UNIVERSITY OF CALIFORNIA
RIVERSIDE

Accelerating Irregular Applications Using Latency Masking Multithreaded Techniques

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

Doctor of Philosophy

in

Computer Science

by

Prerna Budhkar

September 2018

Dissertation Committee:

Dr. Walid Najjar, Chairperson
Dr. Vassilis Tsotras
Dr. Laxmi Bhuyan
Dr. Rajiv Gupta

Copyright by
Prerna Budhkar
2018

The Dissertation of Prerna Budhkar is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The pursuit of Ph.D. has been a period of fruitful learning experience for me, not only in the academic arena, but also on a personal level. I would like to reflect on the many people who have supported and helped me to become who I am today. First and foremost, I would like to thank my advisor, **Dr. Walid Najjar**. He has given my research a direction and a push at every step. Throughout these five years, he has been supportive of me in every situation, academic or otherwise.

I would also like to express my gratitude to **Dr. Vassilis Tsotras**. His technical and editorial advice has been invaluable to my research. I have always felt comfortable in approaching him with the silliest of my doubts.

I would like to thank my other dissertation committee members, Dr. Rajiv Gupta and Dr. Laxmi Bhuyan for taking their time to review this dissertation.

During graduate school, I have met many wonderful fellow graduate students and friends whom I am grateful to. Skyler Windh is a great friend and is always warm and welcoming when I approached him with ideas and problems. I really admire the way he is able to cut through red tape and accomplish what needs to be done. Jose Rodriguez, thank you for being a patient friend. Our long technical discussions have saved me tons of time in my projects.

Graduate school is a long journey that sometimes suddenly hits you the hardest when things don't go your way. I feel very grateful for the many friends who were there to help me grind through it. Thank you *Sayali, Vidya and Divya* for keeping a check on me

all the time. Finally, thank you *Jason Ott* for being such a good friend. Our evening walks back home and the discussion about life and philosophy were the best part of my day.

I would like to thank my family for their enormous support and sacrifices that they made. My grandmother, *Aaji*, has been a pillar of support throughout my life. I am grateful to her strong belief in education. My mother, *Aai*, has always been there kindly supporting me with unwavering love. I owe all of my success to these two strong women in my life. I would like to thank my grandfather, *Aajoba*, for his love and support. I am still stumped by my father, *Baba*, for nonchalantly trusting me with all my life decisions - thank you. Thanks to my brother, *Vaibhav*, who is always available to talk to me at any hour of the day. I am also forever indebted and grateful to my in-laws for their love, understanding and constant prayers. Also, special thanks to my extended family: *Aaji, Ajoba, Smita Aatya, Ramesh Kaka, Jyotsna Aatya, Anant Kaka, Mama and Mami.* You have always stood by us in good and bad times. Thanks to all my cousins for keeping me connected to the family back in India.

None of this would have been possible without the unconditional love, co-operation and belief from my better-half, *Vineet*, for understanding and putting up with me through my grad-student life of classes, publications, accepts, rejects, missing out on family and the many, many little pleasures. I hope to make it up to you!

Finally, this thesis contains one of my previously publish works. This work is included in Chapter 3 of this thesis. The full citation is:

Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. 2016. FPGA-accelerated group-by aggregation using synchronizing

caches. In Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16). ACM, New York

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Accelerating Irregular Applications Using Latency Masking
Multithreaded Techniques

by

Prerna Budhkar

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

The last two decade has witnessed two opposing hardware trends where the DRAM capacity and the access bandwidth has rapidly increased by 128x and 20x respectively. In stark contrast with capacity and bandwidth, DRAM latency has almost remained constant, reducing by only 1.3x in the same time frame. Therefore, long memory latency continues to be a critical performance bottleneck in modern systems. Another emerging trend is the stagnating processor clock speeds due to the end of Dennard scaling. Parallel architectures, like CPUs and GPUs, resolved this problem by increasing parallelism, but developed architectures that rely extensively on *data locality* in the form of large cache hierarchies for multicores, and vectorized execution for SIMD-enabled CPUs and GPUs.

At the same time, many data-intensive applications are moving away from data locality towards irregular memory access behavior. This behavior is observed either because of dataflow (caused by indirection in data access) or control flow (caused by branch instructions) irregularity. Such applications are often memory bound and their performance is primarily determined by the memory latency (also known as the memory wall).

An alternative approach to mask long memory latencies is by using *multithreaded* execution where a running thread relinquishes execution to a ready thread as soon as it performs a long-latency memory access. This dissertation explores how custom hardware accelerators using memory masking multithreaded techniques can be used to improve performance of irregular applications. In hardware multithreaded designs thread states are maintained on-chip, and with enough application parallelism they can fully mask memory latency without storing data in caches.

This thesis explores latency masking *hardware multithreading* on three different applications to achieve better performance on irregular applications. The first two applications, namely *selection* and *group-by aggregation* are relational database operators. The selection operator examines different conditions on various row attributes. Depending on the evaluation, a row is either qualified or disqualified. This operator, therefore, exhibits a control flow irregularity. The proposed selection design shows an improvement of 1.8x over CPU Similarly, it is 3.2x more bandwidth efficient than GPUs. Overall, this dissertation provides the first direct comparison study of selection operator on all three architectures. The group-by aggregation, on the other hand, is a hash-based implementation that exhibits dataflow irregularity. Results show that the FPGA-accelerated approach significantly outperforms CPU-based implementations and yields speedup up to 10x. The third application considered for evaluation is a popular sparse linear algebra operation namely *Sparse Matrix and Vector multiplication* (SpMV). Our results show that the multithreaded SpMV implementation achieves up to 95% of the theoretical upper bound performance whereas GPUs, like Titan GV100, can achieve upto 69% of its peak performance.

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

2 Related Work **8**

2.1 Reconfigurable and Heterogeneous Architectures 9
2.2 Multithreaded Architectures 10
2.2.1 Latency Masking Multithreaded Architectures 12
2.2.2 Custom Multithreaded Architecture 13
2.2.3 Convey HC-2ex: Target Platform 15
2.3 Query Processing on FPGAs 18
2.3.1 Group-by Aggregation 20
2.3.2 Selection Scan Operator 24
2.4 Sparse Matrix-Vector Multiplication 28
2.4.1 Sparse Matrix Formats 29
2.4.2 GPU Approaches 32
2.4.3 FPGA Approaches 33

3 Multithreaded Group-by Aggregation **35**

3.1 Using CAMs on FPGA 36
3.2 Aggregation Engine Workflow on FPGA 38
3.3 FPGA design and Optimizations 41
3.4 Evaluating Group-by Aggregation 43
3.4.1 Software Implementations 43
3.4.2 Throughput Evaluation 45
3.4.3 Merge Overhead on FPGA 48
3.4.4 Performance Analysis 50
3.5 FPGA Area Utilization 50
3.6 Conclusion 51

4 Multithreaded Selection Operator **53**
4.1 MTP Selection Engine . . . 54
4.1.1 Engine Design . . . 54
4.1.2 Engine Workflow . . . 56
4.1.3 MTP Performance Analysis . . . 57
4.2 Evaluating Selection Operator . . . 63
4.2.1 Throughput Evaluation . . . 63
4.2.2 Throughput Efficiency . . . 66
4.2.3 Multithreaded Execution of Selection . . . 68
4.2.4 TPC-H Query Evaluation . . . 69
4.2.5 Datalayout Independence . . . 72
4.2.6 Power Utilization . . . 74
4.3 FPGA Area Utilization . . . 75
4.4 Conclusion . . . 76

5 Sparse Matrix and Vector Multiplication **78**
5.1 SpMV Architecture Overview . . . 79
5.2 Performance Bound . . . 82
5.3 Experimental Results . . . 85
5.3.1 Experimental Setup . . . 85
5.3.2 NNZ Vs Sustained Bandwidth . . . 87
5.3.3 Performance Evaluation of Multithreaded SpMV . . . 92
5.3.4 Performance Comparison to GPUs . . . 95
5.3.5 Scalability . . . 99
5.3.6 Existing FPGA-based SpMV Accelerators . . . 100
5.4 Conclusion . . . 101

6 Conclusions **103**

Bibliography **106**

List of Figures

1.1 Trends illustrating DRAM improvements (referenced from [24]) 2
1.2 Trends illustrating peak clock rate for memory access and the CPU performance (referenced from [15]) 2

2.1 Different types of multithreading techniques. 11
2.2 Custom Multithreaded Architecture 14
2.3 The Convey HC-2ex architecture is divided into software and hardware regions as shown in (a). Each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b) 16
2.4 Example matrix 29
2.5 Coordinate Sparse Matrix Format 30
2.6 Compressed Sparse Row (CSR) Matrix Format 30
2.7 ELLPACK sparse matrix format 31

3.1 Engine Internal Blocks. 38
3.2 A state diagram for jobs in the aggregation engine. 39
3.3 Stall percentage for Tuple Request, HT Lookup and Filter CAM modules for different datasets 41
3.4 Alternative engine placement strategies on a single FPGA with 16 memory channels. 43
3.5 Aggregation throughput of hardware and software approaches for datasets with 256M tuples. 46
3.7 Aggregation throughput of hardware and software approaches for datasets with 256M tuples. 48
3.8 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. 49
3.9 Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the Multiplexed FPGA design for varying dataset sizes and key cardinalities. 49

4.1 Figure (a) represents the sample SQL query which is used as an example query in this section. Figure (b) present the corresponding *predicate control block* (PCB) for the sample query 55
4.2 Selection Accelerator Engine: showing different building blocks and memory channels that read/write from memory 56
4.3 Number of qualified rows after each predicate for a conjunctive query. N is the total number of rows, p is a predicate probability, C_0,C_1,C_2,...,C_k designate k different predicates. 59
4.4 Number of qualified rows after each predicate for a disjunctive query. N is the total number of rows, p is a predicate probability, C_0,C_1,C_2,...,C_k designate k different predicates. 61
4.5 Variance in predicate probability with respect to selectivity for conjunctive (a) and disjunctive(b) query. 62
4.6 Query evaluation runtime measured on MTP, CPU and GPU with varying selectivity and number of predicates 64
4.7 Throughput achieved by MTP, CPU and GPU implementation normalized to their respective bandwidth. Note that the legend description is same as that of Figure 4.6. 64
4.8 Absolute attributes evaluated as the selectivity and number of predicates are varied for (a) conjunctive 68
4.9 Selectivity of TPC-H queries. Each color marks the range of selectivity: (blue) above 60%, (orange) 50%-20%, (green) below 10%. 70
4.10 TPC-H Query6 70
4.11 TPC-H query Q6 performance evaluation. 71
4.12 Performance comparison of the CPU and the MTP implementations with row-major and columnar data layouts. 73
4.13 Comparison of Power Efficiency on MTP, CPU and GPU systems. 74

5.1 SpMV design on FPGA. 79
5.2 Memory controller state machine 80
5.3 Minimum number of non zeros as number of engines (channels) are increased 89
5.4 Sparse matrix structure for (a) *Group 1* and (b,c) *Group2* 92
5.5 Analytical and Experimental Performance Comparison for Group 1 and Group 2 sparse matrices 93
5.6 Effect of block size on the achieved SpMV performance when ran with 8 PEs. As long as number of partitions $\gg$ than number of PEs, the block size does not affect the performance. 95
5.7 Comparison between the achieved and the theoretical upper bound performance for (a) multithreaded SpMV (b) Tesla K40 (c) Titan GV100 98
5.8 Comparison of bandwidth Utilization of multithreaded and GPU-based SpMV implementation 99
5.9 SpMV performance scaling with number of PEs 100

List of Tables

3.1 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 the hash bucket, holding the bucket list's head pointer 40
3.2 FPGA resource utilization for aggregation engines. 51

4.1 Query and input relation parameters used in the analytical model 58
4.2 Summary of relationships between S, k, W_e and W_{wr}. 61
4.3 System Configuration of different architectures used for evaluation. 63
4.4 Performance Evaluation of TPC-H query Q6 on CPU, GPU and MTP implementations. 71
4.5 FPGA area utilization for the selection engine 75

5.1 FPGA resource utilization for SpMV engines. 87
5.2 Benchmarks used for performance evaluation of multithreaded SpMV design 90
5.3 Prior FPGA-based SpMV designs and implementation. '*' and '**' presents Group 1 and Group 2 data respectively. 101

Chapter 1

Introduction

The main factors influencing in-memory processing performance are the high DRAM latency and the growing gap between the memory bandwidth and the speed of the processing unit aka, *the memory wall.* The performance bottleneck due to high DRAM latency becomes more evident by observing the historical trends of DRAM-chip. The work presented in [24] illustrates the historical trends of DRAM-chip. As shown in Figure 1.1, DRAM-based main memory has made rapid progress on capacity and bandwidth, improving 128x and 20x respectively over the past two decades. These improvements mainly follow Moore's Law and Dennard Scaling, which enable more and faster transistors along with more pin. On the contrary, DRAM latency has improved (i.e., reduced) by only 1.3x, a negligible change compared to changes in DRAM capacity and bandwidth during the same time period. As a result, long DRAM latency remains as a significant system performance bottleneck for many modern applications such as in-memory databases [71, 14, 66, 84], graph traversals [28, 82, 89] and Google's datacenter workloads [51].

