in throughput. Unlike the original design, the new multiplexed engine uses 5 memory channels (adding an extra channel for accessing the in-memory hash table). This allows us to place 6 engines ($2 * \lfloor \frac{16}{5} \rfloor$) on a single FPGA.

In this latest design, to further improve the utilization of memory bandwidth, we have reduced the number of memory channels down to 2 per engine. Each engine uses a channel for streaming in the tuples, and all requests to the hash table and linked lasts are multiplexed internally through another channel. This construction enables up to 8 engines per FPGA. While multiplexing more requests over a given channel likely increases the latency of any given job, it enables the engine to always have a request available to issue and can keep the engine in the steady state longer. As the percentage of execution time in the steady state increases relative to the build-up and drain-out phases, the overall throughput of the system increases.

Each engine uses its own CAM for synchronization. As a result, values are aggregated in separate hash tables, which requires an extra merging phase at the end of the computation. Merging overhead grows as we increase the number of engines per FPGA, but it is an overhead that is amortized as the size of the dataset grows.

5.3 Experimental Results

We chose the Convey HC-2ex as our target FPGA platform because of its high bandwidth memory access. In particular, the memory system that interfaces to the FPGA allows up to 16 concurrent memory requests per cycle per FPGA. The FPGA aggregation implementation is compared in terms of overall throughput against the best multi-core approaches [42, 138] running on a single processor with 4 parallel threads. We proceed with a short description of the Convey HC-2ex, followed by a summary of the various software aggregation algorithms as well a description of the datasets used in the experiments.

5.3.1 Software Implementations

In order to evaluate our FPGA-based solution we have implemented the following state-of-the-art multithreaded software aggregation algorithms: (i) Independent Tables[42], (ii) Shared Table [42], (iii) Hybrid Aggregation [42], (iv) Partition with Local Aggregation Table [138] and (v) Partition & Aggregate [138]. Here, (i) and (ii) are considered as non-partitioned approaches, while (iii) and (iv) are hybrid, and (v) is a partitioned approach.

- **Independent Tables** [42] is the approach most similar to our hardware implementation. The tuples are evenly split among separate software threads (without partitioning), and each thread aggregates result into its own hash table. Once the aggregation is complete all tables are merged together, which requites write synchronization.
- **Shared Table (with locking or atomic synchronization)** [42] splits the tuples evenly between threads, but all threads aggregate their results into a single hash table, hence no extra merge step is required. The algorithm could use different synchronization primitives: either pthread mutex implementation or Intel-specific hardware atomic instructions. Preliminary experiments showed that atomic primitives are significantly better on low key cardinalities, and don't have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments.

- **Hybrid Aggregation** [42] is a combination of two previous approaches. This algorithm allocates a small hash table for each thread. The size of the table is calculated based on the processor's L2 size to avoid cache misses. If the local table has enough space for a new value, or the value already exists in the table, that tuple is locally aggregated. Once the local table is filled and the next tuple requires a new slot, the oldest entry in the cached table will be spilled into larger shared table, residing in main memory, thus maintaining only "hot" data in L2 cache. Once aggregation is complete all small cached tables are merged into the large shared table. Merge step is synchronized as in *Independent Tables* case.

- **Partition & Aggregate** [138] (also known as count-then-move [43]) uses individual tables per thread, but before aggregation is performed the tuples are partitioned, in contrast to all aforementioned approaches. Separate partitioning step makes sure that all threads will work on non-overlapping values, hence aggregation could be done without any synchronization and the final tables are simply concatenated, rather than merged. As with the partitioned join implementations *radix clustering* algorithm is a backbone of this preliminary step.

- **PLAT (Partitioning with Local Aggregation Table)** [138] is a combination of two previous techniques. The algorithm takes advantage of the fact that we are performing an additional data scan, while doing a preprocessing step. While partitioning tuples into groups with mutually exclusive keys, each thread tries to aggregate values into its own small L2-resident table, as in *Hybrid Aggregation* approach. Values that do not fit into the small table are partitioned using *radix clustering* algorithm. Once preprocessing is done standard lock-free aggregation is applied. In the end all tables, which were produced during aggregation, are concatenated together, while local aggregation tables are synchronously merged in.

5.3.2 Dataset description

We use five datasets with various s key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [42], Self Similar and Zipf_0.5.

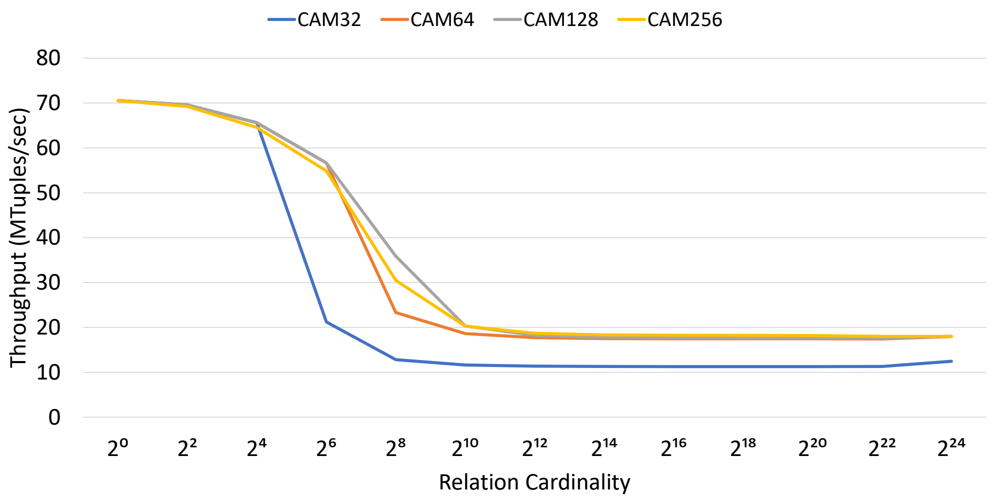
- In the **Uniform** dataset all key values are picked from *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.
- A half of the tuples in the **Heavy Hitter** dataset [42] share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the the entire relation.
- In the **Moving Cluster** dataset [42] tuples are grouped into clusters depending on their key values. Lower key values are more likely to appear at the beginning of the relation, whereas tuples with higher key values are tend to appear at the end of the relation.
- **Self Similar** uses Pareto rule to model key distribution in a dataset: a single key value is shared by 20% of the tuples. Of the remaining 80% of tuples 20% of those share another key value. This process is repeated recursively to generate the relation. Tuples are randomly shuffled. The generation algorithm is described by Gray et al. [59].

- In the **Zipf** dataset key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work[59].

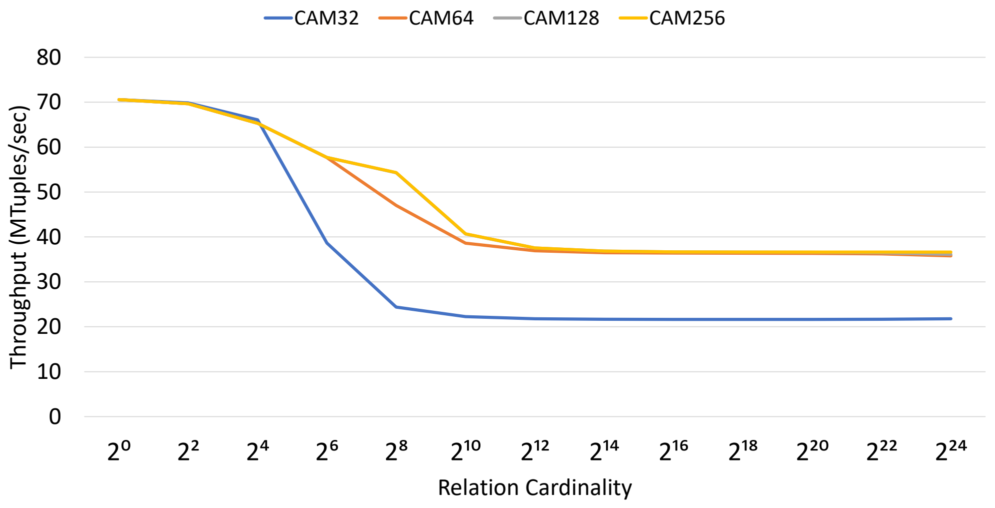
Each dataset consists of several benchmarks with cardinalities ranging from 2^{10} to 2^{22} unique keys. The relation size in all of the experiments was 256 million tuples (in line with previous research [138]). Each dataset used the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query processing algorithms [18, 26, 25] and represents a popular column-wise storage format. The first 4 bytes of the tuple hold the unique primary key, while the rest is reserved for the grouping key. Since we are only interested in counting records with the same grouping keys, our tuples do not store any other information. However, none of the design choices prevent the use of "wide" tuples (i.e. containing fields other than primary and grouping keys). This could be easily supported by adding a key extraction component into the FPGA design. Moreover experimenting with such "skinny" tuple format yields the best performance for software implementations, since it minimizes the cache capacity misses, which would decrease caching effectiveness otherwise.

5.3.3 Effect of Filter CAM size

The throughput of a multithreaded engine is dictated by the number of threads needed to fully mask latency. In this fine-grained locking engine, one of the key controls on the number of threads concurrently working is the size of the Filter CAM. Since every entry in the Filter CAM starts a thread searching the hash table for a node, we started by experimenting on the effect of the CAM size on throughput. Figure 5.4 shows how varying



(a) Uniform



(b) Heavy Hitter

Figure 5.4: Aggregation throughput of single engine for 256M tuples as *Filter CAM* size is changed.

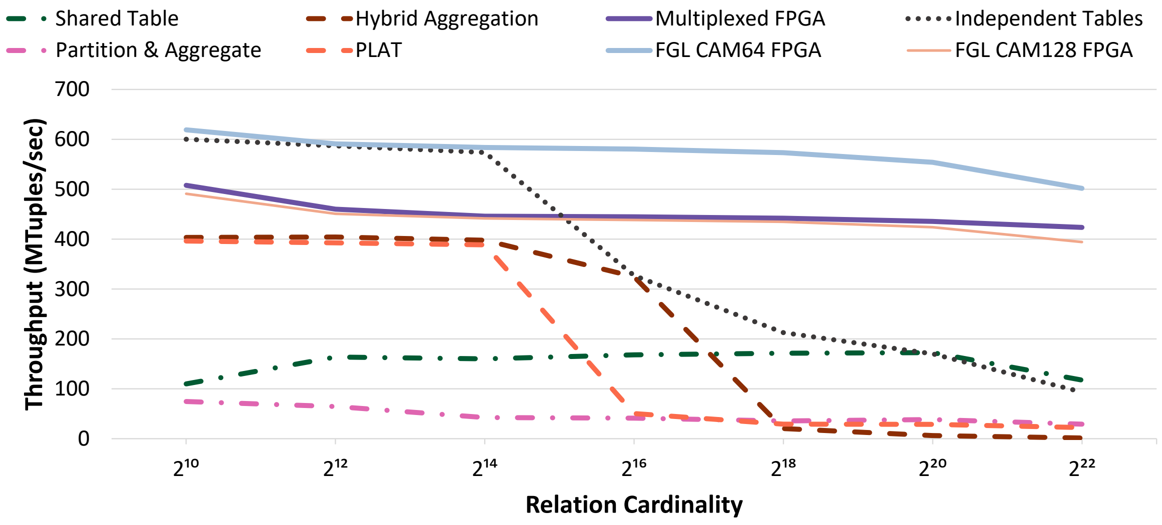
the CAM size changes throughput for two of the data distributions. The other distributions show similar results as these two graphs depending on their skew.

For the uniform distribution (figure 5.4a) there is a sharp drop in throughput where cardinality grows larger than the Filter CAM size. As we increase cardinality, the uniform distribution means that there will be little temporal locality in the tuple stream and pre-aggregation provides little help. As the CAM size increases there are diminishing returns on the throughput. A CAM size of 32 definitely drops the performance, but 64 or 128 provide similar numbers. Especially for larger cardinality where hash table searching dominates.

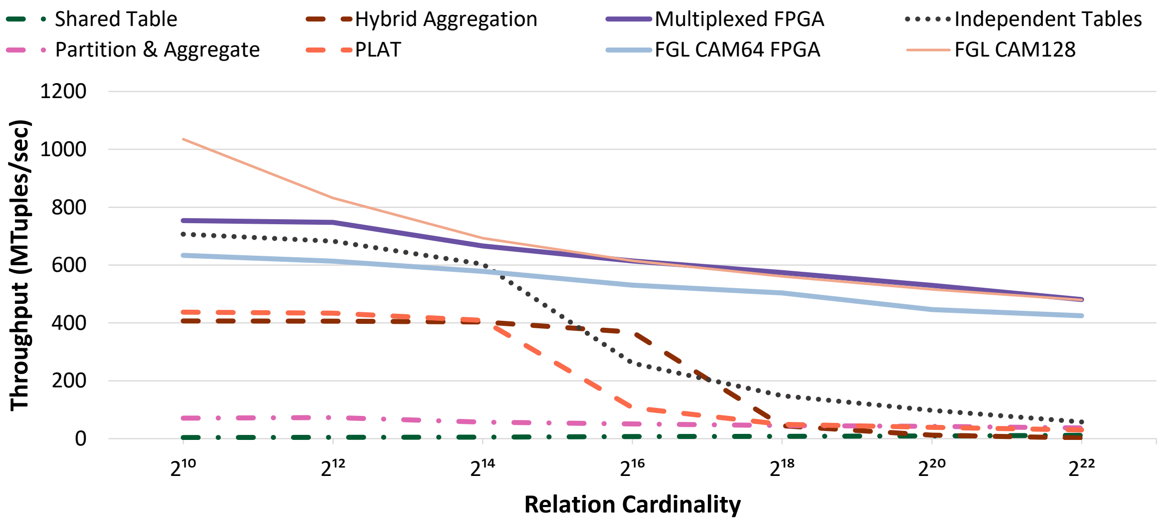
For the heavy hitter distribution (figure 5.4b) there is a similar drop in throughput where cardinality grows larger than the Filter CAM size. However, in this instance the Filter CAM size definitely affects the achievable throughput when hash searching dominates. A CAM size of 32 keeps the throughput similar to the uniform distribution. Cam sizes of 64 and above nearly double the throughput as the Filter CAM is able to provide some benefit in the skewed data. Again, there are diminishing returns for the CAM size above 64.