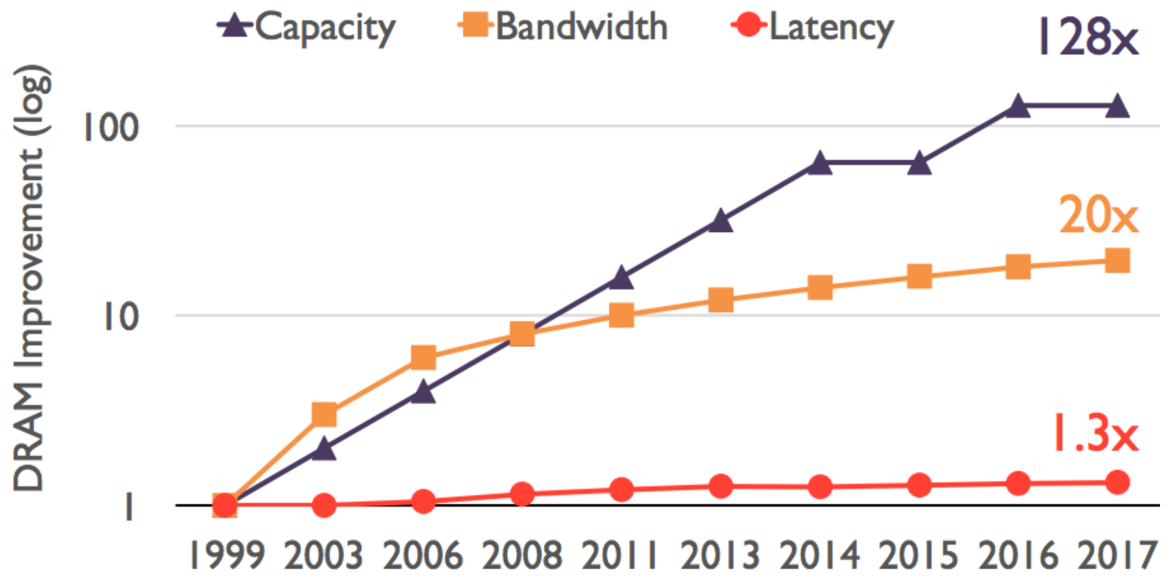


Figure 1.1: Trends illustrating DRAM improvements (referenced from [24])

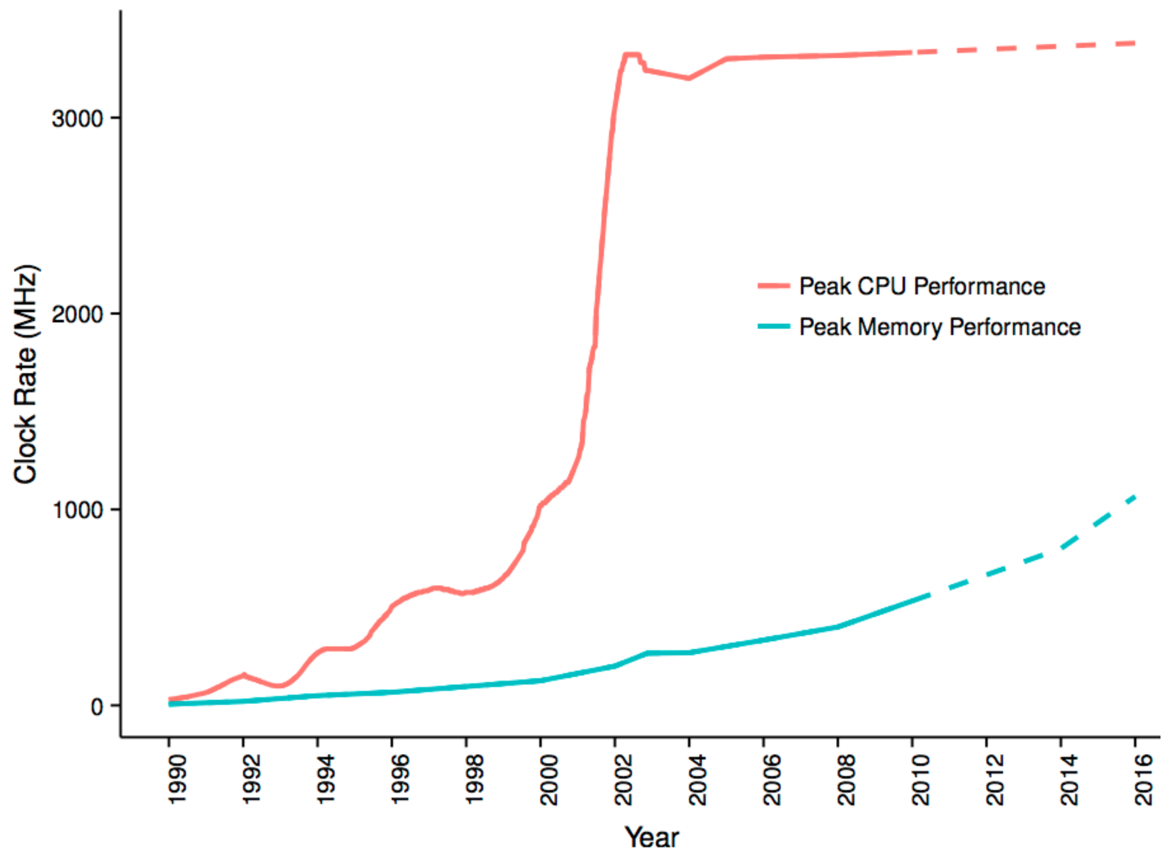


Figure 1.2: Trends illustrating peak clock rate for memory access and the CPU performance (referenced from [15])

The second major performance issue, as mentioned before, is the growing gap between the memory bandwidth and the speed of the processing unit, the so-called *'memory wall'*. The term "memory wall" was coined in [88] by Wulf and McKee in 1995. They predicted that the growing disparity between processor and the memory speed would, in the future, lead to a situation where memory access time will outweigh any performance improvements in the processor. The aforementioned behavior was more accurately modeled using Equation 1.1 that calculates the average memory access time. In this equation t_c and t_m are the cache and DRAM access times and p is the probability of cache hit.

$$t_{avg} = p * t_c + (l - p) * t_m \tag{1.1}$$

Wulf and McKee showed that greater the divergence between t_c and t_m, longer is the memory access time. Memory references eventually dominate the overall system performance and further improvements to CPU performance will have no impact.

A decade after the original paper was published, McKee reflected upon their predictions in [59]. They reported that some commercial applications such as transaction processing workloads were seeing 65% processor idle times and scientific computing applications were suffering from memory bottlenecks with up to 95% node idle times. Much of this behavior was observed because of the memory bottlenecks where the processor was stalled waiting for data to be returned from memory. It was obvious that handling off-chip memory accesses would become critical in achieving better system performance. This growing gap between the processor and the memory is also depicted in Figure 1.2(taken from [15]). It can be seen that even though the CPU growth halted at around 2003, the memory speed has a lot to catch up. Modern processors can now generate 3 orders of magnitude

more requests than the memory architectures can fulfill. This high request rate coupled with long access latencies means that processors spend the majority of their time idle.

The problem of high DRAM latency and the memory wall is more important for multicore CPUs given their higher clock speeds. Multicore CPUs addressed this problem by introducing large *cache hierarchies* that rely on data locality. Caches leverage the spatial and temporal locality within an application to reduce the number of requests back to main memory. Temporal locality exists when a data that is currently accessed by an application is very likely to be accessed again. Similarly, spatial locality exists if a piece of data that has been accessed recently, it is likely that the adjacent pieces of data will be accessed in the near future. Caches exploit these properties to keep values closer to the processing core where the latency is much lower. Hardware accelerators, such as FPGAs and GPUs, deal differently with long memory latencies. FPGA based accelerators rely, mostly, on streamed data; hence the latency cost is paid once for the first element. GPUs on the other hand, offer a different solution leveraging on massive SIMD parallelism and high bandwidth specialized memory (i.e. GDDR). However these architectural solutions still inherently rely on data locality.

The aforementioned locality based solutions work well for many applications, but there exists an important category of applications that do not work well on caches. We call them irregular applications and by definition they have poor locality. There are two measures of irregularity: (1) *Dataflow irregularity* is caused by the indirection in the data access, leading to cache misses. Example applications include sparse linear algebra, hashing based applications etc; (2) *Control flow irregularity* is caused by the dynamic control flow

and leads to branch mis-prediction; this contributes to a large fraction of stall time on CPUs [68, 79] or thread divergence on GPUs [74]. Some of the applications that exhibit control flow irregularity are SQL selection operation, building histograms, breadth first search etc. As a consequence of these irregularities, an application tends to pull data from many different memory locations jumping around memory in seemingly random ways. This behavior introduces a long latency to fetch data from memory.

Over the past few years, many real world applications are drifting towards irregularity, making it difficult for the cache-centric architectures to achieve the maximum performance. However, with the advent of new heterogeneous architectures like Convey HC-2ex, Intel HARP, Stratix 10 with HBM2, it has become easy for the hardware architects to prototype designs without relying on any cache support. Moreover, the applications can enjoy best of both worlds. Caches can be used for regular applications while irregular application can be offloaded to the accelerators.

Without relying on caches, an alternative approach to mask long latencies is by using hardware *multithreading* [70, 50, 80]. Several multithreading models (*simultaneous, fine-grained, coarse grained*) have been proposed. They can be categorized by how temporally close, instructions from different threads may be executed. On general purpose processors, executing multiple threads concurrently requires saving the full context of each thread. This limits the amount of parallelism that can be achieved on these systems.

In a custom architecture (e.g., FPGA) where the datapath is designed for a small number of predefined operations, the required context for each thread is much smaller

than in a general-purpose CPU and hence more threads can be supported. In this model, parallelism is limited only by the number of active threads (ready, executing or waiting).

In this thesis, we explore custom hardware *multithreading* that masks long memory latencies incurred by both data flow irregularity and control flow irregularity. The applications used for data flow irregularity are *group-by aggregation* and *Sparse Matrix and Vector Multiplication (SpMV)* and the *relational select operator* is used for control flow irregularity. We prototype our proof-of-concept on the Convey HC-2ex FPGA machine.

There are two possible implementations of group-by aggregation, hash-based and sort-based. The former is generally preferred because it avoids the high penalty of sorting the input relation. Hence, in this thesis we focus on an in-memory hash-based implementation for group-by aggregation. Hash tables rely on good hashing functions that randomly distribute keys across a range of values, thereby exhibiting a dataflow irregularity.

On the other hand, selection operator is generally used to select rows from a table (relation) that satisfies some given conditions. While selection has a seemingly regular execution pattern (i.e., exhibits spatial locality), it often leads to control flow irregularity while evaluating the selection predicates. In most cases, selection appears early on within a query plan - right above the data scan operator - its performance directly affects the total runtime of the whole query.

Similarly, Sparse Matrix-Vector Multiplication (SpMV) has also received significant attention due to its increasingly important applications in scientific and commercial applications. Although SpMV is a highly parallelizable, the real world sparse matrices often restrict realizable parallelism. The reasons are two fold. On one hand, low compute to

memory ratio makes SpMV a memory bound problem and on the other hand, it exhibits a data flow irregularity while fetching an input or output vector from a memory space which is far too big for cache. Hence, it becomes difficult to utilize the main memory bandwidth which is already scarce.

The rest of the dissertation is organized as follows. Chapter 2 discuss literature survey relevant to this thesis. It includes historical memory trends, multithreading techniques, query processing and SpMV implementations on accelerators. In Chapter 3, we present multithreaded in-memory group by aggregation. Our results demonstrate a speed up of up to 10x over the best multicore software algorithms. In Chapter 4, we present a detailed comparison study of the selection operator on multi-core CPUs, GPU and multithreaded FPGA implementation. In an important parameter space, the multithreaded design achieves a speed of 1.4x-4.6x over CPU SIMD and 1.4x-6.7x over GPU implementations. Unlike other architectures, this design is independent of the data layout. In Chapter 5, we explore the multithreaded implementation of the SpMV kernel. Using a classical roofline model, we theoretically derive the upper bound performance of the SpMV implementation and compare it to our experiments. Our results show that the model is accurate up to 95%. We also demonstrate that the multithreaded SpMV implementation achieves much higher fraction, up to 95%, of the upper bound performance when compared to GPU that achieves up to 69% of their respective upper bound.

Chapter 2

Related Work

Custom architectures like FPGAs are becoming increasingly attractive platforms for many applications. FPGAs provide cheaper solution and faster time to market as compared to ASICs which normally require a lot of resources in terms of time and money to obtain first device. FPGAs are generally used for streaming applications and are therefore known to perform well on regular applications. However, this streaming paradigm again implicitly assumes the existence of some form of locality within the stream since the on-chip memory on most FPGA devices is very limited.

This thesis suggests an alternative approach that expands the application domain of FPGAs by considering applications that cannot be streamed. Techniques like latency masking and multithreadding accompanied by a custom pipelined datapath, allow FPGAs to decide the memory locations it needs at runtime. We apply our designs to SpMV and database applications.

2.1 Reconfigurable and Heterogeneous Architectures

When the performance of algorithms is insufficient on general purpose processors, the architecture community turn towards hardware implementations specifically designed to implement the desired algorithm. Application Specific Integrated Circuits (ASICs) are hardware chips designed to implement one specific algorithm and are able to provide fast and efficient execution. However, the ASIC fabrication process is time consuming and a small change in the design will require fabrication of a whole new ASIC. This process also involves high implementation and deployment costs. Reconfigurable computing, on the other hand, allows relatively easy development cycle of hardware circuits. The hardware can be incrementally improved whilst maintaining a greater degree of flexibility than ASICs.

Field Programmable Gate Array (FPGA) is the most common reconfigurable platform. The way FPGAs typically implement combinatorial logic is with LUTs (LookUp Tables). In general terms LUT is basically a table that determines what the output is for any given input(s). In the context of combinational logic, it is the truth table. This truth table effectively defines how your combinatorial logic behaves. They are also often coupled with coarser-grained functional units such as dedicated multipliers or Digital Signal Processing (DSP) blocks as well as small internal memories, called as BRAMs. As their name suggests, they can be reconfigured with new logic designs even 'in the field' after deployment without the costly tooling overheads of ASICs.

The conventional wisdom dictates that pointer intensive, or sparse applications are not well suited for FPGA-based code acceleration as these algorithms are often memory bound rather than compute bound and therefore will see limited benefit from the massive

parallelism offered by reconfigurable hardware. However, heterogeneous architectures provide some unique opportunities for improving memory performance thanks to low latency links between a general purpose CPU and customizable hardware, both sharing high speed links to external memory. This thesis seeks to examine how the alternative data paths and processing options provided by these platforms can be used to improve memory access performance of irregular algorithms over other platforms.

2.2 Multithreaded Architectures

A multithreaded processor concurrently executes instructions from different threads of control within a single pipeline. Figure 2.1 presents different multithreaded architectures. Each row represents the issue slots for a single execution cycle: a filled box indicates that the processor found an instruction to execute in that issue slot on that cycle; an empty box denotes an unused slot. Figure 2.1(a) shows a sequence from a conventional superscalar, executing a single program, or thread, from which it attempts to find multiple instructions to issue each cycle There are two basic types of multithreaded processors: those that issue instructions only from a single thread in a cycle, and those that issue instructions from multiple threads in the same cycle. Many advanced out-of-order superscalar processors such as the IBM Power7 and Power8 [73] or the latest Intel architectures, like Nehalem [54], support the simultaneous multithreading (SMT) technique. SMT keeps multiple threads active in each core. The processor identifies independent instructions and simultaneously issues them to the cores various execution units, thereby maintaining high utilization of the processor resources.

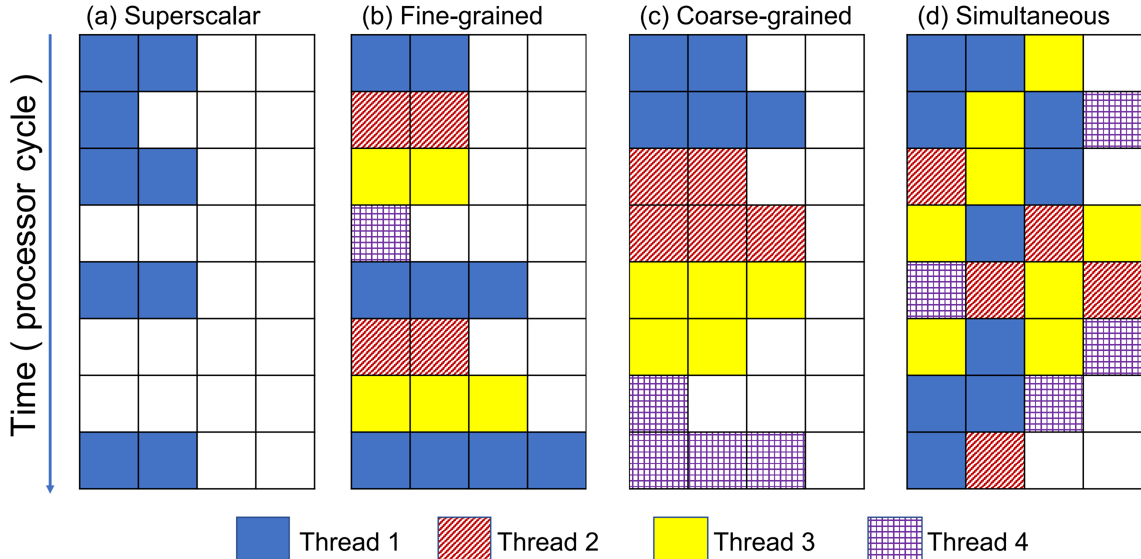


Figure 2.1: Different types of multithreading techniques.

Figure 2.1(d) shows how each cycle an SMT processor selects instructions for execution from all threads. It exploits instruction-level parallelism by selecting instructions from any thread that can (potentially) issue. The processor then dynamically schedules machine resources among the instructions, providing the greatest chance for the highest hardware utilization. If one thread has high instruction-level parallelism, that parallelism can be satisfied; if multiple threads each have low instruction-level parallelism, they can be executed together to compensate. In this way, SMT can recover issue slots lost to both horizontal and vertical waste.

On the other hand, processors that issue instructions from a single thread every clock cycle are known as temporal multithreaded processors. The execution alternate between different threads to keep the (usually in-order) pipeline filled and avoid stalls. This technique was used in the SUN UltraSparc T5 [37] and Tera MTA [8, 7] (now Cray XMT [48]). Temporal multithreaded architectures are generally better suited for irregular

applications because they can tolerate long latency memory accesses by switching to other ready threads while the memory subsystem loads or writes the data, thereby not necessarily requiring caches to reduce access latencies [80].

2.2.1 Latency Masking Multithreaded Architectures

Memory masking multithreaded architectures have existed since the beginning of the computer architecture field. It is a simple idea. Research scientists measured the number of clock cycles it took to fulfill a memory request. In the Horizons case [77] most requests averaged between 50 to 80 cycles, but almost all requests could be handled within 128 cycles. The architects then built custom processors to supported that many outstanding requests; 128 threads in the case of the Horizon. The processors had very fast context switching (one clock cycle) so that once a request was issued by a thread it could immediately switch to another thread. In this way the processor was fully utilized. In the worst case all 128 threads would issue a memory request. However, by the time the 128th request was issued the 1st request would be fulfilled, and the processor could continue running without interruption. This technique is called memory masking, and is integral to a multihreaded architectures performance.

Similarly, the Tera Computer Company released its Tera MTA [8, 7] machine. Each of its processors could run at 300 MHz, and they could support 128 hardware threads.The only physical machine, that we are aware of, was installed at the San Diego Supercomputer Center [1] and it contained 4 processors. Therefore, It could support 512 threads. To lower the network traffic the MTAs instructions were fetched through a shared cache. However, it had no data cache and relied purely on multithreading to mask the memory latency.

Starting in 2005 Sun produced a line of multithreaded multicore processors based on its UltraSPARC architecture. The first processor (UltraSPARCT1) had 8 cores that could support 4 threads concurrently. Each core would context switch between active threads each cycle. This increased the overall latency of a single thread, but allowed the cores to be better utilized. Threads did not need to be from the same application, but cache performance improved if the threads were accessing the same data locations. Sun steadily improved the architecture by adding more floating-point units, memory bandwidth, and increased the number of cores. However, when the UltraSPARC T4 was released they decreased the number of cores, but dramatically increased the clock frequency. This was a move to broaden their target audience by improving the single thread performance.

Nevertheless, these architectures support a relatively *small* and predetermined number of threads (up to 128 on the MTA and up to eight threads per core on the UltraSparc T5). This is because the CPU has to save a huge hardware context (in the form of registers, program counters, stack pointers, etc) for each ready/waiting thread. This context is saved every time the execution switches between threads and is read later when a thread resumes. As a result, the number of threads that can be run in parallel is limited by the total number of physical cores (or hardware contexts for CPUs with hyper-treading).

2.2.2 Custom Multithreaded Architecture

The customized multithreaded architecture exploited in this thesis can be considered as a blend of SMT and the temporal multithreading (discussed in Chapter 2). The structure of the data-path and the number of thread states are designed for the specific target application. Like SMT, in this architecture too, a running thread relinquishes exe-

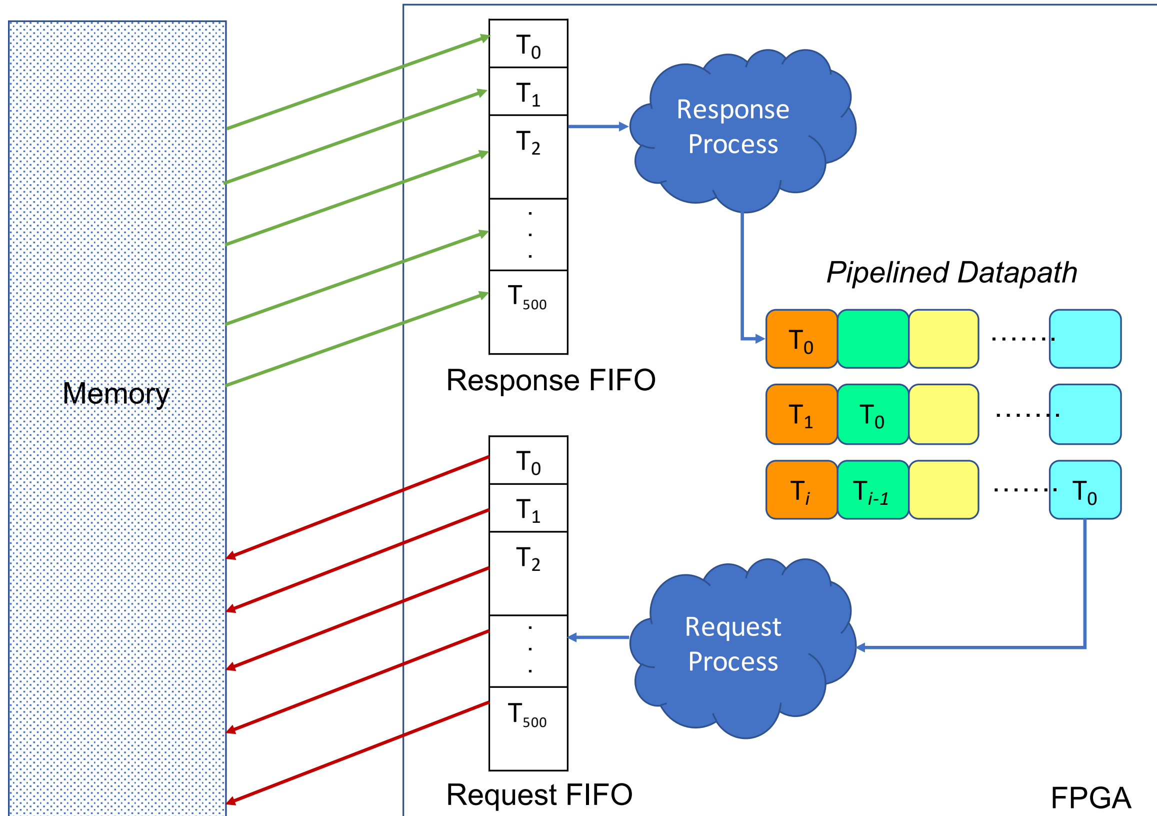


Figure 2.2: Custom Multithreaded Architecture

cution to a ready thread as soon as it performs a long-latency operation. Additionally, the deeply pipelined datapath on FPGA allow different threads to enter the pipeline on each cycle. Moreover, the datapath in a custom architecture (e.g. FPGA) is designed for a small number of predefined operations. As a result, the required context for each thread is much smaller than in a general-purpose CPU and hence more threads can be supported.

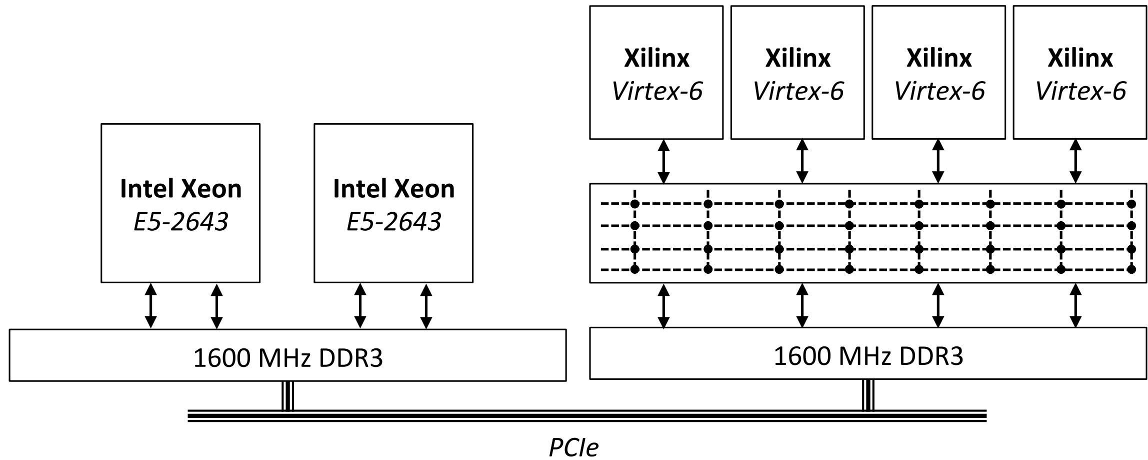
In any multithreaded setup there are typically three states of execution: (1) build-up state (2) steady state and (3) drain state. In a build-up state, threads are generated in a such a way that number of memory accesses in flight is sufficient enough to cover the bandwidth delay product. In an ideal setup, As soon as the last request is sent, the

responses are received that fills the pipeline of a custom datapath. In a steady state, a kernel/core can maintain sufficient in-flight requests that can hide the memory latency. However, towards the end of execution, threads start terminating either because the job is done or all the input data is processed. When this happens, the number of in-flight requests also start decreasing and can no longer mask the latency. If the steady state execution is longer than other two states, the build-up and the drain costs are amortized.

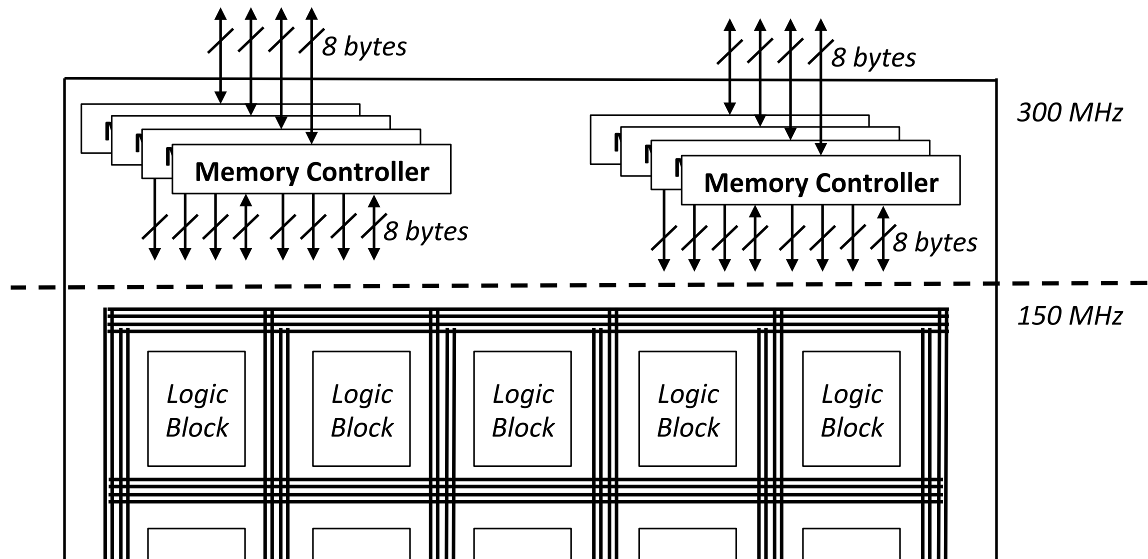
Figure 2.2 describes different components of this model. The execution starts by generating multiple threads (requests) by *'request process'* module. Assuming a fully pipelined datapath and a buffered memory interface, the expected parallelism in such a system equals to *memory bandwidth* × *memory latency* (Little's Law). The number of threads that are generated should commensurate with the memory latency. For instance, the average memory latency on Convey HC-2ex, our target machine, is 500 cycles. Therefore, we generate 500 threads. The idea is simple. As soon as the last memory request is sent, the memory response for the first request is received and is fed to the pipelined datapath. This imply that the parallelism is limited only by the number of outstanding requests that can successfully mask the memory latency.