5.3.4 Throughput Evaluation

Figure 5.7 displays the throughput of the group-by aggregation as the key cardinality is increased, obtained for various datasets. Throughput was measured across 2 FPGA engine designs (multiplexed[8], Fine-Grain Locking with CAM64), and five software (two non-partitioned, two hybrid and one partitioned) implementations. We chose to implement the Fine-grain locking engine using CAM 64 because the smaller CAM is easier to meet timing and get more engines routed on the FPGA. FGL-CAM64 has 8 engines per FPGA. Throughput for skewed Heavy Hitter dataset Figure 5.6b resembles the results for Self Similar dataset Figure 5.5b, while the throughput for moderately skewed data Zipf_0.5 5.7a is similar to the results obtained for Uniform dataset Figure 5.5a. Software implementations



(a) Uniform

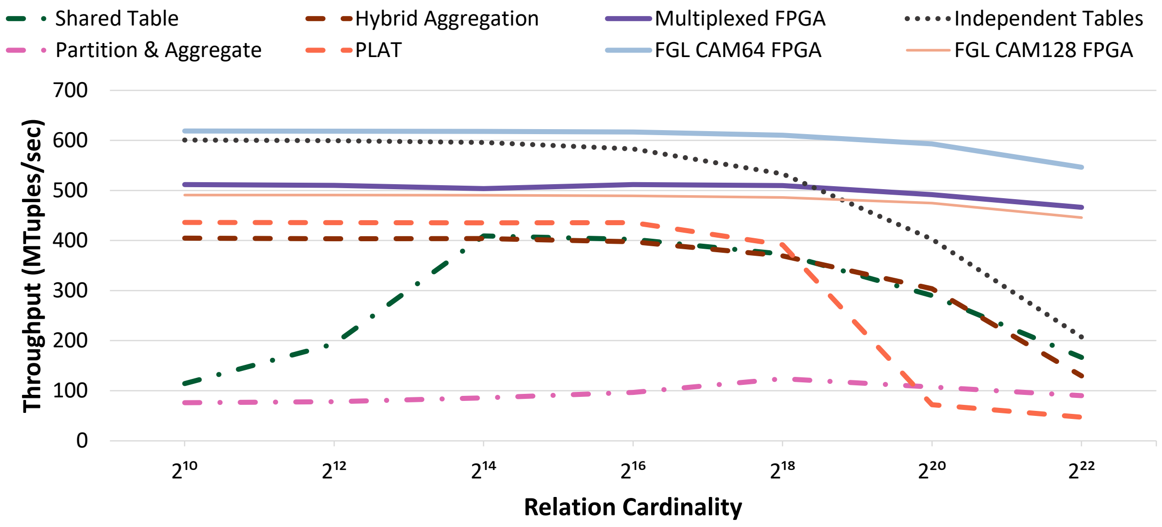


(b) Self Similar

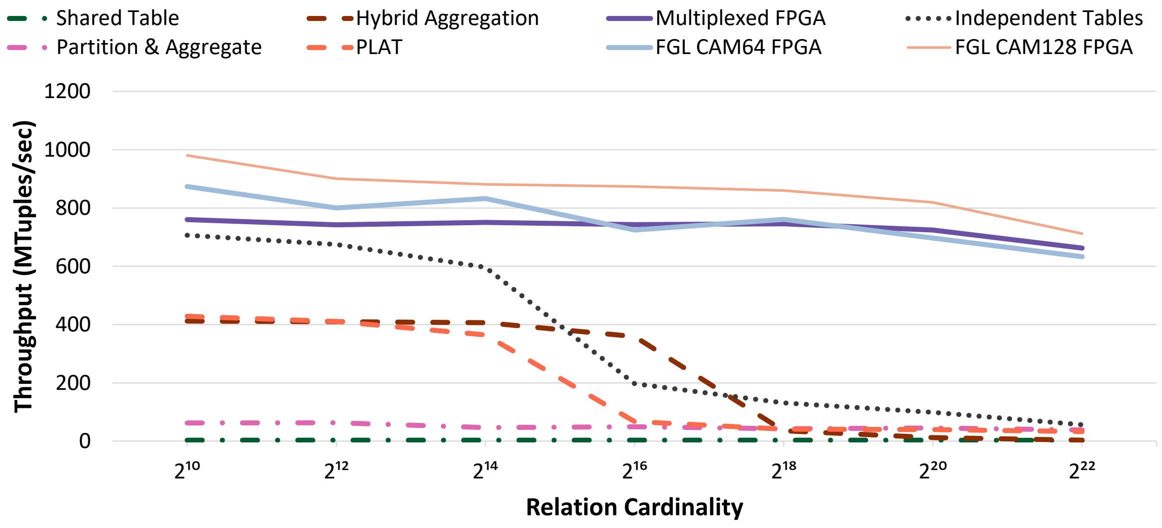
Figure 5.5: Aggregation throughput for uniform and self similar datasets with 256M tuples

demonstrate the best performance on Moving cluster dataset Figure 5.6a due to the property of the data distribution: similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.

Despite all the differences in data distribution CPU aggregation performance mainly depends on the dataset's key cardinality. While the number of unique keys is low,



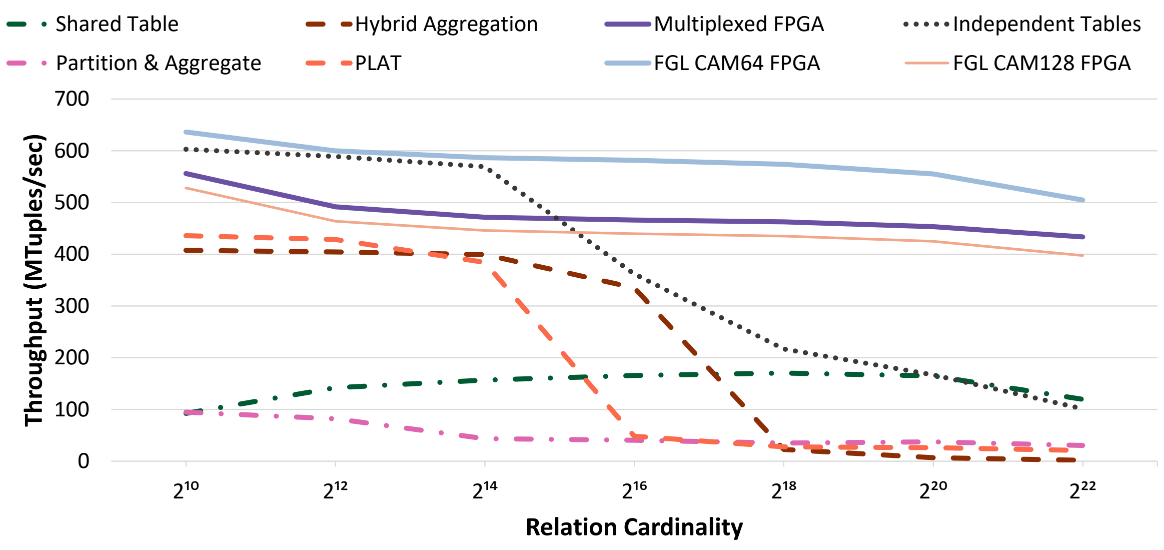
(a) Moving Cluster



(b) Heavy Hitter

Figure 5.6: Aggregation throughput for moving cluster and heavy hitter datasets with 256M tuples

hash tables can fit into the CPU cache entirely. However, as the cardinality increases, cache misses start to hamper the throughput due to high latency memory round-trips. Software performance severely deteriorates at cardinalities higher than 2^{18} on all datasets for all algorithms. Another trend, which appears in all experiments, is that the Independent Tables approach yields the best result across all software algorithms. Nevertheless, that algorithm



(a) Zipf 0.5

Figure 5.7: Aggregation throughput for moving Zipf 0.5 datasets with 256M tuples.

exhibits poor scalability, since the amount of memory needed for aggregation processing grows linearly with the number of parallel threads and the key cardinality. As the number of parallel threads increases, the amount of available memory could quickly become a bottleneck. We could also see that hybrid algorithms (PLAT and Hybrid Aggregation) outperform traditional partitioned (Partition & Aggregate) and non-partitioned (Shared Table) approaches by amortizing the cache miss cost and sustain a throughput around 400 MTuples/sec. This trend continues for cardinalities up to 2^{16}, which marks the end of L3-cache residency. After that point the performance advantage of hybrid algorithms vanishes and drops below 100 MTuples/sec.

The FPGA performance also drops as the key cardinality increases, however this effect is much less profound. Unlike the software throughput, this result is explained by the overhead, introduced by the post-processing merge step. However the overall performance is still up to 10x higher than the software throughput. The results also clearly show the benefits of the fine grained locking over the multiplexed design. There is always one of the FGL implementations better than the multiplexed design. However, it is interesting to note that each implementation jumps back-and-forth between distributions of which is better. On the skewed datasets, it is more important to have the larger CAMs than to have the 2 extra engines. For the less clustered datasets, it is better to have the extra engines over the CAM space. This implies that the best performance would be found with getting 8 engines routed using the CAM128. This should be possible with a couple more iterations of timing analysis and further pipelining. All FPGA designs have the same amount of memory bandwidth available to them, so increased throughut comes from three main factors. First, reducing the number of channel allocations lets us use more engines and hence see increased inter-engine parallelism. Second, because we are multiplexing more requests over the same channel, this design is able to use the available bandwidth more efficiently. This better efficiency improves the latency masking and increases throughput. Last, the fine grain locks mean that we only need to do locking in the beginning of execution while the first nodes are being inserted. The remainder of the execution is lock free.

Discussion: It should be noted that the performance benefits of the FPGA-based approaches come not from architecture-specific features, but from multithreading and efficient memory usage, which allows to utilize the available memory much better than any of the software implementations. Figure 5.8 depicts the ratio of effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and FPGA implementations while varying dataset sizes and key cardinalities. Hardware mutithreading approach allows our FPGA implementation to keep the ratio almost

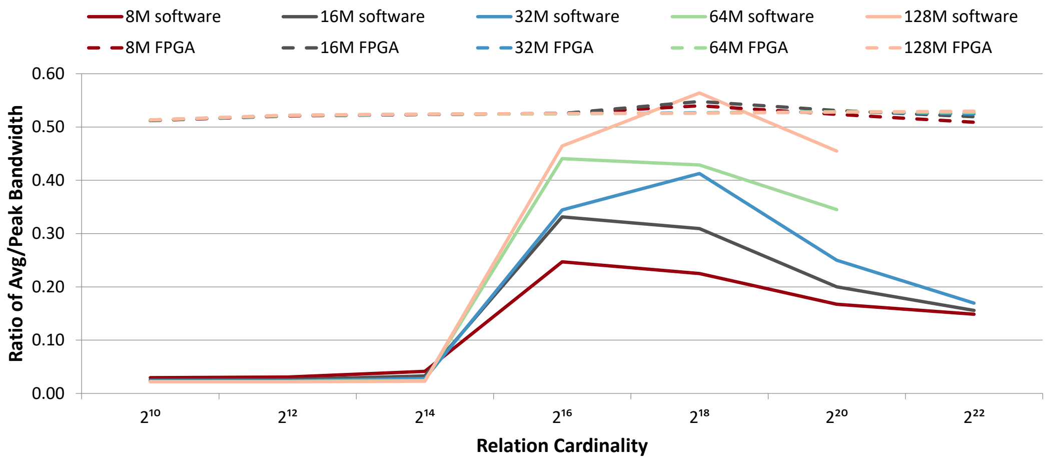


Figure 5.8: Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the FGL CAM64 design with 8 engines per FPGA for varying dataset sizes and key cardinalities.

constant, irrespective of dataset size or key cardinality. On the contrary, the ratio for the software approach varies greatly. The effective memory bandwidth of the CPU implementation tends to grow as the size of the relation increases (from 8M to 128M), whereas the FPGA-based approach is less susceptible to data size variations. For low cardinality the aggregated relation and hash table are cached and there are almost no memory accesses, hence the ratio approaches 0. The software ratio peaks at around 0.5 for cardinality 2^{18}, but drops significantly for higher key cardinalities. For very large cardinalities the FPGA implementation ratio is almost 5 times higher.

5.3.5 Effects of the Merge Operation

The Figure 5.9 shows aggregation throughput while the size of the datasets having Uniform key distribution is increased. The parallel FPGA aggregation step has almost constant throughput of about 550 MTuples/sec, even on very high cardinalities. The merge

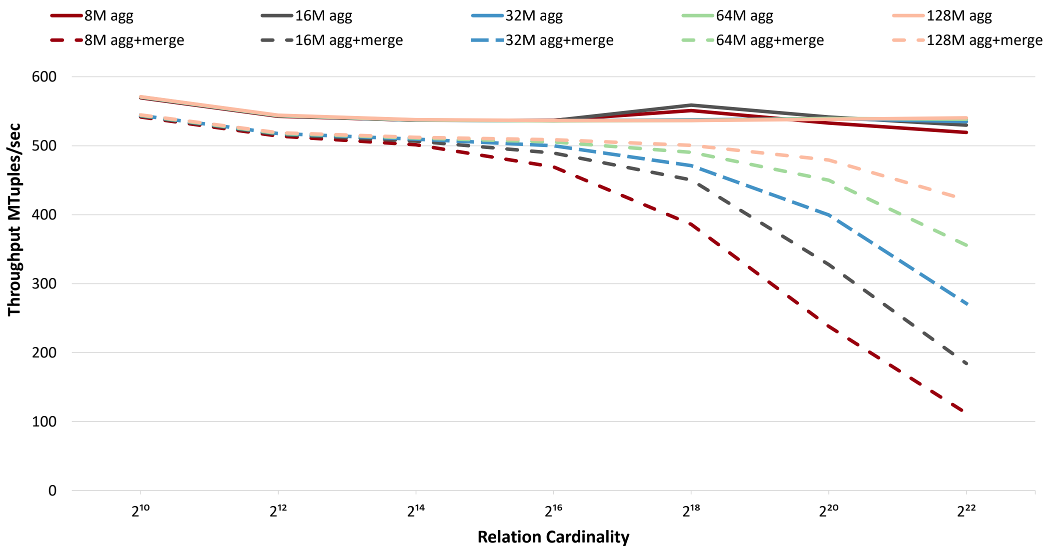


Figure 5.9: Effect of varying relation sizes on the FPGA aggregation throughput for datasets with Uniform key distribution. Solid lines represent throughput of the aggregation step (without merge operation), while dashed lines represent end-to-end (aggregation followed by the merge) throughput.

step introduces an overhead, however it comes at a fixed price. This cost depends solely on the key cardinality because aggregation reduces the initial input into a constant number of streams which should be merged. Hence as the size of the relation grows the merge step overhead gets amortized, so that the full throughput is almost constant for relations greater than 128 million tuples.