2.2.3 Convey HC-2ex: Target Platform

The Convey HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions, allowing us to directly compare the FPGA and CPU in-memory implementations on the same memory architecture. The coprocessor supports multiple instruction sets (referred to as "personalities"), which are optimized for



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



(b) Convey HC-2ex FPGA AE Wrapper

Figure 2.3: The Convey HC-2ex architecture is divided into software and hardware regions as shown in (a). Each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b)

different workloads and dynamically reloaded when an application is run. Each personality includes a base set of instructions that are common to all personalities, as well as extended instructions that are designed for a particular workload. As shown in Figure 2.3a, the memory is divided into regions connected through PCIe with portions closer to the CPU,

and portions closer to the FPGAs. Each processor (CPU or FPGA) can access data from both regions, but data accesses across PCIe are significantly longer. The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache.

The hardware region has 4 Xilinx Virtex6-760 FPGAs, called application engines (AE), connected to the global memory through a full crossbar. The crossbar not only interfaces the AEs to the memory modules but also supports the in-order return of all memory requests. Our multi-threaded designs utilizes this re-ordering to serve all the concurrent threads in the order of their arrival. Each AE also has 8 64-bit memory controllers (MC) running at 300MHz (Figure 2.3b). The MCs translate virtual to physical addresses on behalf of the AEs, and include snoop filters to minimize snoop traffic to the host processor. The Memory Controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs, which are optimized for transfers of 8-byte bursts and provide near peak bandwidth for non-sequential 8-byte accesses. The coprocessor therefore provides a much higher peak bandwidth, and often can deliver a much higher percentage of that bandwidth, than what is available to commodity processors.

The hardware logic on each AE run in a separate 150 MHz clock domain to ease timing and is connected to the memory controllers through 16 *memory channels*. On the HC-2ex, all the reads and writes to the memory are done through these channels. Each channel supports independent and concurrent read-write accesses to memory. The memory latency varies, according to the target memory bank and traffic congestion, between 400 and 800 cycles. Assuming all the channels are used for reads and an average latency of 500 cycle this means there are 8000 (8*400) outstanding memory requests per accelerator

FPGA or 32,000 for the whole system. Obviously, stalls in the pipeline and other hazards reduce the throughput. However, these numbers provide a good estimate of the possible parallelism and the massive number of threads.

2.3 Query Processing on FPGAs

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 [19]. 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 [33, 32]. 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 proposed building hybrid CPU-GPU systems [92, 21, 42, 83, 40]. 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 [62] 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 [69]. The Q100 [87] 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 [3] 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 [75]. 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 [2], 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 [20, 47, 60].

2.3.1 Group-by Aggregation

The performance characteristics of the aggregation operator differs very much depending on the number of groups (distinct keys). If there are few groups, aggregation is very fast because all groups fit into cache. If, however, there are many groups, many cache misses happen. Contention from parallel accesses can be a problem in both cases (if the key distribution is skewed). To achieve good performance and scalability in all these cases, without relying on query optimizer estimates, multi-core CPU architectures provide two main alternatives. The *hardware-conscious* algorithms are tightly tailored to the underlying hardware and perform preliminary data partitioning to reduce cache misses. Instead, the *hardware-oblivious* solutions try to mask latency by relying on hardware-provided multithreading. These contrasting approaches were extensively studied in the context of in-memory hash joins [17, 11] as well as sort-merge joins [12, 53]. Hardware-oblivious implementations of the group-by aggregation were explored by Cieslewicz et al. [25], who showed that performance largely depends on input characteristics (key cardinality). This work examined aggregation in a multi-core environment on the Sun UltraSPARC T1, a chip multiprocessor.

We now discuss the state-of-the-art multithreaded CPU aggregation algorithms. A typical aggregation operator has three phases. The first phase is the startup phase that initializes all data structures. The second phase is the computation phase in which the input data is consumed and used to update the data structures. In the third phase data structure elements are merged to generate a single final result. For example, if each thread has its own private hash table, then those tables must be merged during this stage.

(i) Sort-Based Aggregation: Sort-based aggregation [25] has higher complexity *(O(nlogn))* than hash-based aggregation. The complexity of the hash-based method is linear in the number of records n. Additionally, materializing sorted input records is an I/O intensive operation. It should be noted that sorting is a blocking operator. On the contrary, hash-based approach can be easily pipelined from other database operators. The only time that sort-based aggregation is likely to be competitive is when the input stream is already in sorted order.

(ii) Independent Hash Tables: In this approach [25] each thread is given its own hash table that prevents memory collisions. As a result, the aggregation operator can avoid expensive synchronization primitives. The obvious disadvantage of this approach is memory consumption.

(iii) Shared Table with Locking or atomic synchronization: [25] 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 bet-

ter on low key cardinalities, and dont have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments. To simplify processing, locks are provide for each bucket, rather than for each cell. Locks are not required during the search phase in which the input records key is compared against keys in the bucket. Once a match is found, the bucket is locked to guarantee the atomicity of the update to the aggregate value. Locking is also needed when inserting a new element into a hash bucket.

(iv) Hybrid Aggregation [25] 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 processors 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.

(v) Partition & Aggregate [90] (also known as count then-move [26]) 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.

(vi) PLAT (Partitioning with Local Aggregation Table) [90] 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.

Follow up work [26] explored the partitioning step of hash aggregation and concluded that the thread coordination is a key component influencing the performance of this step. Finally, Ye et al. [90] proposed hybrid algorithms and showed that they outperform pure hardware-conscious and -oblivious implementations.

An FPGA-accelerated implementation of group-by aggregation was first considered by Mueller et al. [61]. This work 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 [13]).

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. 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 [91]. This drawback makes implementing large CAMs on reconfigurable fabrics notoriously hard. Dhawan et al. [34] explored various designs of CAMs and introduced a trade-off between CAM size and update time.

2.3.2 Selection Scan Operator

There are three commonly used selection evaluation algorithms. In this discussion, we assume that selection operates directly on the input array of records and writes qualified tuples into a new output array. Listing 2.1 presents the most straightforward way of implementing selection. It shows the *branching scan* method for a conjunctive query using '<' comparison. This technique is often called a 'short-circuit evaluation' because the computation of further predicates can be skipped when the first predicate is already evaluated to false. The logical-AND (&&) operator is typically compiled into k conditional branch instructions.

```
for (i=0; i < number_of_tuples; i++)
  if((t_i[0] < v_1) && (t_i[1] < v_2) && ... (t_i[k] < v_k))
  { out[j++] = i; }
```

Listing 2.1: Algorithm - Branching Scan

Assuming that the predicates have increasing selectivity, this method is optimal in terms of processing cycles. However, it was shown that on CPUs it leads to heavy branch mispredictions, causing considerable performance penalties [68, 22]. An alternative implementation, presented in Listing 2.2, uses bitwise-AND (&) instead of logical-AND (&&). This approach reduces k conditionals to a single branch.

```
for (i=0; i < number_of_tuples; i++)
  if(t_i[0] < v_1 & t_i[1] < v_2 & ... t_i[k] < v_k )
  { out[j++] = i; }
```

Listing 2.2: Algorithm - Predicate Scan with Bitwise-AND (&)

Here all predicates for a given tuple, t_i, are evaluated and then, depending upon the result of the evaluation, a branch is executed. This method reduces branch misprediction penalties at the cost of higher computational work. Finally, a no-branch implementation [68] is shown in Listing 2.3. This approach completely eliminates penalties caused by branch mispredictions by increasing computation costs.

```
for (i=0; i < number_of_tuples; i++)
  out[j] = i;
  j += (t_i[0] < v_1 & t_i[1] < v_2 & ... t_i[k] < v_k )
```

Listing 2.3: Algorithm - No-Branch

The techniques presented in Listing 2.2 and 2.3 either reduce or completely eliminate the branches. Yet, both approaches process all predicates and miss the opportunity to skip irrelevant evaluation as in the *branching scan*.

Selection on Parallel Architectures

An efficient implementation of Listings 2.1-2.3 must be designed for a specific data layout to effectively leverage the extensive cache hierarchy. Many efficient algorithms have been proposed for both row [68] and column [66, 96, 36] storage layouts. Moreover, columnar format allows effective compression and increases the throughput by more than an order of magnitude over traditional CPU row-store database systems [36]. However, all these approaches require an overhead of converting data in row-major format to a columnar layout either in the background or keeping two separate representations of the same record.

In addition to thread-parallelism, modern architectures support data-level parallelism and provide new opportunities for *vectorized* implementations. All modern CPUs are equipped with SIMD registers that can be used to accelerate many database operations [94, 84], including selection. All three algorithms described above can be vectorized. However, algorithm in Listing 2.1 requires additional bookkeeping to track which row should be dropped from future evaluation. This can be achieved by using non-contiguous loads/store (scatter/gather) operations[67] available only in the latest processors supporting AVX2 vectorization which make the design less portable. However, *branching scan* still can be implemented using older SIMD intrinsics but it will be sensitive to small predicate selectivity.

Likewise, GPUs implement *no-branch* algorithm because conditional execution (Listings 2.1 and 2.2) results in thread divergence and reduces the overall system performance. Selection is executed in two steps: (1) evaluating the predicate conditions and (2) gathering the indices of the qualifying rows [46]. During processing, all threads within a warp execute in lock-step irrespective of the predicate selectivity. However, being oblivious

to predicate selectivity can lead to wasted memory bandwidth since it does not provide any opportunities to stop fetching new predicates based on the last evaluation (Listing 2.1).

For low selectivity, GPUs may overcome this limitation by relying on indexes [46], thus reducing the overall tuple evaluation latency at the expense of processing throughput. Overall, maintaining an index can be costly when frequent updates are expected thus in this work, we focus on throughput optimization thus avoiding the use of indexes.

Selection on FPGA

The work presented in [75] focused on row decompression and predicate evaluation. In this technique, pre-generated bit files are produced (for different types of queries and for various hardware configurations) and the one that is best-matched to the given workload is loaded. Stream processing applications typically processes long-running queries which justifies invoking a time-consuming synthesis process for every new query.

In contrast to stream processing, in a data warehouse scenario, the query workload is unpredictable and queries are not always long-running. Thus, it is important to avoid time consuming synthesis process in such cases. This problem was addressed in [31] by applying a special FPGA technique called *partial reconfiguration*, which allows them to build query plans from pre-compiled components at runtime. It is a useful technique to time-multiplex circuits that do not fit on the same chip. Another work in [86], employs an FPGA between a DB system (MySQL) and SSDs to offload filter and aggregation queries. A complex WHERE clause is broken into individual base predicates and compared to the fixed size constant. The result is stored in a truth table which is transmitted to a BRAM. However, the memory consumption grows exponentially with the number of predicates. Our

MTP selection engine also maintains the query in a BRAM; however, the on-chip memory consumption is linear to the number of predicates in the query. Finally, [23] explores FPGAs as accelerators for join operations in columnar main-memory databases. Recently, a custom hardware multithreading design was applied to hash-joins [43].

2.4 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) is a highly exercised scientific kernel that solves y = Ax, where x and y are dense vectors, while A is a sparse matrix. (SpMV) has received significant attention due to its increasingly important applications in scientific and commercial applications. Although SpMV is a highly parallelizable, the real world sparse matrices often restrict realizable parallelism. It is mostly treated as a memory bound application [28]. Therefore, its performance on platforms like CPUs and GPUs depends on the memory bandwidth or the amount of cache on the respective platform. Our target architecture (the Convey HC2-ex) provides a peak bandwidth of 19GB/s memory bandwidth per FPGA, whereas current CPUs have 100 GB/s and current GPUs have 870 GB/s of memory bandwidth. However, it's a bit of a simplified justification of the problem.

There is a small niche where FPGAs can excel. When the matrix and vector sizes become large, around 10 million values [78], CPU performance drastically decreases. When this happens the CPU experiences a lot of x vector cache misses. To address this issue many turn to GPUs. GPUs are well known to be sensitive to sparsity and therefore to achieve better performance on SpMV, they expand the storage size of the matrix. This means that matrices that surpasses the GPU RAM size perform badly due to the transfer overheads.

It has been shown time and again that an *improper* sparse matrix format can degrade performance by an order of magnitude or more [16]. Therefore, to avoid excessive waste in bandwidth and storage, the sparse matrix is typically encoded using a sparse format, which stores only the non-zero values of the matrix along with meta-data that identifies a non-zeros location in the matrix.

2.4.1 Sparse Matrix Formats

Using the right sparse matrix format is essential because at runtime, the metadata must be decoded for each non-zero, which is used to identify the corresponding x vector value for calculating the dot product. Many different approaches have been proposed throughout the years, and we highlight the most general ones here. As an example, consider the nonsymmetric matrix defined by

$$\begin{bmatrix} a & 0 & 0 & 0 & b & 0 \\ x & y & 0 & 0 & 0 & z \\ 0 & l & m & 0 & 0 & 0 \\ p & 0 & q & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.4: Example matrix

Coordinate Format

The simplest sparse format is the Coordinate format (COO). This format stores the row index, column index along with the non-zero values. It is simple and for any pattern of sparsity the storage required is dependent on the number of non-zero values. It is implemented with three 1-dimensional arrays, one to store the non-zero values and the other two to hold the row and column index of the corresponding nonzero values.

Even though it is not quite as efficient from the memory requirement point of view, it is attractive because of its simplicity. In particular, the sparse matrix-vector product routine in the CMSSL library adopts a slight variant of the COO format.

row index	0	0	1	1	1	2	2	3	3
col index	0	4	0	1	5	1	2	0	2
matrix value	a	b	x	y	z	l	m	p	q

Figure 2.5: Coordinate Sparse Matrix Format

Compressed Storage Format

Compressed Storage Format (CSR) is the most popular and general format that provides excellent compression for both structured and unstructured sparse matrices in high performance architecture. SpMV with CSR format shows good performance improvement when implemented on CPUs and all algorithms like BLAS, LAPACK and CUSparse supports this format only. It uses three 1-dimensional arrays, one to hold the non-zero values, the second one holds the number of non-zero values per row and the third one holds the column index of the non-zero values. The CSR format of the example matrix is as follows: The compressed sparse column format (CSC) is almost identical to CSR, but instead of

row pointer	0	2	5	7	9				
col index	0	4	0	1	5	1	2	0	2
matrix value	a	b	x	y	z	l	m	p	q

Figure 2.6: Compressed Sparse Row (CSR) Matrix Format

compressing the row array it compresses the column array. The CSC format is less widely used because its workloads are harder to distribute evenly. SpMV has one output per row (not per column), and with CSR each output can be computed within a single process-

a	b	0
x	y	z
l	m	0
p	1	0

(a) Value Vector

0	4	*
0	1	5
1	2	*
0	2	*

(b) Column Vector

Figure 2.7: ELLPACK sparse matrix format

ing element (PE). CSC would require synchronization across multiple PEs to prevent race conditions.

ELLPACK Format

The ELLPACK (ELL) format rose in popularity with GPUs. Peak performance required each processor to run the same workload. The traditional ELL format compresses the matrix data into a rectangular dense matrix, and adds zeros to force each column in the matrix to have the same number of elements. The matrix data is then stored column by column, followed by their column indices in the original sparse matrix. Since each column is aligned to a 32-element boundary, the format is able to fully utilize the mechanism of the memory coalescing on a GPU. This leads to a significant increase in the loading throughput and improved performance of the SpMV for some matrices. The format handles matrices with similar short row lengths well because of the local column storage. However, if the row length for a matrix varies significantly and many zeros are needed, the ELL format becomes inefficient. A major challenge in computing y = Ax is the irregularity of accesses to the x vector (for row-wise traversal) or y vector (for column-wise traversal) as a result of the non-zero access patterns in the matrix. Hardware accelerators, such as FPGAs and GPUs, deal differently with this situation.

2.4.2 GPU Approaches

GPGPUs are known to be throughput-oriented processors. Therefore harnessing the best performance out of them requires exposing substantial fine-grained parallelism as well as sufficient regularity on execution paths and memory access patterns.

It is important to note that GPGPUs are very sensitive to the sparse matrix format. The format and resulting metadata can impact the load balance, resource utilization, and memory efficiency. One of the pioneering work on GPU-based SpMV, Bell and Garland [16] focused on different data storage formats, including CSR, diagonal (DIA), coordinate and ELL, and the design of parallel kernels operating efficiently on the corresponding formats. Bell and Garland implemented two parallel kernels: CSR-Scalar and CSR-Vector. In CSR-Scalar based approach, matrix rows are statically distributed over CUDA threads with each thread processing one row. This kernel takes advantage of the thread-level fine-grained parallelism. However, it still suffers from uncoalesced memory access within a warp. Furthermore, if consecutive rows assigned to a warp have different row lengths (the row length is defined as the number of non-zeros in a row), all of the other threads within a warp have to keep idle until the threads with the longest rows have completed. CSR-Vector, on the other hand, statically distributes matrix rows over a fixed number of warps in a round-robin fashion. However, this kernel can cause underutilization of hardware resources for short rows of lengths less than the warp size, resulting in low GPU occupancy. Load imbalance is a major cause of poor performance on GPUs. The CUSP [27] library handles this issue by applying a method that first computes the average row length in the whole matrix and then determines the vector size per row based on this average value. Similarly,

recent work presented in [58] target the same problem of load imbalance by introducing various heuristics.

2.4.3 FPGA Approaches

A number of FPGA SpMV accelerators have been proposed and demonstrate the potential of FPGAs as an SpMV acceleration substrate. However, many of these works focus either on developing efficient floating-point accumulators or removing all irregularity, by caching input or output vectors on on-chip memory or a compiler support for generating the corresponding hardware.

A substantial amount of work has focused on the efficient floating-point Multiply Accumulate (MAc) unit. Floating-point multiplication takes around 12 cycles, and addition can take up to 27 cycles. Producing partial sums every cycle is a non-trivial job because this requires managing multiple states during the accumulation for varying row lengths. Adder tree structures with feedback loop are proposed in [76, 95]. However, the number of non-zeros must be greater than number of channels, otherwise, the design is underutilized. Work presented in [30, 72] statically assign the partial products to multiple processing engines. This design is limited to accumulating just two rows concurrently. To support more row, multiple MACC units are placed across FPGA. An accumulation reduction circuit supporting an arbitrary number of rows is proposed in [38]. This circuit can read a new value every cycle. However, all data from one row must enter the circuit before any data from another row enters. A control unit arbitrates the dataflow between the floating-point addition unit and temporary buffers. We use a variant of a reduction circuit presented in [38] in our implementation.

In this thesis, we restrict our review to existing work which directly interfaces DRAM. Obviously data compression techniques like [24] can further improve the performance of SpMV. However, in this work we focus on multithreading paradigm and its benefits on well known matrix representations. Most of the work on SpMV using FPGAs use a row-major traversal [52, 93, 49]. The work in [81] does column-major traversal primarily to avoid caching the x vector. The downside of this approach is that it requires a y vector cache. To prevent cache misses, a large cache is proposed to store partial sum. However, this necessarily means that larger matrices will see worse performance. So, this cache will not be particularly efficient for matrices with heights of 100,000 or more. A detailed qualitative study of various approaches is provided in Chapter 5.

Chapter 3

Multithreaded Group-by Aggregation

Aggregation is widely used in relational databases to group information, or to count the occurrence of various values. As shown in Listing 3.1, there is a single relation, 'R', whose tuples, 't', must first be read; then the hash table is probed to find a match using each tuple's key (i.e. if a group already exists for this key), introducing a data flow irregularity. After that, either the key is new and thus a new entry (group) in the hash table needs to be inserted, or a match for this tuple's key was found and an update to an existing entry in the hash table is performed to increase the entry's count (assume for simplicity the aggregation is a SUM). The mixed read-write nature of aggregation requires us to use explicit synchronization to ensure correctness. Using atomic operations is one option, but this approach severely impacts the performance.

```
for (t in R)
        find_linked_list()
        for (node in linked_list)
                if(t.key == node.key)
                        update()
        if (t not in linked_list)
                create_node()
                insert_node()
```

Listing 3.1: Pseudo Code for Group-by Aggregation

Moreover, aggregated tuples exhibit temporal locality. We propose a novel multithreaded aggregation implementation based on CAMs in [6]. The design leverages explicit synchronization combined with the cache-like properties of the CAM, more details in Section 3.1.

3.1 Using 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 CAMs 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).

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 the next section, we discuss how to use this approach for synchronization in the multithreading group-by aggregation algorithm.

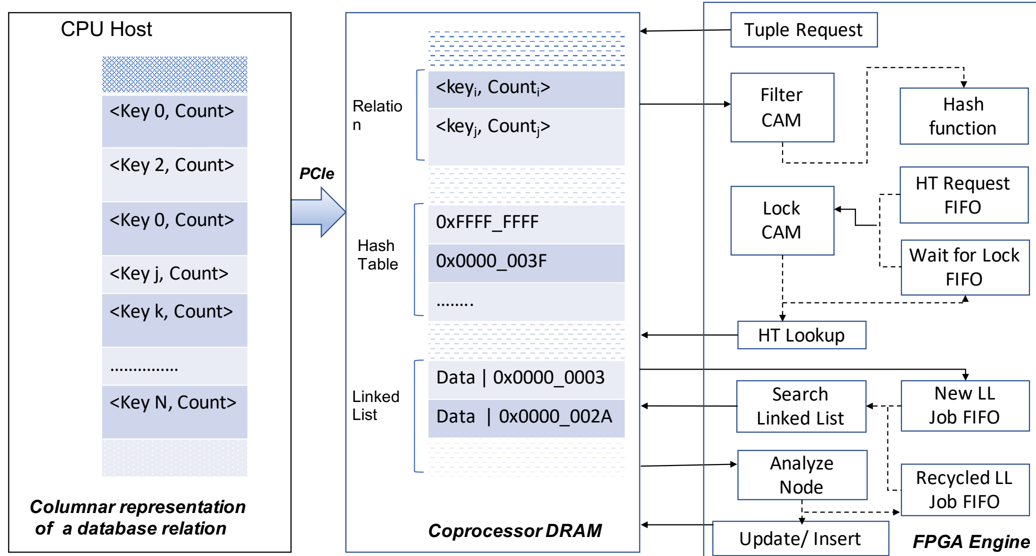


Figure 3.1: Engine Internal Blocks.

3.2 Aggregation Engine Workflow on FPGA

Figure 3.1[1] shows the layout, and memory channels of the aggregation engine. Each tuple from the relation is treated as a unique job, and is assigned its own thread on the FPGA. Jobs are first streamed from global memory by the Tuple Request component. Upon arrival they are sent to the first data CAM filter keys which combines duplicate keys into a single job on the FPGA.

The state diagram of a single thread inside the aggregation engine is shown in Figure 3.2. 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 hash table bucket

[1] Hardware design done with the help of Robert Halstead [45]

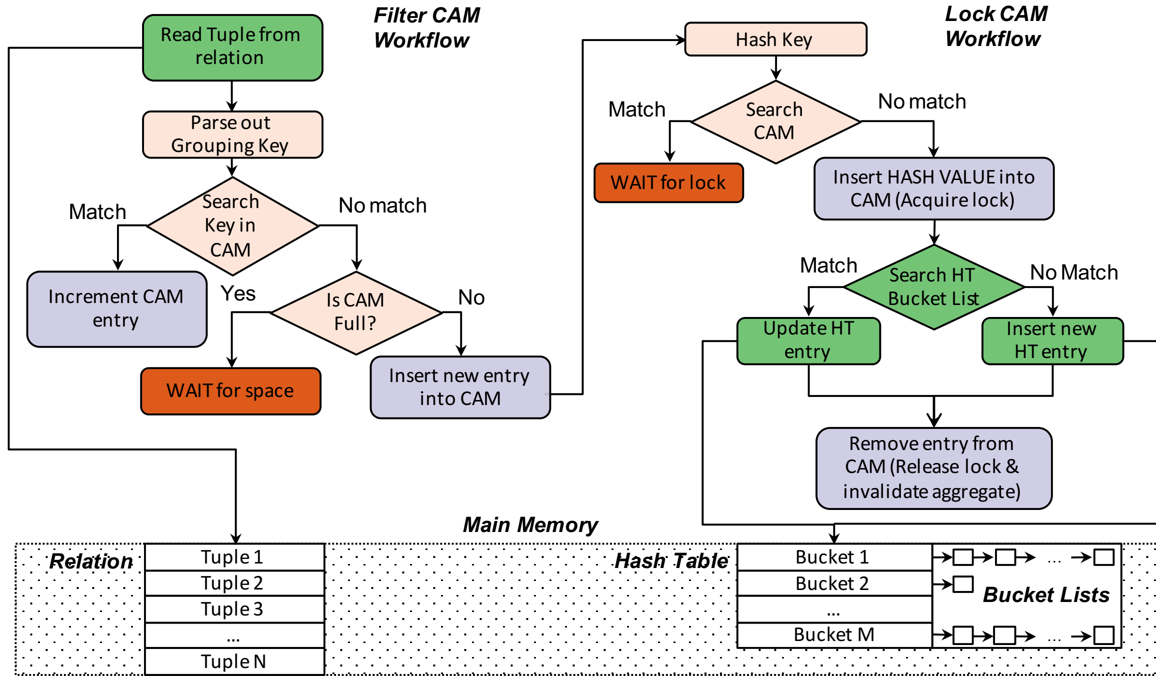


Figure 3.2: A state diagram for jobs in the aggregation engine.

Table 3.1 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 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 $Cycle_4$. *Lock CAM* maintains the locks for all buckets which are currently being searched or modified. In particular, after the job obtains a lock, it starts the bucket list search process and subsequently either updates an aggregate value or inserts a new entry into the bucket list for a certain key. Once a job completes, it invalidates the record in both CAMs, therefore frees up resources for other jobs. Jobs, waiting for a place

Cycle	Key	Filter CAM	Lock CAM	HashTable	Comments
1	A	*Miss, Insert (A,1)* {(A,1)}	*Miss, Insert hash(A)* {hash(A)}	{}	$Bucket_{hash(A)}$ is locked Request to search key A in HT is sent
2	C	*Miss, Insert (C,1)* {(A,1), (C,1)}	*Hit, since hash(A) =hash(C)* {hash(A)}	{}	*Job 2* waits for the lock
3	A	*Hit, Update (A,2)* {(A,2), (C,1)}	{hash(A)}	{}	*Job 3* is discarded
4		*Job 1 removes entry for key A* {(C,1)}	*Job 1 releases lock on hash(A)* {}	{(A,2)}	Key A was not found in HT. Create new entry (A,2) in HT
5		{(C,1)}	*Job 2 obtains lock on hash(C)*{hash(C)}	{(A,2)}	$Bucket_{hash(C)}$ is locked Request to search key C in HT is sent
6	B	*Miss, Insert (B,1)* {(B,1), (C,1)}	*Miss, Insert hash(B)* {hash(C), hash(B)}	{(A,2)}	$Bucket_{hash(B)}$ is locked Request to search key B in HT is sent
7		*Job 2 removes entry for key C* {(B,1)}	*Job 2 releases lock on hash(C)* {hash(B)}	{(C,1), (A,2)}	Key A was not found in HT. Create new entry (C,1) in HT
8	A	*Miss, Insert (A,1)* {(B,1), (A,1)}	*Miss, Insert hash(A)* {hash(A), hash(B)}	{(C,1), (A,2)}	$Bucket_{hash(A)}$ is locked Request to search key A in HT is sent
9		*Job 6 removes entry for key B* {(A,1)}	*Job 6 releases lock on hash(B)* {hash(B)}	{(B,1), (C,1), (A,2)}	Key B was not found in HT. Create new entry (B,1) in HT
10		*Job 8 removes entry for key A*{}	*Job 8 releases lock on hash(A)* {}	{(A,3), (B,1), (C,1)}	Key A was found in HT Update entry for the key A in HT to (A,3)

Table 3.1: 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 the hash bucket, holding the bucket list's head pointer

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*.