5.3.6 FPGA Area Utilization

Table 5.2 shows the resource utilization (registers, LUTs, and BRAMs used) for the three FPGA aggregation designs (multiplexed, FGL-CAM64, FGL-CAM128) as the number of engines is scaled up. The biggest drivers of resource usage in these engines are the CAMs. The CAMs are the largest components in the engines and dictate size and timing constraints. It's interesting to note that the 8 engine FGL design is only slightly

# Engines	Registers	LUTs	BRAMs
1 Original	99,597(11%)	87,194(18%)	126(17%)
3-MUX	179,641(18%)	200,175(42%)	250(34%)
1, CAM64	90,254(9%)	78,306(16%)	74(3%)
8, CAM64	200,989(21%)	199,533(42%)	144(6%)
1, CAM128	91,897(9%)	84,783(17%)	75(3%)
6, CAM128	181,049(19%)	181,049(42%)	130(6%)

Table 5.2: FPGA resource utilization for aggregation engines.

larger than the 6 engine FGL design, and that is entirely due to the smaller CAMs. Both designs are comparable to the previous design, showing that the increased complexity of the lower level locks is not too complex to implement in hardware. We were also able to save significantly in BRAM usage as well. The aggregation design uses only 42% of the available resources showing there is still room to incorporate other relational operations on the FPGA fabric.

5.4 Hash Join using CAMs

The presented hash join engine in our prior work [64] is in line with previous research efforts [17, 25, 82], where all relations were broken into "skinny" tables with small tuples (8 or 16 byte key/value pairs). The focus is on finding matching tuples between relations, but we also outline how the approach can be extended with existing research to further improve performance. That work relied upon atomic instructions provided by the Convey MX machine for synchronizing hash table updates. In this section, we describe how the build phase can be made platform agnostic using CAMs for synchronization and how the probe phase can be extended to support left-, right-, and full outer joins. We target datasets that are too large to store locally on the FPGA, and therefore all data structures are stored in global memory. The join phases are nicely split such that the every tuple in the build relation incurs write to the data structures, and every tuple in the probe relation only reads from these data structures. This separation simplifies the hash table operations for both the CPU and FPGA, compared to other hash-based algorithms (i.e. group-by aggregation in Section 5.2).

5.4.1 Build Phase Engine

Since our target datasets are too large to keep in local FPGA BRAMs, our design trades off small and fast on-chip memory for larger but 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 FPGA. Because of the FPGA's inherent parallelism, multiple threads can be activated during the same cycle while other threads are issuing memory requests and going idle.

The entire build relation along with the hash table and the linked lists are stored in main memory as shown in Figure 5.10. 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. We use a special value (0xFFF...FFF) to represent empty buckets and end of chains in the hash table.

Figure 5.10 also shows how the build engine (FPGA logic) makes requests to the main memory data structures utilizing 4 channels. In the FPGA logic, local registers are programed at runtime and hold pointers to the relation, hash table, and linked lists. They

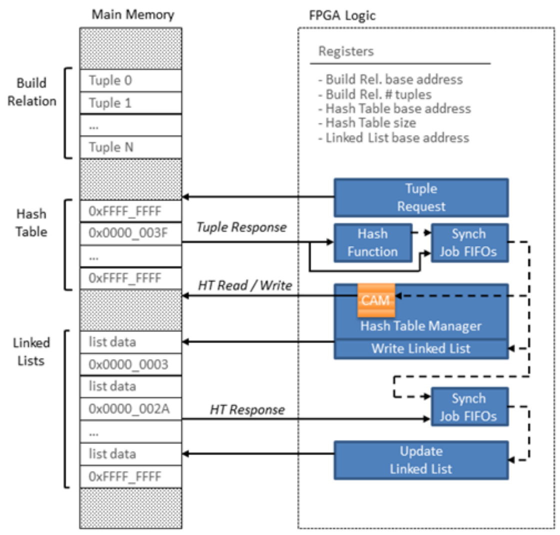


Figure 5.10: The FPGA Build Phase Engine.

also hold information about the number of tuples, the tuple sizes, and the join key position within 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 4 bytes, and it is set at runtime with a register. In Section 5.5 we show how this design can be extend to handle larger keys and tuples of arbitrary sizes. Requests are continually issued until all tuples have been processed, or the memory architecture issues a stall. Once 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 reactivated, 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 linked list bucket. Instead of issuing an atomic swap command as in the previous design, the *Hash Table Manager* component issues several requests protected by the cam to do the update. First, the thread checks the contents of the CAM to see if the hash table bucket is currently being updated. If the address is in the CAM, the thread must recycle in a FIFO and wait. If the address is not in the CAM, the thread writes the address and now has exclusive access to the hash table bucket. With the lock secured, the thread issues a read for the old bucket head pointer. Once we get a response from memory, the old bucket head pointer is added to the thread's state and the thread issues a write of the updated value. When memory responds with a write complete message, the address is flushed from the CAM, freeing the lock for other threads. Synchronization is needed here because a single FPGA engine can have hundreds of threads in flight, and issuing separate unprotected reads and writes would create race conditions. After the write is complete, the new node pointer is added to the thread's state.

As the write requests are fulfilled the thread is again reactivated, and the *Update Linked List* component updates the bucket chain pointer. If no previous nodes hashed to that location then the hash table read request will return the empty bucket value, which by design is also used to signify the end of a list chain. Otherwise, the old head pointer that was returned is used to extend the list.

The key insight to this design is to realize that all items in the relation must end up in the linked list memory space. Since we know the number of elements, we know exactly how much memory we need in the linked-list node space. Instead of dynamically allocating nodes with keys as we see them, the keys can have a fixed location in memory and it is the next pointer slot that changes based on the current state of the hash table. We are effectively building the linked list around the nodes as they sit in memory. If there are n

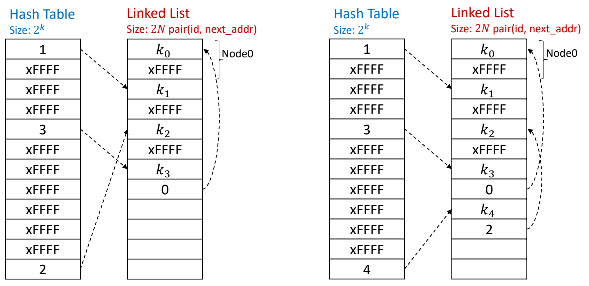


Figure 5.11: Inserting an element into the Hash Table. Dashed lines represent logical connections. Assume $hash(k_4) = k - 1$. Address 2 is read from the $hash\_table[k - 1]$ and 4 is written in its place. k_4 is written to the 8th slot, and the old address 2 is written as it's next pointer in the 9th slot.

elements in the relation, the linked-list structure will be $2 * n$ (one word for the key, and one word for the next pointer). Every $i'th$ element will always end up in the $2 * i$ position and its next pointer will always be at $2 * i + 1$. The dynamic nature of the list building comes from swapping the pointers out of the hash table to write into the linked-list table. Figure 5.11 gives a visual example.

5.4.2 Probe Phase Engine

The probe engine also assumes that all data structures are stored in main memory. Like the build engine it uses memory masking to cope with high memory latency and maintain peak performance. Because no data is stored locally for either engine, the same FPGA used during the build phase can be reprogrammed with the probe engine (useful for smaller FPGA). Larger FPGAs can hold both engines and switch state depending on the required computation. The prior design[64] of the probe engine only handled inner-join queries. If key_a and key_b are in both the build and probe tables, then the joined tuple was emitted. In this current design, we support the full array of join variants: inner, left,

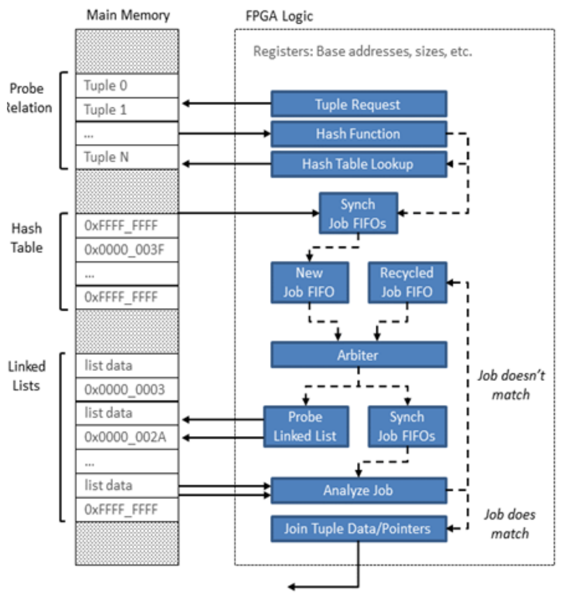


Figure 5.12: The FPGA Probe Phase Engine.

right, and full outer join. These can be toggled on and off in the data path to configure the join operation desired. Left outer join has no performance overhead as the work to identify the null match is already done while answering an inner join. For the right outer join, the design incurs one final scan of the build table to emit the tuples in the build table that were not matched during the query.

Figure 5.12 shows how the probe engine makes requests to the data structures in main memory (using 5 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 synchronizing operations. The thread only looks up the proper head pointer by probing hashed key into the table. The special 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 for inner join or emitted with null for left join. 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. We devote two channels to this component because it issues the bulk of read requests, and its performance is vital to the engine's throughput.

After the proper node is retrieved from memory the *Analyze Job* component determines if there was a match. Matching tuples are sent to the *Join Tuple* component. If the query is handling a right outer join, then the engine will issue a write to the hash table node to mark it as matched. This enables the final scan of right outer join to identify unmatched nodes in the build relation. If a node is the last in the bucket chain then its thread is dropped from the datapath for inner join and emitted with null for left join. 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.

5.4.3 Possible Optimizations

In practice 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 waiting threads do for CAM locks to free. 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, however since the design is fully pipelined [119] this would not affect 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 [120, 114, 65]. In the special case where an entire tuple fits in one memory word the probe engine presented in this section 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. The case when a tuple length exceeds memory word size, thus requiring multiple-cycle lookup is covered further in Section 5.5.1. 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.

Partitioning is a common optimization applied to multi-core hash join, which can eliminate costly thread synchronization by keeping the partitioned tuples cache resident [117]. However, the multithreaded approach used on the FPGA requires hundreds of outstanding requests, and would still enforce some form of synchronization to avoid race conditions. Moreover hardware design does not rely on caching since in general case hash join has neither spatial nor temporal locality, thus partitioning will not be able to decrease the total number of jobs.

5.4.4 Experimental Results

In this section we show how our approach is implemented on a Convey MX architecture. We explain how the engines can be duplicated to increase parallelism and better utilize the available memory bandwidth. FPGA synthesis is known to be a time intensive process. The designs presented here are general enough to handle different join queries **without** needing to re-synthesize the FPGA logic. Our FPGA implementation is compared, in terms of overall throughput, to the best multi-core approaches we could find [17]. We attempt to match the FPGA's and CPU's memory bandwidth (38.4 GB/s for the FPGA vs 51.2 GB/s for the CPU) because hash join is a memory bounded problem. Scalability and area utilization results are also presented.

FPGA & Software Implementations

The hash join approach presented in this section is implemented using the Convey HC-2EX platform (Section 2.5), but the proposed methods are platform independent. The *Probe Engines* are easily ported between boards because the FIFOs (Xilinx IP Cores) are the only component specific to the FPGA and could be easily replaced with generic FIFOs. The *Build Engines* are also portable because they utilize CAMs for synchronization instead of the atomic instructions of the previous design.

Peak FPGA performance depends on the number of concurrent engines, and their clock frequency. The number of engines is limited by the memory bandwidth. The Convey HC-2EX has 16 memory channels per FPGA, which run at 150MHz. The build engine described in Section 5.4.1 requires 4 channels, and therefore each FPGA can hold 4 engines. Assuming no stalls the peak throughput for the build phase is 600 MTuples/s (4 x 150) per FPGA. Similarly, the probe engine mentioned in section 5.4.2 uses 5 memory channels, and therefore each FPGA can hold 3 engines. The probe phase has a peak throughput of 450 MTuples/s per FPGA.

The state-of-the-art multi-core hash join approach [17] we compare against has 2 types of join algorithms: a hardware-oblivious non-partitioned joins and a hardware-conscious algorithms, which performs preliminary partitioning of their input. Both implementations perform the traditional hash join with build and probe phases, however they differ in the way they are utilizing multi-core CPU architecture. The non-partitioned 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, thus 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 architecture.

Dataset Description

Our experimental evaluation uses four datasets. Within each dataset we have a collection of build and probe relation pairs ranging in size from 2^{20} to 2^{30} elements. Each dataset uses the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query engines [26]. 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.