3.3 FPGA design and Optimizations

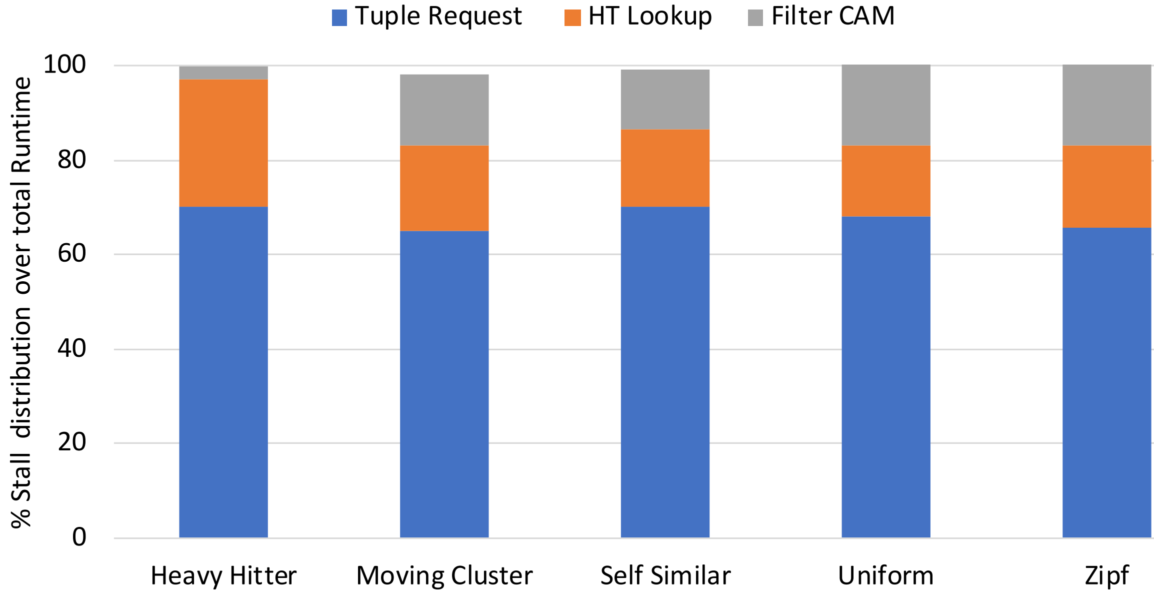


Figure 3.3: Stall percentage for Tuple Request, HT Lookup and Filter CAM modules for different datasets

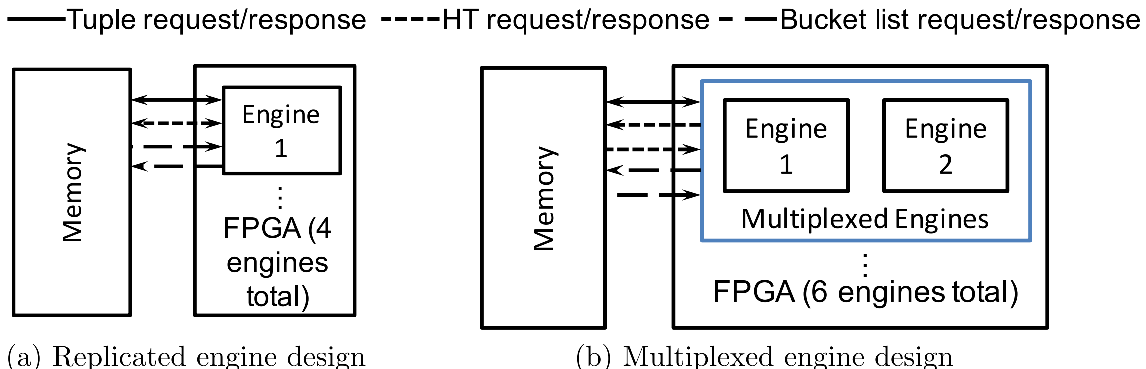
The main bottleneck of our design is memory bandwidth. In this paper we use a Convey-HC-2ex machine, but our designs are platform independent. In the Convey the communication between the FPGA and main memory relies on the abstraction called *channel*. Each channel supports independent and concurrent read/write accesses to memory. The initial design of our aggregation engine requires 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 replicate 4 engines ($\frac{16}{4}$) on a single FPGA thus leveraging inter-engine parallelism. Figure 3.4(a) demonstrates the design and channel assignment of the replicated engine approach. Each replicated engine uses its own CAM for synchronization. As a result, values are aggregated in separate hash tables. However, this requires an extra merging phase at

the end of the computation, an overhead which grows as we increase the number of engines per FPGA. In addition to inter-engine parallelism we also improve intra-engine channel usage.

We profiled our baseline implementation by adding counters to the main modules: *Tuple Request, HT Lookup and Filter CAM.* Figure 3.3 presents the distribution of stall cycles across different modules for varying datasets. Some observations are: (1) the *Filter CAM* module is the bottleneck across different datasets. This imply that as Filter CAM gets full, it stops fetching data from *Tuple Request.* This action causes FIFOs to become full and eventually stalls all outgoing memory requests. This behavior imply that the size of the Filter CAM is the bottleneck (2) As a consequence, memory channel connected to the *Tuple* request is overly underutilized, almost by 70%. 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. This clearly shows that memory channels can indeed fetch more data, however, the size of CAMs limit us to do so.

In order to increase memory utilization we have *multiplexed*[2] a pair of engines on a 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 previous design, the new multiplexed engine uses 5

[2]Main contribution of this thesis



(a) Replicated engine design (b) Multiplexed engine design

Figure 3.4: Alternative engine placement strategies on a single FPGA with 16 memory channels.

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. Figure 3.4(b) shows how engines are multiplexed on a single FPGA and depicts channel allocation in this design.

3.4 Evaluating Group-by Aggregation

The FPGA aggregation implementation is compared in terms of overall throughput against the best multi-core approaches [25, 90] running on a single processor with 4 parallel threads. We have already described our target architecture in Section 2.2.3. The subsequent section summarize various software aggregation algorithms as well a description of the datasets used in the experiments.

3.4.1 Software Implementations

In order to evaluate our FPGA-based solution we have implemented the following state-of-the-art multithreaded software aggregation algorithms[3]: (i) Independent Tables[25],

[3]Software aggregation algorithms are implemented by Ildar Absalyamov added here for the sake of comparisons.

(ii) Shared Table [25], (iii) Hybrid Aggregation [25], (iv) Partition with Local Aggregation Table [90] and (v) Partition & Aggregate [90]. Here, (i) and (ii) are considered as non-partitioned approaches, while (iii) and (iv) are hybrid, and (v) is a partitioned approach.

Dataset Description

We use five datasets with various s key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [25], Self Similar and Zipf_0.5.

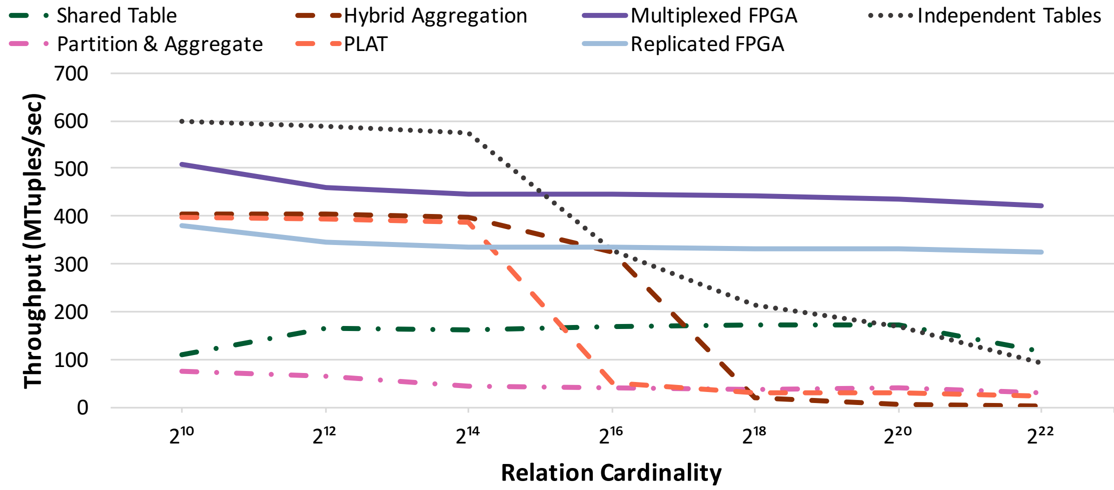
- In the **Uniform** dataset all key values are picked from *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.
- A half of the tuples in the **Heavy Hitter** dataset [25] share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the the entire relation.
- In the **Moving Cluster** dataset [25] 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. [41].

- In the **Zipf** dataset key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work[41].

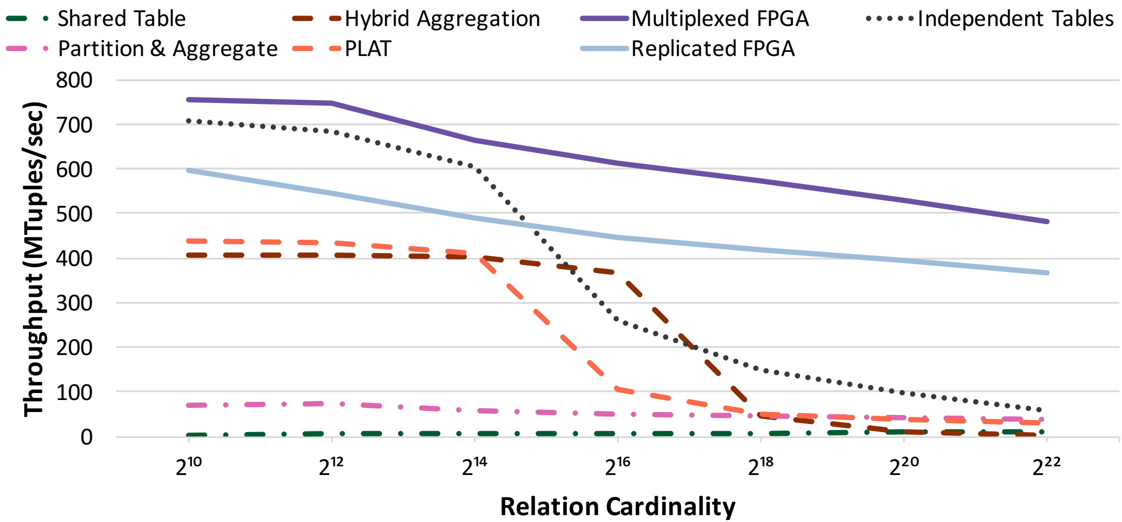
Each dataset consists of several benchmarks with cardinalities ranging from 2^{10} to 2^{22} unique keys. The relation size in all of the experiments was 256 million tuples (in line with previous research [90]). Each dataset used the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query processing algorithms [12, 18, 17] 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.

3.4.2 Throughput Evaluation

Figure 3.7 displays the throughput of the group-by aggregation as the key cardinality is increased, obtained for various datasets. Throughput was measured across two FPGA engine designs (regular and multiplexed), and five software (two non-partitioned, two hybrid and one partitioned) implementations. Throughput for skewed Heavy Hitter dataset Figure 3.6a resembles the results for Self Similar dataset Figure 3.5b, while the throughput for moderately skewed data Zipf_0.5 Figure 3.7a is similar to the results obtained for Uniform dataset Figure 3.5a. Software implementations demonstrate the best performance on Moving cluster dataset Figure 3.6b due to the property of the data distribution: similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.



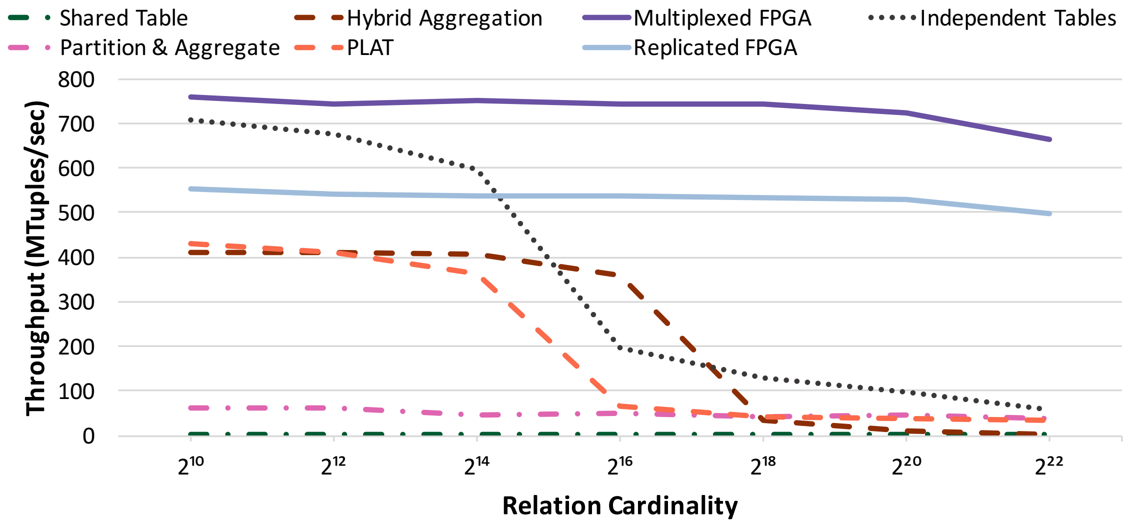
(a) Uniform



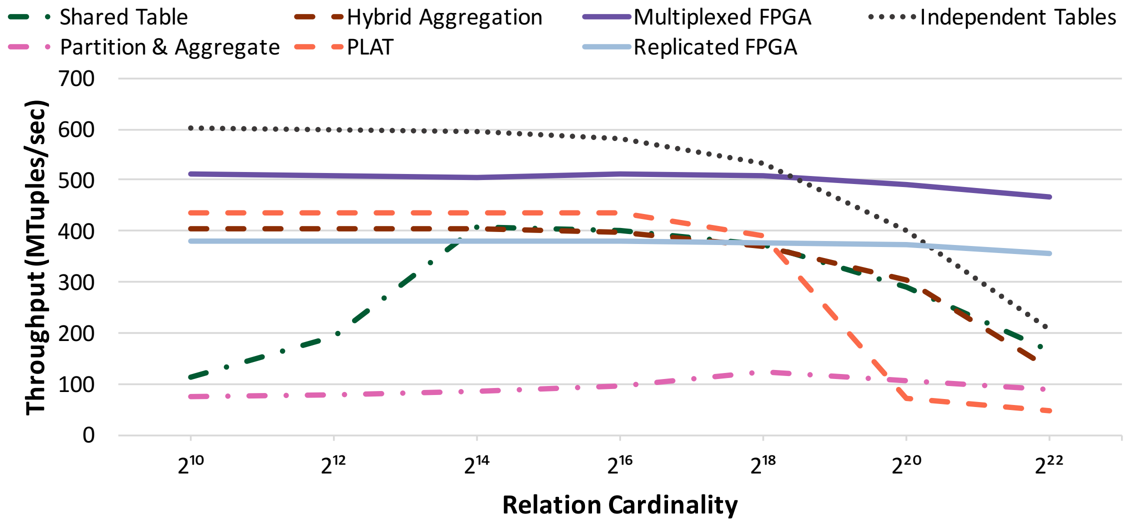
(b) Self Similar

Figure 3.5: Aggregation throughput of hardware and software approaches for datasets with 256M tuples.