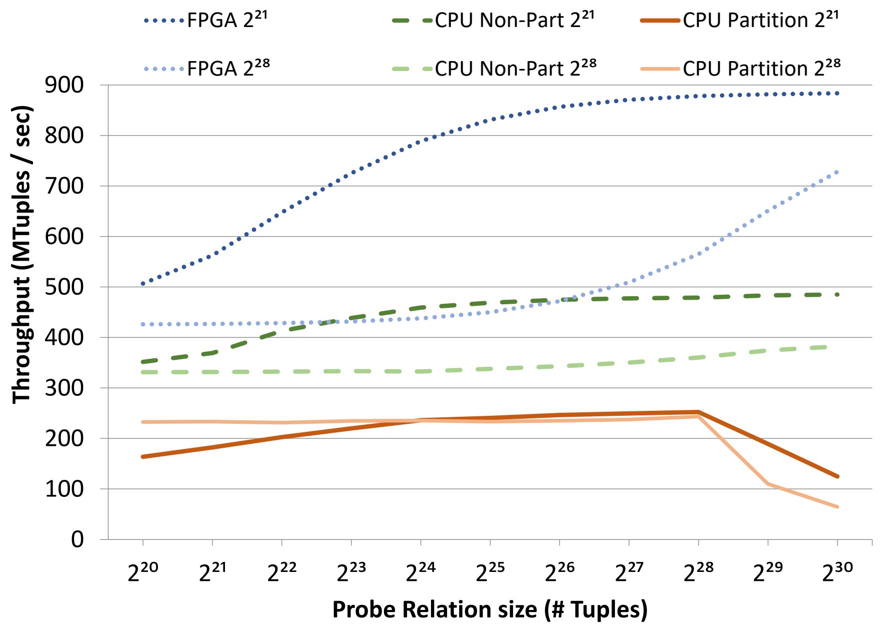
The first dataset, termed *Unique*, uses incrementally increasing keys which are randomly shuffled. It represents the case when the build relation has no duplicates, thus keys in the hash table are uniformly distributed with exactly one key per bucket (assuming simple modulo hashing). The next dataset (*Random*) uses random data drawn uniformly from *uint32* value 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 by Gray et al. [59]. 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.

Throughput evaluation

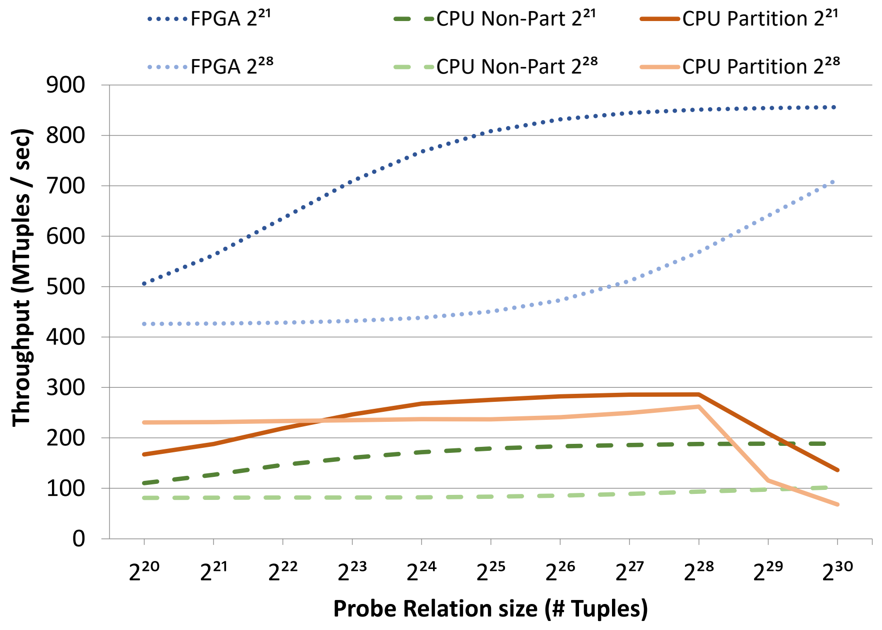
We report the multi-core results for both partitioning-based and non-partitioned algorithms. Results are obtained with a single Intel Xeon E5-2643 CPU, running on full load with 8 hardware threads. However because of the memory-bounded nature of hash join we use two FPGAs to offset the CPUs bandwidth 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). Obviously, given of the parallel nature of hash join, the CPU and FPGA performance could easily be improved by adding more hardware resources.

Figures 5.13 and 5.14 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 5.4.4. The FPGA performance shows two plateaus for the *Unique*, *Random* and *Zipf_0.5* data distributions on Figures 5.13a, 5.13b and 5.14a.

The FPGA sustains a throughput of 820-850 MTuples/s when the probe phase dominates the computation (that is, when the size of the probe relation is much larger

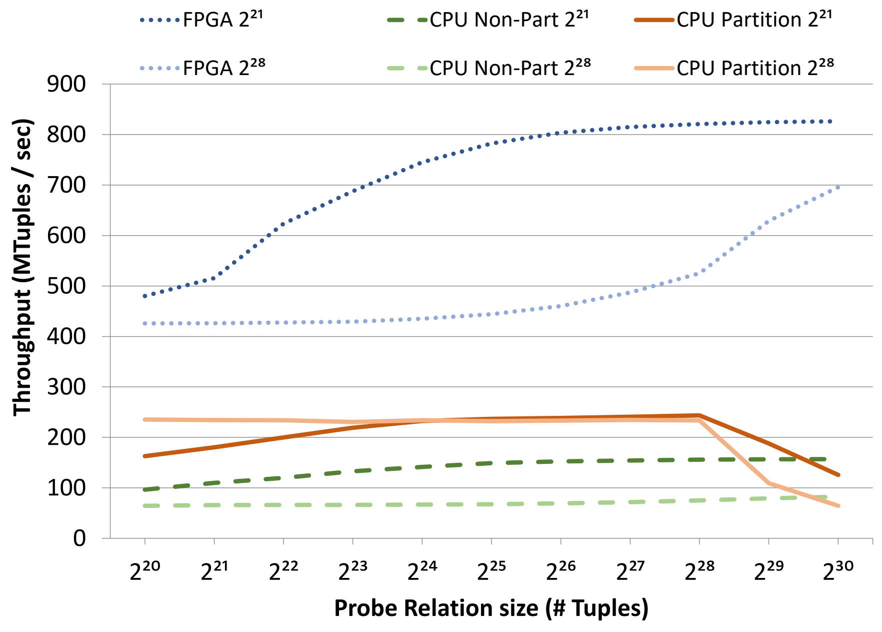


(a) *Unique* dataset

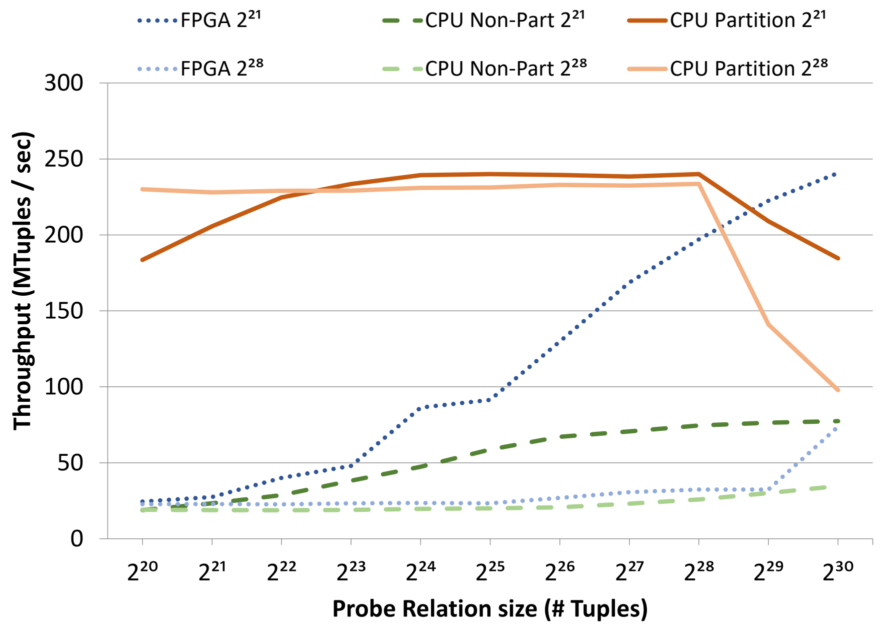


(b) *Random* dataset

Figure 5.13: Unique and Random throughput as the build relation size is increased.



(a) *Zipf_0.5* dataset



(b) *Zipf_1.0* dataset

Figure 5.14: Zipf throughput as the build relation size is increased.

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 6 engines on 2 FPGAs. When the build phase dominates the computation, synchronization restrict FPGA throughput to about 425 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 500 MTuples/s when both relations are of the same size. For the highly skewed dataset, *Zipf_1.0*, (shown in Figure 5.14b) 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, thus greatly affecting the throughput.

When compared to the results in the previous work, the throughput numbers and trends are nearly identical. Only the extremely skewed dataset *Zipf_1.0* shows a performance hit from synchronizing in the CAM. For non-skewed datasets, synchronizing with the CAM provides platform independent performance.

The CPU results are consistent with the experiments presented in [17]. 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. We have seen the same pattern in the FPGA case, when the throughput of the build phase is lower than the probe phase. The non-partitioned algorithm always behaves 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 drops. For the highly 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.

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.

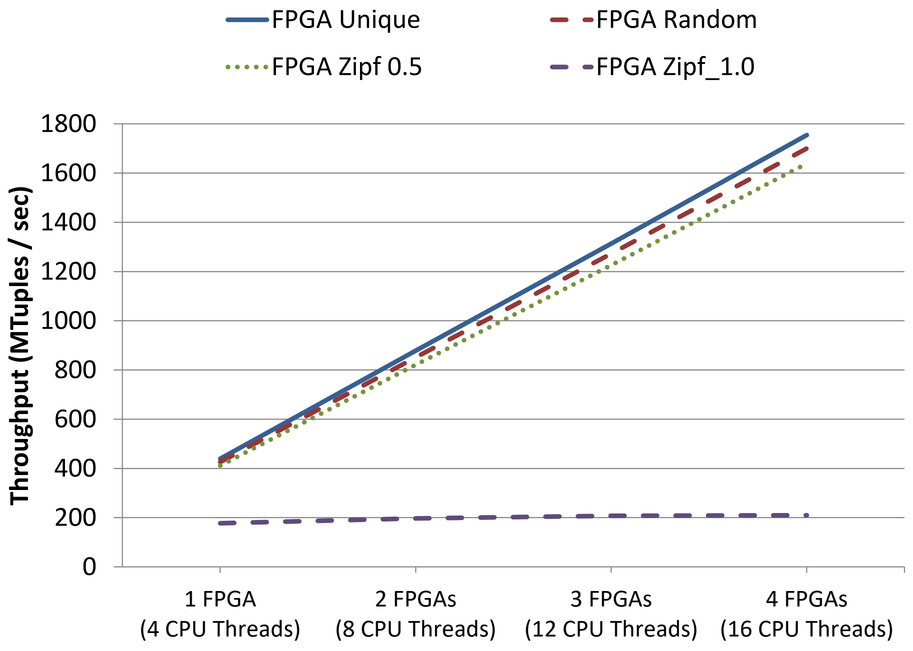
Figures 5.15a,5.16a and 5.17a show the results when the probe phase dominates the computation. The FPGA scales linearly on datasets *Unique*, *Random* and *Zipf_0.5* (Figure 5.15a). However, for the *Zipf_1.0* dataset, the performance does not scale because of the high skew. Each probe job searches through 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 ability of a new jobs to enter 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 5.16a). 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 5.17a).

The FPGA scales at a lower rate when the build and probe relation are of the same size (Figure 5.15b), since the throughput of the build phase scales at a lower rate and is a larger percentage of the overall runtime. However, this is significantly better scaling than our prior results, where the scaling of the build started to flatline after 2 FPGAs. The CAMs provide better scaling over more FPGAs than the atomic instructions. The slope of the scale graph is almost comparable to the CPU implementations (shown on Figures 5.16b and 5.17b) again with the exception of highly skewed data.

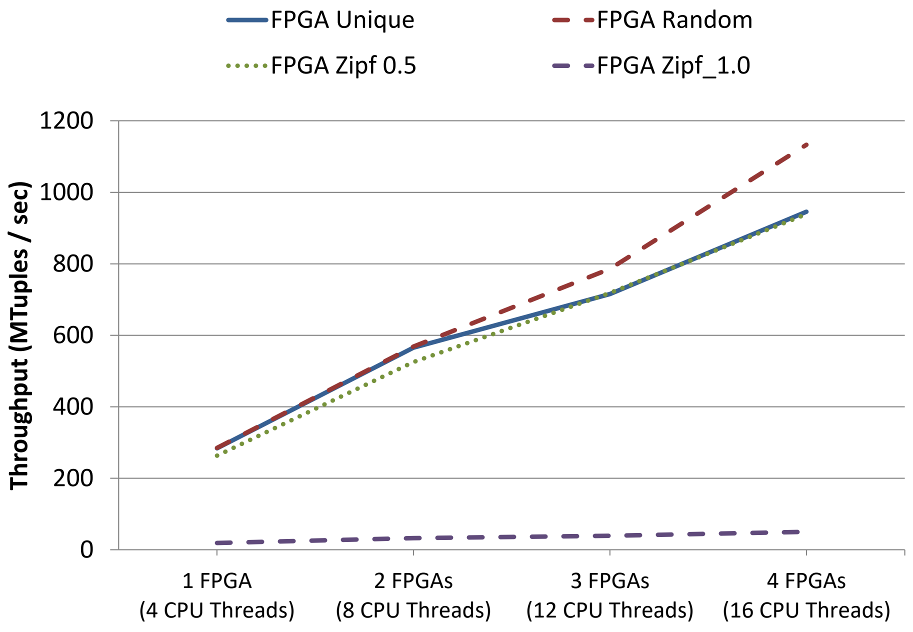
5.5 Hash Join with Arbitrary Size Tuples

The presented hash join engine in our prior work [64] is in line with previous research efforts [17, 25, 82], where all relations were broken into "skinny" tables with small tuples (8 or 16 byte key/value pairs). While this approach is valid for in-memory column-oriented databases, it is not practical for traditional DBMSs with row-major storage format.

In this section we show how our FPGA engines can be extended to support both wider tuples, and larger key length. This is done by modifying the initial memory requests to support variable tuple lengths, and increasing the FPGA's internal datapath to support wider keys and values. The updated engines are compared with modified versions of the partitioned and non-partitioned software approaches from Section 5.4. The performance of the modified join implementation is then evaluated using the TPC-H benchmark.

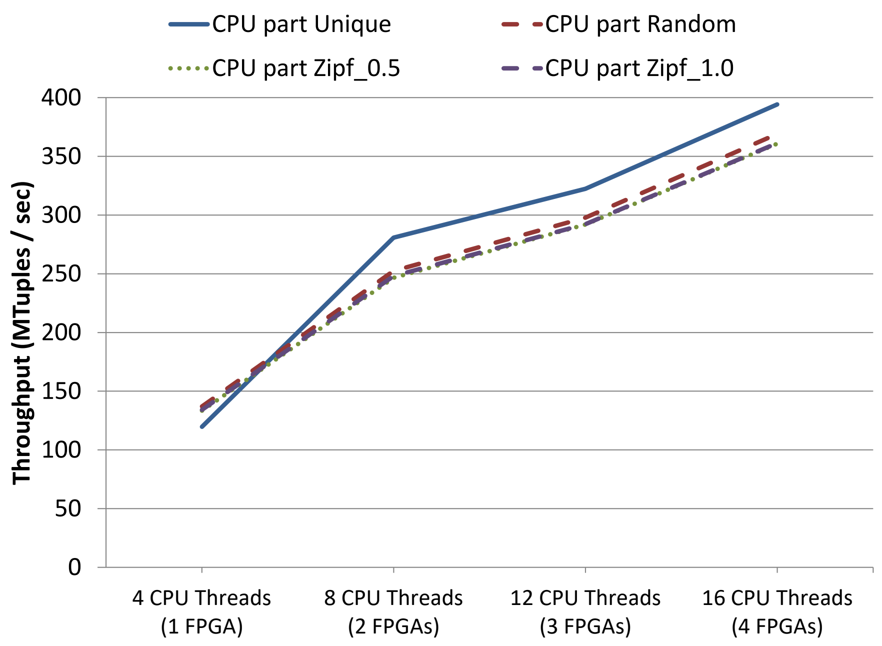


(a) Build Relation has 2^{21}, Probe has 2^{28} tuples

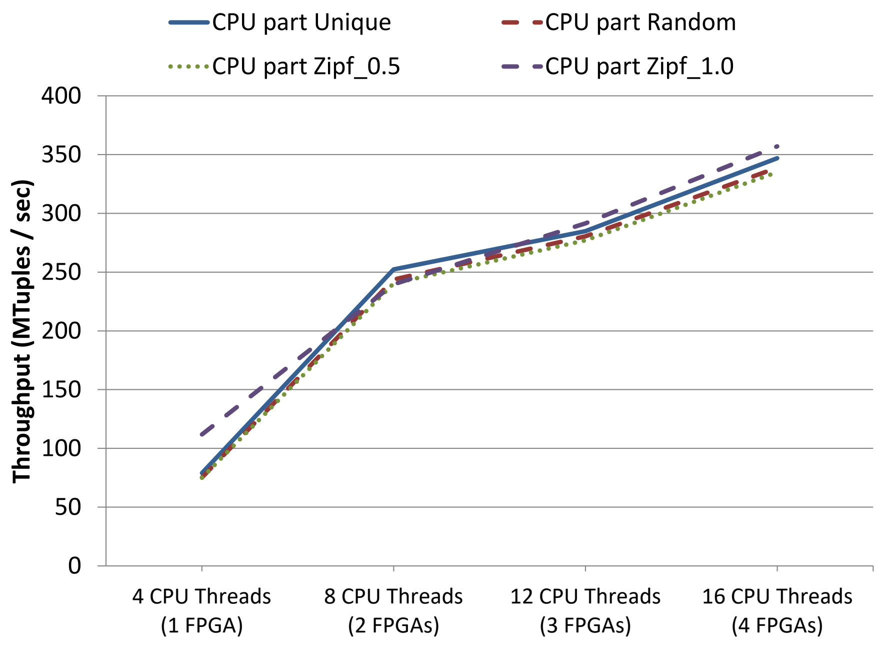


(b) Build and Probe Relations both have 2^{28} tuples

Figure 5.15: FPGA Throughput comparison as the bandwidth and number of threads are increased.

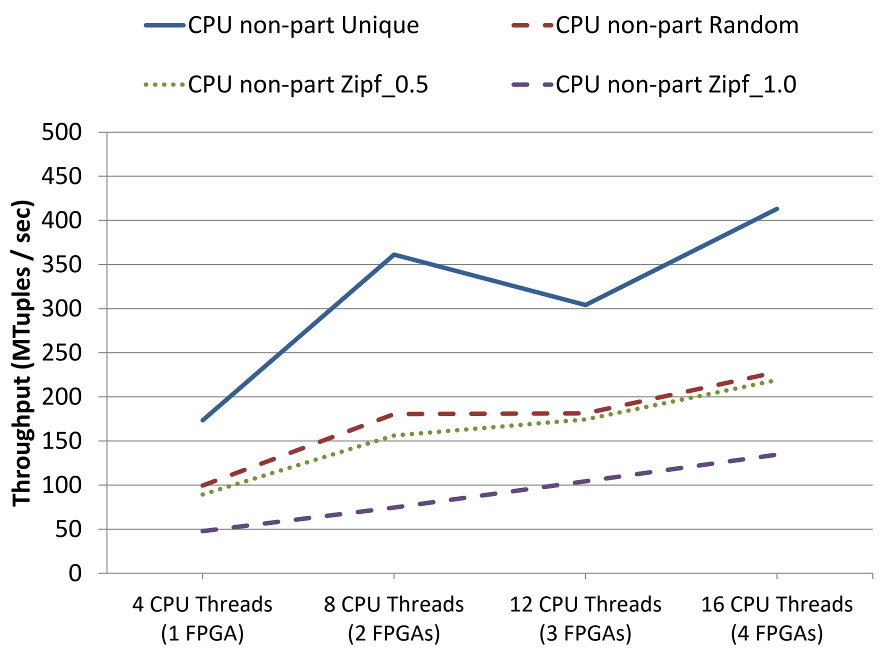


(a) Build Relation has 2^{21}, Probe has 2^{28} tuples

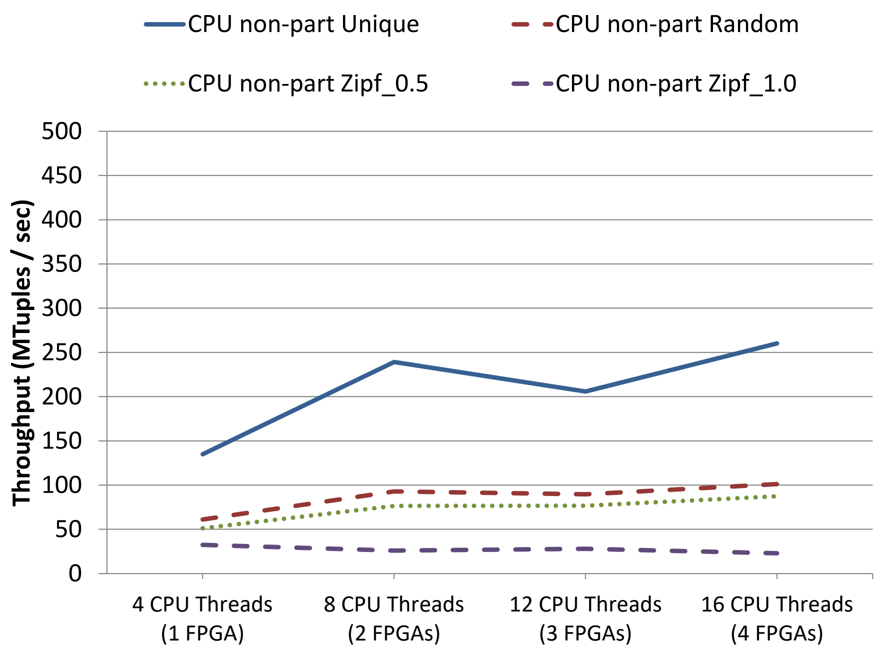


(b) Build and Probe Relations both have 2^{28} tuples

Figure 5.16: Partitioned CPU throughput comparison as the bandwidth and number of threads are increased.



(a) Build Relation has 2^{21}, Probe has 2^{28} tuples



(b) Build and Probe Relations both have 2^{28} tuples

Figure 5.17: Non-partitioned CPU throughput comparison as the bandwidth and number of threads are increased.

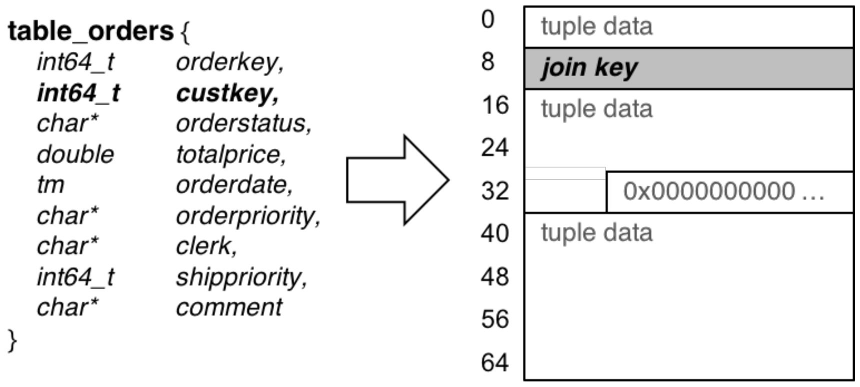


Figure 5.18: Wide tuples are stored as contiguous memory blocks, but the join operation only needs the key value. Its offset can be computed at runtime.

5.5.1 Supporting Variable Tuple Lengths

The build engine and probe engine datapaths are logically similar to those presented in Sections 5.4.1 and 5.4.2. Only the *Tuple Request* component is modified allowing us to issue non-sequential memory requests for the join keys. This change is done to correctly handle the scenario, when tuples occupy multiple memory locations. Consider a sample tuple from the TPC-H Orders table, shown in Figure 5.18. The size of this tuple is 72-bytes. Assume that we implement the join operation ($table\_orders \bowtie_{custkey} table\_customer$). In the build phase only the join key (*custkey* which is 8-byte long) is kept inside the hash table, and in the probe phase only the join key is needed to determine a match. Therefore, the join operation only needs to consider 8 bytes from each tuple. However, once a match is verified the full tuple data will have to be streamed into the FPGA and merged. Since only the join key is used to identify a match we can reuse the key/value pair model from the previous section. The value is a memory pointer to the actual tuple. To compute the stride length, runtime programmable registers are used to hold the tuple's width and the field's offset value for a given relation. In our example the field offset is 8-bytes. This allows the FPGA to handle various tuple sizes without needing to be re-synthesized.

Using only the join key to identify matches presents an additional challenge during the probe phase. Since it is not known a priori if a tuple has a match, unnecessarily streaming a large tuple would hinder performance. This issue exacerbates as the tuple width increases. In addition, after synthesis the FPGA datapath has a fixed maximum width. Any tuple larger than the datapath will require multiple cycles to transfer between components. A fixed width key/value pair allows the FPGA designer to optimize the datapath such that all threads flow through the pipeline stages in a single cycle. One approach is to cache the tuple locally (i.e. on-chip BRAMs). However, careful design is needed since jobs are not guaranteed to complete in order (due to variable length linked lists) and the cache could become fragmented. In addition the cache size must be large enough to store all running jobs (in the thousands for high-bandwidth architectures).

Lastly we need to address the issue of a join key that spans multiple words in memory. In Figure 5.18 the join key is nicely placed inside a single word of memory and thus it requires a single memory request. This is a common case when using aligned C structs, which pad short tuple fields with zeros to store them in memory. However some databases (especially in-memory DBMSs) could eliminate this padding to achieve better data compression. Such optimization could cause the join key to be split across two words of memory. The modified *Tuple Request* component handles this case by issuing multiple memory requests. A reorder block is used to realign the join key before sending it out of the *Tuple Request* component.

5.5.2 Increasing the Key/Value Size

As 8-byte memory words are becoming the standard we now describe how the build engine and probe engine are expanded to support the larger key & value sizes. Increasing the key and value sizes requires the FPGA's datapath to be widened. From a design perspective the modification boils down to simple change of variables type from *uint*32*_t* to *uint*64*_t*. Throughout the HDL specification wire sizes are increased from 32-bits to 64-bits as well. Increasing the key and value sizes also increases the memory bandwidth demands, requiring design changes since the Convey bandwidth is fixed. The memory channels could be reallocated, but it will reduce the number of engines per FPGA. The FPGA engines could also duplex extra requests through the same physical channels, but it will effectively double the execution time.

With this in mind our implementation opted to keep the number of engines per FPGA the same (4 engines per FPGA), and increased the number of memory requests over certain channels. Each tuple for the wider datapath requires one extra memory request during the build phase. The *Write Linked List* component now breaks the key/value pair into two distinct memory writes. Because this is not an atomic operation we duplex the writes together through the same channel.

For the probe engine, earlier experiments (Section 5.4.4) showed that the architecture can achieve close to peak performance. In the updated design a minimum of two extra memory accesses are required for the following reasons. First, in the initial design the build phase key/value pairs fit in a single word of memory, but in the updated design they occupy two words of memory. Second, the updated design outputs two 8-byte values instead of two 4-byte values merged into a single 8-byte word. Because the probe phase requires two extra requests, two physical channels could be duplexed; one from the *Probe Linked List* component and another from the *Join Tuple* component. However, duplexing any memory channel doubles the number of memory requests, and will cut the throughput performance in half. A better option is to reallocate the memory channels per each FPGA, so that the updated design uses 7 channels per engine. As a result, only two updated probe engines will fit in the 16 available memory channels for the Convey FPGAs, whereas the previous design (Section 5.4.4) could fit three probe engines with the same channel budget. Therefore, the throughput performance is decreased only by one third.

5.5.3 Adjustments to software approaches

Although the memory is byte-addressable for both hardware and software join implementations, CPUs incur special kind of memory access reading in the whole L1 cache line (64 bytes on our architecture) and bringing it into the cache to take advantage of the locality. However, increasing the tuple length cannot utilize any of the locality properties, neither temporal (tuples are read only once both during build and probe phases), nor spatial (size of the tuple typically is greater than a single cache line). Thus the extensive caching, which could not be disabled, only puts additional pressure on memory bandwidth, effectively deceasing the processing throughput.

Because such penalty should be paid each time the join key is accessed we choose to implement a projection step and materialize the intermediate result in a form of key/value pairs, with the same format used in Section 5.4.4. Projection step essentially allows us to amortize join key access overhead by allocating additional memory to store projected relations. Our initial experiments showed that executing projection as a separate step, without incorporating it into build or probe phase, yields better performance results due to better locality.