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, 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

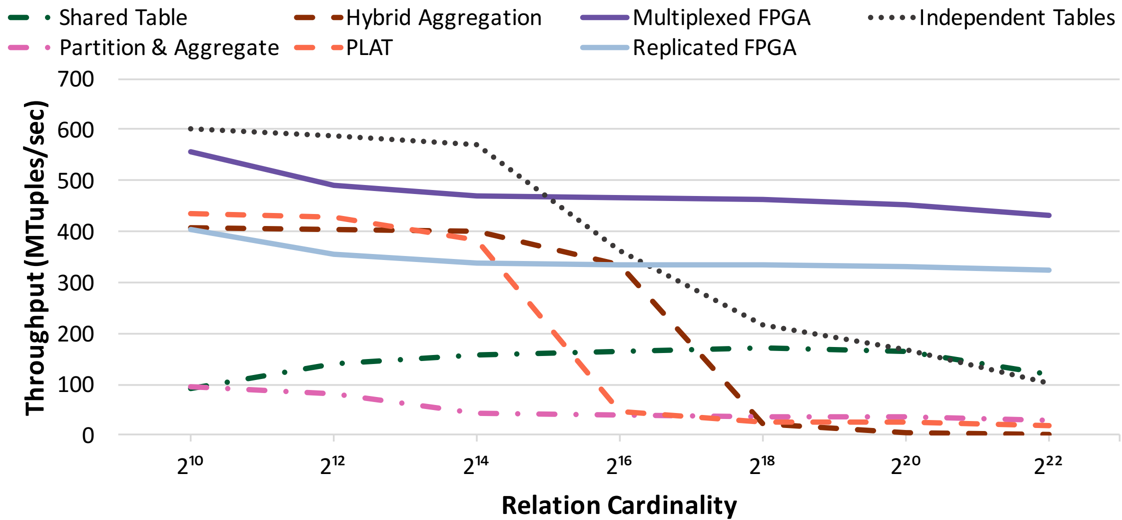


(a) Heavy Hitter



(b) Moving Cluster

performance severely deteriorates at cardinalities higher than 2^{18} on all datasets for all algorithms.



(a) Zipf

Figure 3.7: Aggregation throughput of hardware and software approaches for datasets with 256M tuples.

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.

3.4.3 Merge Overhead on FPGA

The Figure 3.8 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 450 MTuples/sec, even on very high cardinalities. The merge 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.

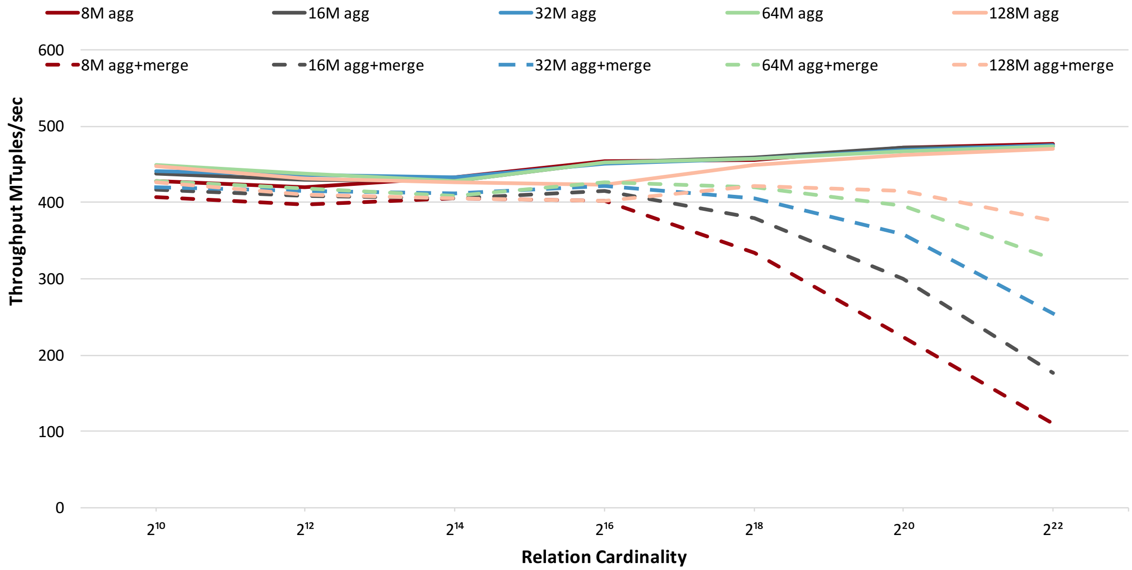


Figure 3.8: Effect of varying relation sizes on the FPGA aggregation throughput for datasets with Uniform key distribution. Solid lines represent throughput of the aggregation step (without merge operation), while dashed lines represent end-to-end (aggregation followed by the merge) throughput.

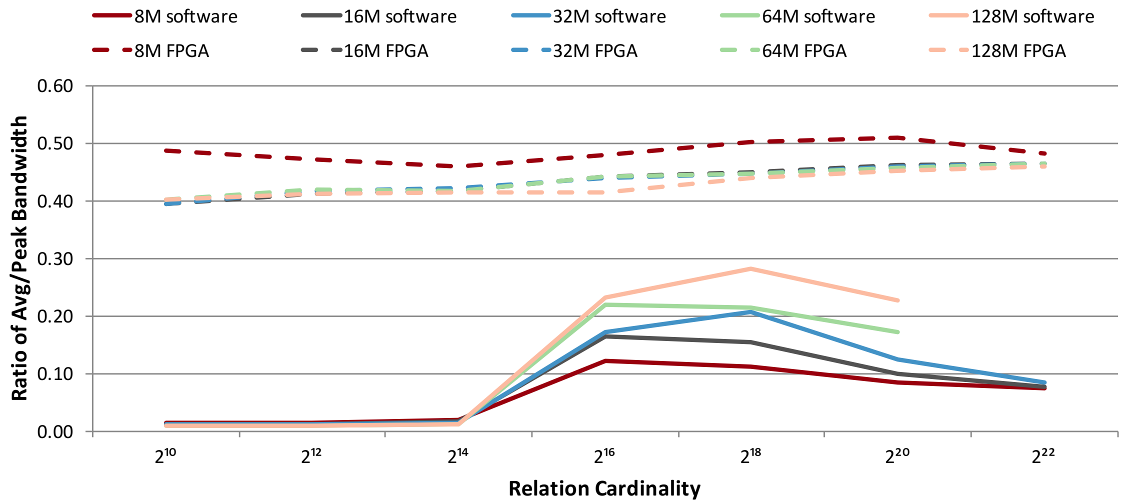


Figure 3.9: Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the Multiplexed FPGA design for varying dataset sizes and key cardinalities.

3.4.4 Performance Analysis

It should be noted that the performance benefits of the FPGA-based approaches come not from architecture-specific features, but from multithreading, which allows to utilize the available memory much better than any of the software implementations. Figure 3.9 depicts the ratio of effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and FPGA (multiplexed) implementations while varying dataset sizes and key cardinalities. Hardware mutithreading approach allows our FPGA implementation to keep the ratio almost constant, irrespectively 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 0.3 for cardinality 2^{18}, but drops significantly for higher key cardinalities. For very large cardinalities the FPGA implementation ratio is almost 5 times higher.

3.5 FPGA Area Utilization

Table 3.2 shows the resource utilization (registers, LUTs, and BRAMs used) for both FPGA aggregation designs (replicated and multiplexed) as the number of engines is scaled up. As we can see increasing the number of engines by one only adds an additional

Table 3.2: FPGA resource utilization for aggregation engines.

# of Engines	Registers	LUTs	BRAMs
1	99597 (11%)	87194 (18%)	126 (17%)
2	116635 (13%)	100497 (21%)	147 (19%)
3	135517 (15%)	115560 (24%)	184 (24%)
4	152132 (17%)	129775 (27%)	206 (28%)
1-Multiplexed	113695 (11%)	114280 (24%)	142 (19%)
2-Multiplexed	145690 (15%)	140684 (29%)	196 (27%)
3-Multiplexed	179641 (18%)	200175 (42%)	250 (34%)

2% for registers, 3% for LUTs, and 4% for BRAMs for replicated engine design. This happens because a lot of the components are shared across the engines. However as we start multiplexing the engines we stop sharing the resources due to timing constraints. This results in growth of FPGA resource utilization as we increase the number of engines.

The aggregation design utilizes a lot of LUTs, which are extensively used in our CAM implementation. The hardened BRAM blocks only have two channels. This property is too restrictive for the CAMs, which must access all locations in parallel. The aggregation design uses only 42% of the available resources

3.6 Conclusion

In this chapter we presented a multithreaded FPGA implementation of the group-by hash aggregation operation. All data structures are stored in main memory, which allows the DBMS to seamlessly transition between software and hardware execution. We introduce a portable approach which uses CAMs to provide fast caching and enforce synchronization. We explore various FPGA designs and apply optimizations to further improve the performance. Experimental results show that the aggregation throughput is consistent

and predictable regardless of a relations size and cardinality. Despite the fact that the final merge step does affect performance, we show that this overhead is amortized when the relation size increases. Experiments show that the multithreaded FPGA approach can significantly outperform all existing software approaches and demonstrate especially good performance for high cardinality benchmarks. Throughput ranges between 700 to 150 MTuples/sec depending on the dataset distribution and key cardinality, with a speedup up to 10x.

Chapter 4

Multithreaded Selection Operator

In this chapter, we focus on one commonly-used database operation, namely applying a conjunction of selection conditions to a set of database records. One wishes to obtain those records satisfying the conditions in as efficient a way as possible. A selection condition is usually formed by the number of predicates with the following structure $< COL$ comp_op $CONST >$ that appear in the WHERE clause of an SQL query. In a typical query plan selection operators appear early on, right above data scan (or an indexed-based access method) thus their performance directly affects the total runtime of the whole query. We also describe the design of a programmable hardware accelerated selection engine (hereafter referred to as MTP) for DBMS which deals with long memory latencies using hardware multithreading. MTP allows us to fetch only the needed parts of the record, thus maximizing the utilization of the available memory bandwidth. It is important that our approach works irrespective from physical database storage layout allowing to run efficient analytics on transactional row-oriented data without replicating it in columnar representation.

This technique restricts the number of memory accesses to the in-memory relation to be proportional to the size of the query (number of predicates in query),

4.1 MTP Selection Engine

For our MTP implementation we take advantage of the *early termination* provided by the *branching-scan* algorithm. We spawn thousands of lightweight threads (one for each tuple) which allows us to do fine-grained data access (fetching only the values needed for the query evaluation on each tuple) without incurring branch misprediction penalties and independently of the data layout.

4.1.1 Engine Design

labelsec:design In the rest of our paper, we assume that the input relation is too large to be stored in local FPGA BRAMs. Therefore, our design trades off small and fast on-chip memory for larger and slower off-chip memory. The *selection engine* is a custom datapath that copes with the long memory latencies by issuing thousands of threads and maintaining their states locally on the FGPA. Because of the inherent FPGA parallelism, multiple threads can be activated during the same cycle while other threads are issuing memory requests and going idle. The selection engine is capable of processing different selection queries **without** needing to re-configure the logic on the FPGA.

To program the selection engine at run-time for different queries, we arrange our queries in a standard *disjunctive normal form* or DNF. We build the DNF by parsing the query and creating a data structure, called the Predicate Control Block (PCB). The PCB

```
SELECT (*)
FROM OrderDetails
WHERE
(UnitPrice > 5  AND
Quantity > 10) OR (TotalPrice > 100)
```

(a) Example SQL Query

Column	Opr	Const	Col_True	Col_False
UnitPrice	>	5	*Quantity*	*TotalPrice*
Quantity	>	10	TRUE	*TotalPrice*
TotalPrice	>	100	TRUE	FALSE

(b) Corresponding Predicate Control Block.

Figure 4.1: Figure (a) represents the sample SQL query which is used as an example query in this section. Figure (b) present the corresponding *predicate control block* (PCB) for the sample query

size is linear to the number of predicates in a query as it maintains a single record for each predicate. Each record of the PCB stores the comparison operator (e.g. less than, equal, etc.), the constant, and two column offsets(*True or False*) that are needed to direct further evaluation.

Figure 4.1a shows a sample SQL query and its corresponding PCB. It has three rows that correspond to the three predicates in the query. We parameterize all three parts of our predicate: (1) the column ID, (2) the comparison operator, for which one out of six possibilities is selected (=, <>, <, >, <=, >=), and (3) a constant value. Given a tuple from *OrderDetails*, after evaluating the first predicate ($UnitPrice > 5$) in the query, the next column offset requested is *Quantity* if the condition evaluates to true. Otherwise, we request *TotalPrice*. Notice that if ($UnitPrice > 5$) is false, we continue immediately with *TotalPrice* instead of finishing the next comparison. This provides *early termination* within clauses as well as queries.

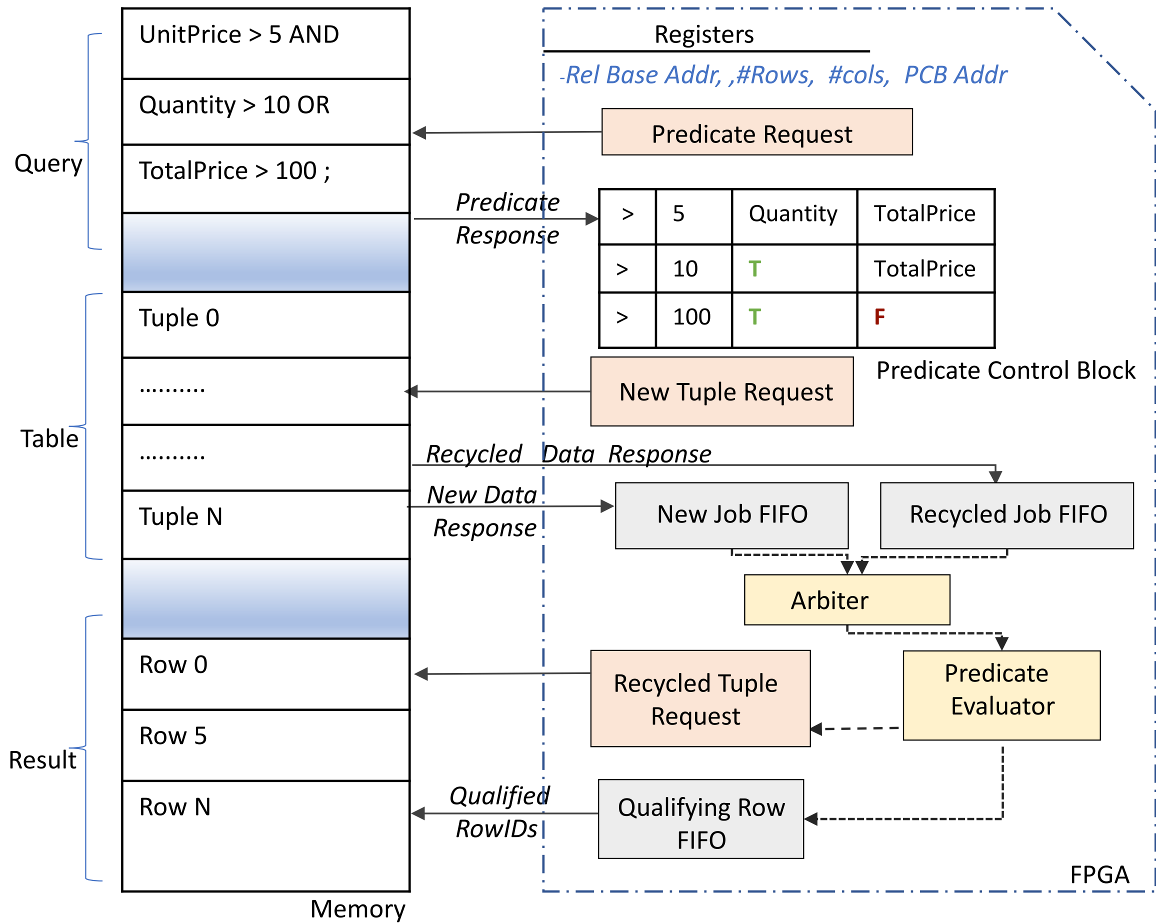


Figure 4.2: Selection Accelerator Engine: showing different building blocks and memory channels that read/write from memory

4.1.2 Engine Workflow

Figure 4.2 shows the design of our selection engine. Local registers are programmed at run-time and hold pointers to the database table and PCB. They also hold information about the number of tuples, row size (in terms of number of columns), and the starting column offset (column ID in the first predicate). Lastly, the registers hold the base address of the memory space where qualifying row offsets will be written back.