5.5.4 Experimental Results

All results obtained in this subsection were collected by running tests on the Convey HC-2ex platform. In Section 5.4.4 we showed that FPGA performance is predictable for most relations; the exception being highly skewed datasets. However these experiments used “skinny” key/value relations. In this subsection we show that the same FPGA predictability holds for larger real world datasets. Results show that the join performance on two tables (Orders and Customer) from the TPC-H benchmark suite with varying scale factors.

TPC-H Dataset

For our wide tuple experiments we used the TPC-H benchmark from the Transaction Processing Performance Council. It is a decision support benchmark which is traditionally used to compare analytical query performance on commercial database systems. The schema consists of 8 different relations, and 22 unique queries. A scale factor (SF) is used to control the datasets’ size, and allows it to generate relations between 1GB and 100TBs. While the absolute size of the relations might vary, their relative sizes are fixed. For example the size of the Orders table is 10 times bigger than the size of Customer relation, regardless of the SF.

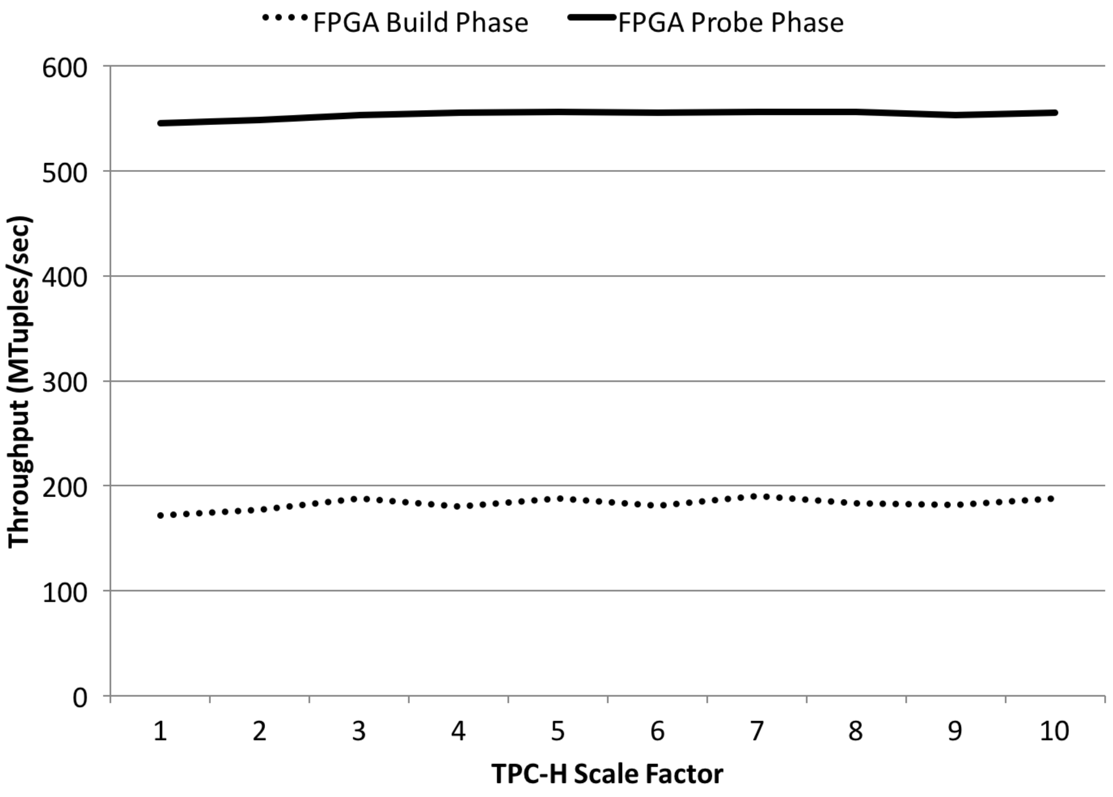


Figure 5.19: FPGA throughput results for the build and probe phases as the TPC-H scale factor is increased.

As discussed in Section 5.4.3 the FPGA can offer significant speed up for selection and projection. However, our intent is to study only the join performance, and to remove any variability in our tests we ignore all other operations. Our experiments perform the $Orders \bowtie_{custkey} Customer$ operation, which is used in queries $Q3$, $Q5$, $Q7$, $Q8$, $Q10$, $Q13$, and $Q18$. For all tests the smaller *Customer* table is used as the build relation, and the *Orders* table is used as the probe relation. We use the *qGen* utility to generate 10 datasets with the scale factor varying between 1 and 10.

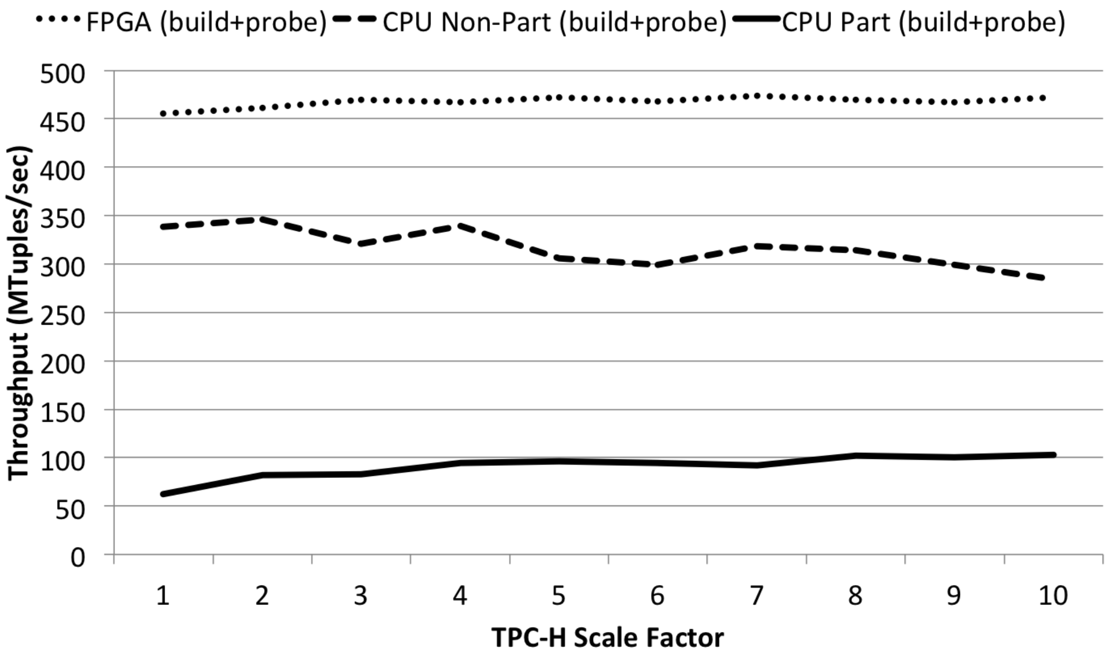


Figure 5.20: FPGA and CPU total throughput performance as the TPC-H scale factor is increased. Results include both the build and probe phase execution times. The partitioned CPU performance also includes the preprocessing time.

Throughput Results

As in Section 5.4.4, we compare the throughput results between one CPU to two FPGAs, in effort to match the memory bandwidth. Figure 5.19 shows the FPGA throughput results for the build phase and probe phase separately. We see that the build phase is still bottlenecked by synchronization as it was in Section 5.4.4. Peak performance for 8 engines at 150MHz and one duplexed memory channel per FPGA is 600 MTuples/sec, while sustained performance is around 200 MTuples/sec.

The probe phase achieves near peak performance (about 550 MTuples/sec), again in line with the experiments from Section 5.4.4. Because of the increased key/value pair size only 4 engines could fit on two FPGAs, thus the combined peak theoretical throughput is 600 MTpules/sec.

In Figure 5.20 we show the end-to-end throughput for all three approaches: FPGA, CPU Partitioned, and CPU Non-Partitioned. In the TPC-H benchmark the probe table is 10x larger than the build table, and therefore dominates the computation time. This does not have a big impact on both CPU approaches, but for the FPGA it skews the performance toward the probe phase's throughput. The FPGA achieves throughput results between 450 MTuples/sec and 475 MTuples/sec depending on the scale factor. Non-partitioned CPU algorithm achieves better throughput (between 300 and 350 MTuples/sec) in comparison to partitioning-based one (between 50 and 100 MTuples/sec). In the Customer relation each join key is encountered exactly once, and therefore its key distribution is identical to the *Unique* dataset from Section 5.4.4. As explained earlier, this is why the non-partitioned approach performs better than it's partitioning-based counterpart.

5.6 Conclusions

In this paper we implemented and evaluated two relational database operations, join and aggregation, using hardware multithreading techniques on FPGA hardware accelerators. The data structures are kept in global memory, which increases the access latency compared to on-chip BRAMs, but allows us to tackle much larger problem sizes. Multi-threading allows the FPGA to mask the longer latency by issuing thousands of job threads.

In Section 5.2 we used the same multithreading techniques to implement a group-by aggregation function on the FPGA. Aggregation is a more complex operation because jobs can either update an existing node, or create a new node. Compared to hash join where every job in the build relation creates a new node, and every job in the probe relation only reads the data structures. We evaluate the FPGA against five software approaches (both partitioned, and non-partitioned) over five different datasets. Experiments show a sharp decline in performance for the software approaches as the cardinality increases. The FPGA's throughput is unaffected by the benchmark's cardinality, and can sustain between 300 and 600 MTuples/sec depending on the key distribution.

In Section 5.4 we presented a hash join design for key/value store databases. Throughput experiments over three datasets, ranging from uniform to slightly skewed, showed that the FPGA can outperform the best currently available software implementations by 2x to 3.4x. However, on extremely skewed datasets the FPGA's performance suffered compared to software approaches.

In Section 5.5 we extended our hash join design to support wider tuples, and larger key sizes. We proved that our design is not only limited to column-major storage formats, but can offer performance improvements in traditional row-based DBMSes. We tested out implementation on join queries from the TPC-H benchmark suite. Throughput results dropped compared to the key/value design for both the FPGA and software. However, the FPGA outperformed the non-partitioned CPU results by 1.3x to 1.5x. The FPGA outperformed the partitioned CPU results by over 4x.

Chapter 6

Better Building Blocks

One of the key drivers of the performance and efficiency of the latency masking multithreaded model is the efficient use of memory requests. A perfect example of this can be seen in the Breadth First Search algorithm (Chapter 4). It would be possible to report an extremely high throughput on a graph with one node by simply requesting that one node over and over continuously. This simple construction lets us guarantee a request every cycle, and fully mask latency. However, it should be obvious to the reader that even though the throughput would look high, the implementation provides little utility. This implies that we want to do requests every cycle, but for perfect utility we would like each request to provide progress. There is a notion of efficiency of requests such that some requests accomplish work and other requests accomplish nothing.

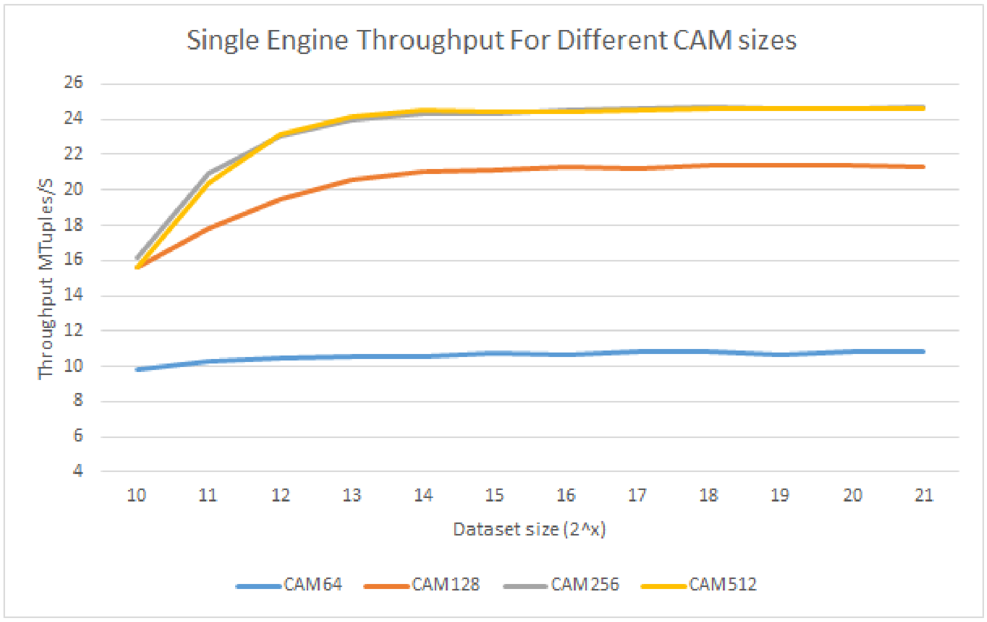
It was under that insight that we demonstrated in Chapter 4 how to use a CAM to reduce redundant requests in the top down algorithm. However, one of the limiting details of that design is how large of a CAM we can make reach timing. More CAM space means that we can have more threads operating concurrently. And more CAM IDs in flight means more jobs that might be able to be merged.

In the previous chapters, we were limited to CAM depths of 128 or in some cases 64. In this Chapter, we will give another example from the aggregation algorithm to show we are not perfectly saturating memory because of the CAM size. We then explore what is the bottle neck in the CAM and how we can develop a batter CAM that's less limiting.

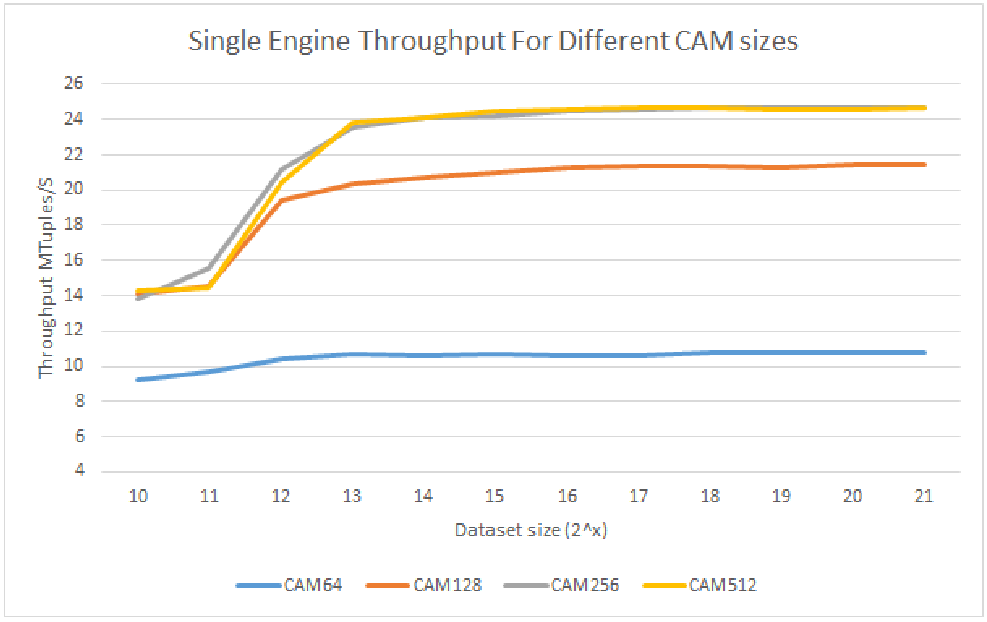
6.1 Improving CAM Performance

To validate the importance of depth to performance, we focused on how varying the CAM size influences throughput of the aggregation engine. Figures 6.1a and 6.1b show the single engine throughput for varying CAM sizes. There are two main take-aways. CAM size does affect performance, but there are diminishing returns. For this application, that limit is the memory response time. The Convey-WX runtime simulates memory with a fixed 300 cycle response time. That is apparent in the graphs since the throughput increases up until CAM size 256. The throughput between CAM 256 and CAM 512 is indistinguishable. It is also noteworthy that the performance reduction for really small CAMs cancels out the steady state performance gains typically seen in multithreaded applications.

The other sizes show the CAM as lock design works and still shows the typical multithreaded performance ramp. The distribution differences between the datasets seems to have negligible effects on performance. However, A CAM size of 128 is too small. The results in our previous work would see a 10-15% performance boost simply by increasing



(a) Throughput varying CAM size, random dataset



(b) Throughput varying CAM size, zipf_0.5 dataset

the CAM size. It is worthwhile to explore different CAM architectures to make it easier to meet timing at larger sizes.

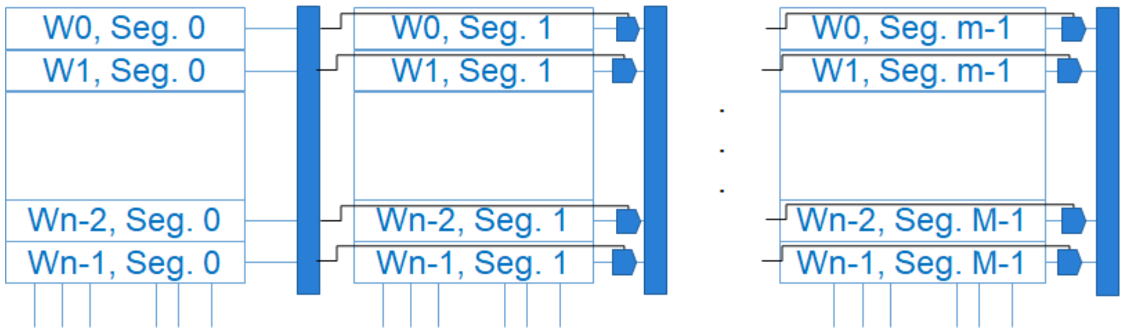


Figure 6.2: Pipelined CAM layout

To ease timing constraints, we took some insights from the ASIC CAM research area and implemented a pipelined CAM with power gating at each stage. By breaking the CAM search logic into smaller stages, the timing reduces to the timing of the width of each stage. At 16 bit wide stages, CAMS as deep as 2K were able to meet 200 MHz timing. Gating the search logic at each pipeline stage helps the design reduce dynamic power usage. Most search words will fail early in the word. The gating logic only activates a search stage if the previous stage matched. By the end, only a few rows will likely be searching. Figure 6.2 shows a visual breakdown of the layout, and Verilog code for the implementation is available on bitbucket[108].

6.2 Cuckoo Hashing

Another primitive common in the previous chapters is the use of a hash table. A hash table is needed for fast lookups of keys for quick updates. However, not all hash table models are equal. The previous implementations used a traditional hash table that used a separate chaining strategy for collisions. If collision rate is small, the linked lists won't cost too much overhead in searching the lists. But if the skew of the dataset increases, the length of the linked-lists can become a significant bottleneck in the performance. All of our threads will continually hopping down the same lists over and over again to find the slot where they need to do work. In terms of multithreading efficiency, we can think of these as inefficient, redundant requests.

With that in mind, we started looking at other hash table models to alleviate this skewed data issue. And in particular, a model that provides constant time access. Constant time access means we won't pay repeatedly growing inefficient requests. We may not eliminate all inefficient requests, but we putting a constant bound on it should really improve the overall efficiency.

6.2.1 Related Work

Traditional hash tables and their assorted conflict resolution techniques (separate chaining, linear probing, etc.) provide an *expected O(1)* lookup cost that can degenerate to a worst case *O(n)* search through all stored elements. Improving the worst case behavior and guaranteeing *O(1)* lookups is the motivation behind the Cuckoo Hash [107]. Initially, the cuckoo hash used two tables and two hash functions chosen from a universal hash family. A universal hash function has the property that most data will be widely and evenly scattered even if input elements are close in numerical order. A function from this family will generally provide minor collisions with another function from within the family. The cuckoo hash takes advantage of this property to spread the input items over the underlying tables as evenly as possible.

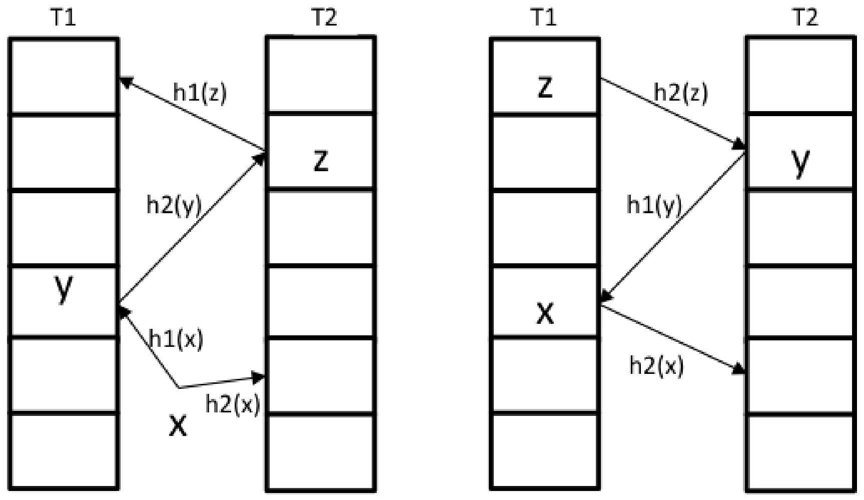


Figure 6.3: Inserting item x into a cuckoo hash table: the insertion into T1 evicts y, which in turn evicts z before the insertion is completed. Arrows denote alternate hash locations for each element.

Example: Consider inserting item X into the hash table Figure 6.3. First, X is hashed using $h1()$ to its location in table T1. Checking this location leads to two outcomes - either the slot is empty, in which case X is stored here and the insert is finished, or, the slot is occupied, say by Y. In the occupied case, Y is evicted and rehashed it using $h2()$ to find its location in T2. Storing Y in T2 may result in "kicking out" an item. This continues until all keys are able to "find a nest".

This construction guarantees that there are only two locations that an element can exist in the table, and thus O(1) lookups. However, the likelihood that any given insert will succeed is only guaranteed while the table utilization is under 50%. Later work showed that increasing the number of hash functions (d-ary cuckoo hashing) as well as increasing the associativity of each location could significantly increase the max capacity utilization [55, 113]. By using four functions with one bucket per cell, 97% utilization is achievable. Using a small stash (in the form of a CAM) could allow for schemes that only require one memory move per insert [83]. A well configured cuckoo hash can provide good cache properties and outperform linear probing (often chosen over other schemes for its cache behavior) [113].

The cuckoo hash idea was first used on an FPGA [121] as a component for a Network Intrusion Detection algorithm; however, it was specifically built for the problem and was not a general purpose. Recently, a more generic hash implementation was demonstrated that charted its utilization up to 3 tables [78]. Cuckoo hashing was also used by [52] to build a low latency table for high frequency algorithmic stock trading. Bloom filters were used in [41] to build a low latency key-value store which have analogous properties to cuckoo hashes. More recently, Cuckoo hashes were used in the Caribou system[77] to build an in memory key-value story for a near-data processing in database engine imlplemented with FPGAs. The cuckoo hash was used to improve the throughput of random accesses to memory.

6.2.2 Implementation

Guided by the results of the previous research, we implemented a configurable software model in C++ so we can test the utility of a cuckoo hash for the group-by aggregation algorithm. One benefit of a software model is the flexibility in testing many configurations. This model was designed to take the number of underlying cuckoo tables as a parameter and randomly construct universal hash functions for each table. The interface to the model can be seen in Figure 6.4

```
struct hashItem
{
    unsigned key;
    unsigned address;
    unsigned count;
     ...
};

class CuckooHash
{
private :

    std :: vector<UniversalHashFunction> hashFunctions;
    std :: vector< std :: vector< std :: vector<hashItem> > > cuckooTables;
    std :: vector<hashItem> stash;
    std :: mt19937_64 rand_engine;

    //stats  collection
    unsigned timesToStash = 0;
    unsigned memoryRequestCount = 0;

public :
    CuckooHash();
    CuckooHash(unsigned numFunctions, unsigned tableSize, unsigned numBuckets, unsigned stashSize);

    void  insert (hashItem hItem, std :: function<void(hashItem &h)> updateIfContains);
    void  remove(hashItem hItem);
     ...
};
```

Figure 6.4: Cuckoo Hash Software Model interface

Each insert into the table can cause several memory accesses as there is an unknown number of cuckoo bounces. There are also interactions with the internal eviction cache used to cap the worst case depth. The model keeps count of each and every interaction with memory to report to the total number of requests at the end of execution. Running this model with our aggregation input dataset of size 8M and 2^{20} cardinality, The independent tables algorithm uses 394 million memory requests to answer the query while the cuckoo hash model only uses 51 million requests. That's approximately 7.7x less memory requests, and the gap should grow with larger datasets. The code for this model is available publicly at bitbucket[46].

Chapter 7

Conclusions

Many analytics processes today are becoming more and more irregular as data continues to grow and our processing gets more and more distributed. With data at such massive scales, caches become less and less useful and processing cores spend a majority of their time idle. In this thesis, we have presented several ideas on how to work around the problem of the "memory wall". By employing a coarse-grained temporal multithreading model, we can effectively hide the latency of slow DRAM. FPGAs provide a flexible fabric with the ability to design custom pipelines and FIFO based thread ordering to implement this model. However, it is not limited to FPGAs and all of this work could also be implemented as an ASIC chip.

In a model whose performance depends on a massive number of threads, it is inevitable that some algorithms will need synchronization for correctness. This thesis has shown that we can use Content Addressable memories as extremely flexible synchronizing caches to fill that role. The short cycle memory-lookup means we can use these primitives as a cache to extract performance out of streams that have any type of locality. This caching can boost the performance of our multithreading engines. The quick lookup also means we can store memory addresses and use them as locks to synchronize threads and guarantee serial access to memory. This flexible primitive is platform independent and brings synchronization closer to the processor.

To demonstrate these ideas, this thesis implements this model for several irregular applications: Breadth First Search, Group-By Aggregation, and Hash-join. All applications used both multithreading and CAMs to achieve performance.

In Chapter 3, we explored the use of tools to automate the design of algorithms from the programmers perspective. We looked at both academic projects and commercial offerings and compared their usability and performance. While the tools were useful for some applications, they were still limited in scope and needed significant expertise and hand-holding to guide them to the best hardware circuits. We concluded while the tools were useful for regular applications, irregularity was still a ways off.

In Chapter 4, we implemented a Breadth First Search engine using a ring network of kernels. Each kernel was able to operate independently with hundreds of threads issuing requests. Those memory responses were then able to filter through a CAM to remove redundant work and make sure the bandwidth was efficiently used. For several example graphs, we clearly showed that the multithreading effect builds as the graphs get larger and more time is spent in the steady state issuing requests every cycle. And the CAMs proved to be effective, eliminating between 50% - 80% of redundant jobs.

In Chapter 5, we demonstrated how to use the CAM as a synchronizing cache for aggregation and hash-join. We improved an aggregation engine to move the locking to the lowest level of the hash table to improve the concurrency. We showed a boost in performance over prior FPGA designs and achieved over 10x performance over software designs.

Finally, in Chapter 6 we explored how to further improve these models by improving the base primitives they are built from. One of the challenges of the CAM is getting it to meet timing on the FPGA as the size grows. We showed that there are diminishing returns for the size of the CAM in some contexts, yet we still were not able to reach the size for best performance. We explored how to improve the design of the CAM to get better timing, and developed a flexible, pipelined CAM than can be tuned for application needs. That CAM was able to scale up to 2k deep and still meet 200MHz timing.

This chapter also showed how we could improve the performance of hash based algorithms using a better hash table. Cuckoo hashes provide constant $O(1)$ lookup while also achieving 99% capacity with reasonable worst case insert guarantees. We implemented a software model showing the group-by aggregation using the cuckoo hash and showed it ran with over 5x less memory requests. That should provide more thorughput to the multithreaded design as the memory requests are more efficient and spend less time hopping through the table.

Bibliography

[1] *Altera SDK for OpenCL Best Practice Guide.*

[2] *Altera SDK for OpenCL Programming Guide.*

[3] *Avalon interface specifications.*

[4] *FIPS PUB 197: ADVANCED ENCRYPTION STANDARD (AES).* National Institute for Standards and Technology, November 2001.

[5] *Bluespec SystemVerilog Reference Guide, rev. 17.*, January 2012.

[6] Riverside Optimizing Compiler for Configurable Computing. *http://roccc.cs.ucr.edu/*, 2012.

[7] Xilinx Vivado. *http://www.xilinx.com/tools/autoesl_instructions.htm*, 2013.

[8] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 11:1–11:9, New York, NY, USA, 2016. ACM.

[9] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. Scalable graph exploration on multicore processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[10] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[11] Alpha Data. http://www.alpha-data.com/dcp/capi.php, 2015.

[12] Apache TinkerPop. Property Graph Model.
https://github.com/tinkerpop/blueprints/wiki/property-graph-model.

[13] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 228–235. IEEE, 2014.

[14] D. F. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *Commun. ACM*, 56(4):56–63, April 2013.

[15] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[16] Jason D Bakos. High-performance heterogeneous computing with the convey hc-1. *Computing in Science & Engineering*, 12(6):80–87, 2010.

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

[18] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[19] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast data stream algorithms using associative memories. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 247–256, 2007.

[20] Scott Beamer, Krste Asanovic, David A Patterson, Scott Beamer, and David Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117*, 2011.

[21] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1618–1627. IEEE, 2013.

[22] P. Bertin, D. Roncin, and J. Vuillemin. *Introduction to programmable active memories*, pages 300–309. Prentice Hall, 1989.

[23] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In *Parallel Architectures and Their Efficient Use. Paderborn, Germany*, Lecture Notes in Computer Science, pages 119–130. Springer Verlag, 1992.

[24] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15. IEEE, 2012.

[25] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 37–48, 2011.

[26] Peter A. Boncz, Stefan Manegold, and Martin 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.

[27] Haran Boral and David J DeWitt. Database machines: an idea whose time has passed? a critique of the future of database machines. In *Parallel architectures for database systems*, pages 11–28. IEEE Press, 1989.

[28] Jeremy Branscome, Michael Corwin, Liuxi Yang, James Shau, Ravi Krishnamurthy, and Joseph I. Chamdani. Processing elements of a hardware accelerated reconfigurable processor for accelerating database operations and queries. Patent: US 20080189251 A1, 2008.

[29] Sebastian Breßand Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, August 2013.

[30] D. Buell, J. Arnold, and W. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine.* IEEE CS Press, 1996.

[31] B. Buyukkurt, Z. Guo, and W. A. Najjar. Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs. In *Proc. Int. Workshop On Applied Reconfigurable Computing*, 2006.

[32] Betul Buyukkurt and Walid A. Najjar. Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs. In *FPL, International Conference on Field Programmable Logic and Applications, Heidelberg, Germany*, pages 655–658, 2008.

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

[34] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[35] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, et al. From software to accelerators with legup high-level synthesis. In *Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, page 18. IEEE Press, 2013.

[36] Joao M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):13:1–13:65, June 2010.

[37] Jared Casper and Kunle Olukotun. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, pages 151–160, 2014.

[38] Kevin K. Chang. Understanding and improving the latency of dram-based memory systems. *CoRR*, abs/1712.08304, 2017.

[39] D. Chen and D. Singh. Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *Field Programmable Logic and Applications (FPL)*, pages 5–12, Aug 2012.

[40] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *Design Automation Conference (ASP-DAC)*, pages 297–304, Jan 2013.

[41] Jae Min Cho and Kiyoung Choi. An fpga implementation of high-throughput key-value store using bloom filter. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, 2014.

[42] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *International Conference on Very Large Data Bases*, VLDB '07, pages 339–350, 2007.

[43] John Cieslewicz and Kenneth A. Ross. Data Partitioning on Chip Multiprocessors. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 25–34, 2008.

[44] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[45] Convey Computers. www.conveycomputer.com, 2015.

[46] Cuckoo Hash Model Bitbucket. `https://bitbucket.org/sky_windh/cuckoohashmodel/`.

[47] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From OpenCL to high-performance hardware on FPGAs. In *Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, 2012.

[48] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In Jerzy Swiqtek, Adam Grzech, Paweł Swiatek, and Jakub M.Tomczak, editors, *Advances in Systems Science*, volume 240 of *Advances in Intelligent Systems and Computing*, pages 483–492. Springer International Publishing, 2014.

[49] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[50] David J DeWitt, Robert H Gerber, Goetz Graefe, Michael L Heytens, Krishna B Kumar, and M Muralikrishna. Gamma - a high performance dataflow database machine.

[51] Udit Dhawan and André Dehon. Area-efficient near-associative memories on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):30:1–30:22, January 2015.

[52] Milan Dvorak and Jan Korenek. Low latency book handling in fpga for high frequency trading. In *Design and Diagnostics of Electronic Circuits & Systems, 17th International Symposium on*, pages 175–178. IEEE, 2014.

[53] Stephen A. Edwards. The challenges of synthesizing hardware from c-like languages. *IEEE Design & Test of Computers*, 23(5):375–386, 2006.

[54] Esam El-Araby, Saumil G. Merchant, and Tarek El-Ghazawi. A framework for evaluating high-level design methodologies for high-performance reconfigurable computers. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):33–45, 2011.

[55] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, 2006.

[56] John Feehrer, Sumti Jairath, Paul Loewenstein, Ram Sivaramakrishnan, David Smentek, Sebastian Turullols, and Ali Vahidsafa. The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets. *IEEE Micro*, 33(2):48–57, March 2013.

[57] Tom Feist. Vivado design suite. *Xilinx, White Paper Version*, 1, 2012.

[58] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[59] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter Weinberger. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 243–252, 1994.

[60] Greenplum. http://www.pivotal.io/big-data/pivotal-greenplum-database, 2015.

[61] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144, April 2011.

[62] Zhi Guo, Betul Buyukkurt, John Cortes, Abhishek Mitra, and Walid A. Najjar. A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming*, 36(5):493–520, 2008.

[63] Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pages 249–256, New York, NY, USA, June 2004. ACM Press.

[64] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. Fpga-based multithreading for in-memory hash joins. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[65] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating Join Operation for Relational Databases with FPGAs. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20, April 2013.

[66] Robert J. Halstead, Jason Villarreal, and Walid A. Najjar. Compiling irregular applications for reconfigurable systems. *Int. J. High Perform. Comput. Netw.*, 7(4):258–268, June 2014.

[67] Robert Joseph Halstead. *Using Multithreaded Techniques to Mask Memory Latency on FPGA Accelerators.* PhD thesis, UC Riverside, 2015.

[68] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen. Design and implementation of low-area and low-power aes encryption hardware core. In *Digital System Design: Architectures, Methods and Tools*, pages 577–583, 2006.

[69] Jeffrey Hammes, A. P. Wim Böhm, Charlie Ross, Monica Chawathe, Bruce A. Draper, Robert Rinker, and Walid A. Najjar. Loop fusion and temporal common subexpression elimination in window-based loops. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA*, page 142, 2001.

[70] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

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

[72] J. Hiraiwa and H. Amano. An FPGA implementation of reconfigurable real-time vision architecture. In *Advanced Information Networking and Applications Workshops (WAINA)*, pages 150–155, March 2013.

[73] D.T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–192. CS Press, Loa Alamitos, CA, 1993.

[74] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.

[75] IBM Netezza. www.ibm.com/software/data/netezza/, 2014.

[76] M.F. Ionescu and K.E. Schauser. Optimizing Parallel Bitonic Sort. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 303–309, 1997.

[77] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, August 2017.

[78] Lukas Kekely, Martin Zadnik, Jiri Matousek, and Jan Korenek. Fast lookup for dynamic packet filtering in fpga. In *Design and Diagnostics of Electronic Circuits & Systems, 17th International Symposium on*, pages 219–222. IEEE, 2014.

[79] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[80] Khronos. https://www.khronos.org/news/press/2008/12.

[81] Kickfire. http://www.teradata.com/, 2015.

[82] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, August 2009.

[83] Adam Kirsch and Michael Mitzenmacher. The power of one move: Hashing schemes for hardware. *Networking, IEEE/ACM Transactions on*, 18(6):1752–1765, 2010.

[84] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proc. VLDB Endow.*, 9(4):252–263, December 2015.

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

[86] M. Kumar and D.S. 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, March 1983.

[87] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

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

[89] LLVM. http://llvm.org.

[90] X. Ma, W.A. Najjar, and A.K. Roy-Chowdhury. Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs. *Circuits and Systems for Video Technology, IEEE Transactions on*, PP(99):1–1, 2014.

[91] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, July 2002.

[92] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[93] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[94] Krishnan Meiyyappan, Liuxi Yang, Jeremy Branscome, Michael Corwin, Ravi Krishnamurthy, Kapil Surlaker, James Shau, and Joseph I. Chamdani. Accessing data in a column store database based on hardware compatible indexing and replicated reordered columns. Patent: US 20090254516 A1, 2009.

[95] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[96] Alessandro Morari, Vito Giovanni Castellana, David Haglin, John Feo, Jesse Weaver, Antonino Tumeo, and Oreste Villa. Accelerating semantic graph databases on commodity clusters. In *Big Data, 2013 IEEE International Conference on*, pages 768–772. IEEE, 2013.

[97] Alessandro Morari, Vito Giovanni Castellana, Oreste Villa, Antonino Tumeo, Jesse Weaver, David Haglin, Sutanay Choudhury, and John Feo. Scaling semantic graph databases in size and performance. *IEEE Micro*, 34(4):16–26, 2014.

[98] Roger Moussalli, Ildar Absalyamov, Marcos R Vieira, Walid Najjar, and Vassilis J Tsotras. High performance FPGA and GPU complex pattern matching over spatio-temporal streams. *GeoInformatica*, pages 1–30, 2014.

[99] Roger Moussalli, Mariam Salloum, Robert Halstead, Walid Najjar, and Vassilis J Tsotras. A study on parallelizing XML path filtering using accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):93, 2014.

[100] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proceedings of the VLDB Endowment*, 2(1):229–240, August 2009.

[101] René Müller, Jens Teubner, and Gustavo Alonso. Streams on wires - A query compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.

[102] D. Navarro, O. Lucia, L. A. Barragan, I. Urriza, and O. Jimenez. High-level synthesis for accelerating the FPGA implementation of computationally demanding control algorithms for power converters. *Industrial Informatics, IEEE Transactions on*, 9(3):1371–1379, Aug 2013.

[103] Stephen Neuendorffer, Thomas Li, and Devin Wang. *Accelerating OpenCV Applications.* Xilinx, Inc.

[104] Rishiyur Nikhil. Bluespec SystemVerilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[105] Rishiyur S. Nikhil and Arvind. What is Bluespec? *SIGDA Newsl.*, 39(1):1–1, January 2009.

[106] Oracle Exadata. http://www.oracle.com/engineered-systems/exadata/index.html, 2014.

[107] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[108] Pipelined Cam Bitbucket. `https://bitbucket.org/sky_windh/pipelined_cam`.

[109] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–178. CS Press, Loa Alamitos, CA, 1993.

[110] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[111] N. K. Ratha, D. T. Jain, and D. T. Rover. Convolution on Splash 2. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pages 204–213. CS Press, Loa Alamitos, CA, 1995.

[112] N. K. Ratha, D. T. Jain, and D. T. Rover. Fingerprint matching on Splash 2. In *Splash 2: FPGAs in a Custom Computing Machine*, pages 117–140. IEEE CS Press, 1996.

[113] Kenneth Ross. Efficient hash probes on modern processors. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1297–1301. IEEE, 2007.

[114] 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, April 2012.

[115] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, 3(1-2):1525–1528, September 2010.

[116] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the cell/be processor. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1381–1395, 2008.

[117] Ambuj Shatdal, Chander Kant, and Jeffrey 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.

[118] D. Singh. *Implementing FPGA Design with the OpenCL Standard.* Altera Corporaton.

[119] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, 2012.

[120] Jens Teubner and Rene Mueller. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 625–636, New York, NY, USA, 2011. ACM.

[121] Tran Ngoc Thinh, Surin Kittitornkun, and Shigenori Tomiyama. Applying cuckoo hashing for fpga-based pattern matching in nids/nips. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 121–128. IEEE, 2007.

[122] 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.

[123] TigerMIPS. http://www.program-transformation.org/tiger/mips.

[124] N. Trevett. OpenCL introduction. SIGGRAPH Asia, 2013.

[125] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Prog. Custom Comp. Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127 –134, may 2010.

[126] Peter Benjamin Volk, Dirk Habich, and Wolfgang Lehner. Gpu-based speculative query processing for database operations. In Rajesh Bordawekar and Christian A. Lang, editors, *ADMS@VLDB*, pages 51–60, 2010.

[127] J.E. Vuillemin, P. Bertin, D. Roncin, M. Sh, H. Touati, A.H., and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Trans. Very Large Scale Integr. Syst.*, 4(1):56–69, 1996.

[128] Kevin Wadleigh, John Amelio, Kirby Collins, and Glen Edwards. Poster: Hybrid breadth first search implementation for hybrid-core computers. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1355–1355. IEEE, 2012.

[129] Li Wang, Minqi Zhou, Zhenjie Zhang, Ming-Chien Shan, and Aoying Zhou. NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains. *Knowledge and Data Engineering, IEEE Transactions on*, 27(4):1071–1084, April 2015.

[130] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[131] Skyler Windh, Prerna Budhkar, and Walid A. Najjar. CAMs as Synchronizing Caches for Multithreaded Irregular Applications on FPGAs. ICCAD'15, pages 331–336, 2015.

[132] Skyler Windh, Xiaoyin Ma, Robert J Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[133] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *Field-Programmable Technology (FPT)*, pages 362–365, Dec 2013.

[134] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth 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*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[135] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth 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.

[136] Xilinx Zynq. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, 2015.

[137] xkcd: Compiling. https://www.xkcd.com/303/.

[138] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9, 2011.

[139] Peter Yiannacouras and Jonathan Rose. A parameterized automatic cache generator for FPGAs. FPT'03, pages 324–327, 2003.

[140] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25. IEEE Computer Society, 2005.

[141] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.*, 6(12):1374–1377, August 2013.

[142] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. *High-Level Synthesis: from Algorithm to Digital Circuit*, 2008.

[143] Jianlong Zhong and Bingsheng He. An overview of medusa: simplified graph processing on gpus. In *ACM SIGPLAN Notices*, volume 47, pages 283–284. ACM, 2012.

[144] Shijie Zhou and Viktor K Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.