The execution starts with loading the PCB (*query*) from the memory. The *Predicate Request* module loads the PCB and stores it locally on the FPGA in BRAM. The query

evaluation starts when all the PCB entries have been received and stored. In the meantime, the *Tuple Request* module will create a thread for each tuple and starts issuing requests that corresponds to the data located at *(Row_i,Col_j)*. Requests are issued continuously until all tuples have been processed. When a thread issues a request, the tuple's pointer and the next predicate to be evaluated for this tuple are added to the thread state and the thread goes idle. Threads are issued in-order but qualifying rows can be written back to memory out-of-order.

As data requests are completed, the thread is activated again and a corresponding predicate yet-to-be evaluated is evicted from the stored thread state. Then the *Predicate Evaluator* either qualifies a row, disqualifies a row, or requests a new data for column *(Row_i,$Col_{j'}$)* to further evaluate a query. In the latter case the thread pointer for this row along with *column j'* find its place in *recycled requests.*

The *Arbiter* decides which active thread will issue the next request to memory. This request either brings in a new row or uses a recycled row. Priority is given to the recycled threads to reduce the number of concurrent jobs and ensure that the design will not deadlock by filling up the request queue with new row requests.

4.1.3 MTP Performance Analysis

Since the MTP selection engine implements the branching-scan algorithm, there is a strong correlation between the measured throughput and *selectivity (S)*, *predicate probability (p)*, and the *number of predicates (k)* of the selection query. S is defined as the fraction of records in the input relation that satisfy the given query conditions, while p is the probability of an individual predicate to be true. To simplify our analysis we assume

that there is no correlation between columns (attributes). We also group all the predicates on a single column into one; hence in the rest of the paper we assume that every predicate is applied on a different column. With this assumption, the proposed model can calculate the expected number of predicates evaluated for a given value of S and k and describe the variability in throughput based upon these predicate evaluations. Table 4.1 depicts the parameters used in this analysis.

Table 4.1: Query and input relation parameters used in the analytical model

S	query selectivity
p	predicate probability
k	number of predicates
N	total number of rows
W_e	work done to evaluate predicates
W_{wr}	work done to writeback the qualifying rows

Query evaluation involves reading column values from the memory and writing back the IDs of the rows that qualify. Therefore, the total work done to process a query can easily be partitioned into the amount of work done to evaluate the query, $\mathbf{W_e}$, and the work done to write back the ID of the qualifying rows, $\mathbf{W_{wr}}$. Note that W_e includes sending a request and receiving a response from the memory, evaluating the value of a column against a constant, and sending further requests for other predicates until a decision on a row can be made.

Clearly, W_e is proportional to the total number of predicates evaluated and W_{wr} is proportional to the total number of qualifying rows. Hence, the overall throughput of the

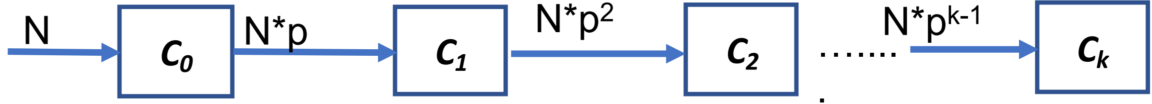


Figure 4.3: Number of qualified rows after each predicate for a conjunctive query. N is the total number of rows, p is a predicate probability, C_0,C_1,C_2,...,C_k designate k different predicates.

system can be defined as follows:

$$W_e \propto \text{Total number of evaluated predicates} \tag{4.1}$$

$$W_{wr} \propto \text{Total number of qualified rows} \tag{4.2}$$

$$\text{Throughput} \propto 1/(W_e + W_{wr}) \tag{4.3}$$

The rest of the section further elaborates upon W_e and W_{wr} on two types of queries, namely, a query that contains only conjunction of predicates and a query that consists only of disjunction of predicates. Finally we discuss the case of a mixed query (that contains combination of ANDs and ORs).

Conjunctive Queries. Figure 4.3 shows the expected number of qualifying rows after each predicate in a conjunctive query. Equation 4.4 calculates the expected number of *evaluated* predicates by summing up the number of rows reaching each predicate. Equation 4.7 defines the expected rows satisfying the last predicate condition based upon query selectivity. As a result, we obtain W_e and W_{wr} for conjunctive query in Equations 4.5 and 4.8 respectively.

$$\text{Exp. number of evaluated predicates} = N * \sum_{i=0}^{k-1} p^i \tag{4.4}$$

$$\text{From Equation 4.1, } W_e \propto N * \sum_{i=0}^{k-1} p^i \tag{4.5}$$

$$\text{Selectivity of a conjunctive query, } S_{and} = p^k \tag{4.6}$$

$$\text{Expected number of qualified rows} = N * S_{and} \tag{4.7}$$

$$\text{From Equation 4.2, } W_{wr} \propto N * S_{and} \tag{4.8}$$

Equation 4.6 shows that the selectivity, S, is a function of p and k. This relation is plotted in Figure 4.5a. An important observation is that as S and k increase, p increases logarithmically. Higher values of p imply that more predicates per row are evaluated, i.e. increasing W_e.

Disjunctive Queries. We perform the same analysis for a disjunctive query, shown in Figure 4.4 and obtain the following equations:

$$W_e \propto N * \sum_{i=0}^{k-1} (1-p)^i \tag{4.9}$$

$$S_{or} = 1 - (1-p)^k \tag{4.10}$$

$$W_{wr} \propto N * S_{or} \tag{4.11}$$

Once again we plot p with varying values of S and k in Figure 4.5b. Key observations from this graph are: (1) For a given k, p increases with S. Higher probabilities cause

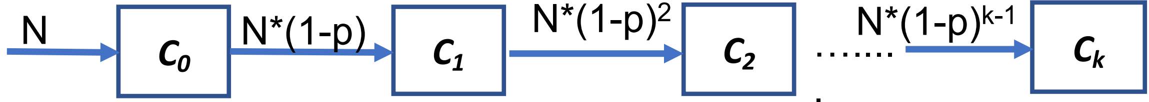


Figure 4.4: Number of qualified rows after each predicate for a disjunctive query. N is the total number of rows, p is a predicate probability, C_0,C_1,C_2,...,C_k designate k different predicates.

disjunctive queries to terminate early, hence W_e decreases. (2) Keeping S constant, if we increase k, probability p decreases, increasing W_e.

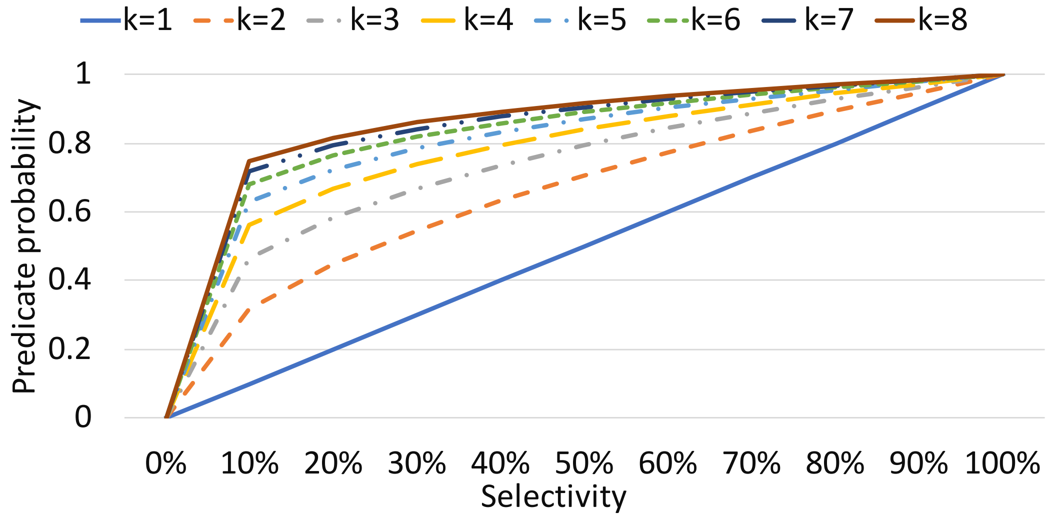
Also, note the opposing nature of probabilities for conjunctive and disjunctive queries in Figure 4.5. This visually illustrates Equations 4.4 and 4.9 showing that for any given selectivity, S, the number of predicates evaluated for both types of queries are complement to each other. We observe the same trend in our experiments. Table 4.2 summarizes the different trends in W_e and W_{wr} for both conjunctive and disjunctive queries.

Table 4.2: Summary of relationships between S, k, W_e and W_{wr}.

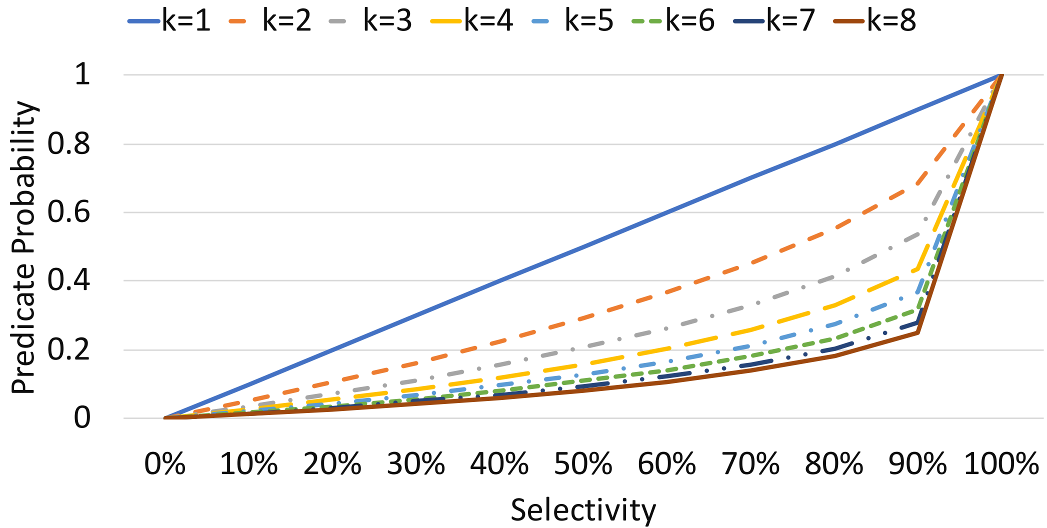
	Conjunctive		Disjunctive	
	S Const, $k\uparrow$	k Const, $S\uparrow$	S Const, $k\uparrow$	k Const, $S\uparrow$
W_e	linearly $\uparrow$	$f(p,k)\uparrow$	linearly $\uparrow$	$f(p,k)\downarrow$
W_{wr}	$\propto S$	linearly $\uparrow$	$\propto S$	linearly$\uparrow$

Mixed Queries. We can extend the same analysis to loosely bound the throughput of a query with different combinations of *AND-OR*. We again assume the mixed query arranged in the standard DNF form.

For any query with k predicates, the minimum possible number of evaluated predicates per row is 1 while the maximum is k. The maximum performance (minimum number of evaluated predicates) is achieved by a conjunctive query at $p = 0$ and a disjunctive query at $p = 1$. Similarly, the minimum performance of a conjunctive query is achieved at



(a) Predicate probability vs selectivity for a conjunctive query.



(b) Predicate probability vs selectivity for a disjunctive query.

Figure 4.5: Variance in predicate probability with respect to selectivity for conjunctive (a) and disjunctive(b) query.

$p = 1$ and at $p = 0$ for the disjunctive query. Therefore, for any given value of S and k, the performance of a mixed query will always be bounded by a maximum and minimum performance of a pure conjunctive and disjunctive query. For any query with k predicates, the minimum possible number of evaluated predicates per row is 1 while the maximum is k. The maximum performance (minimum number of evaluated predicates) is achieved by

a conjunctive query at $p = 0$ and a disjunctive query at $p = 1$. Similarly, the minimum performance of a conjunctive query is achieved at $p = 1$ and at $p = 0$ for the disjunctive query. Therefore, for any given value of S and k, the performance of a mixed query will always be bounded by a maximum and minimum performance of a pure conjunctive and disjunctive query.

4.2 Evaluating Selection Operator

We again choose Convey HC-2ex as our target platform. The details of this platform are described in Section 2.2.3. Table 4.3 summarizes the characteristics of all the platforms used. Figure 4.6 compares the absolute query runtime on the CPU, GPU, and MTP implementations.

Device	Make & Model	Clock, MHz	Cores	Memory Size, GB	Memory Bandwidth, GB/s
CPU	Intel Xeon E5-2643	3300	8	128	51.2
GPU	NVIDIA Titan X	1500	3584	12	480
FPGA	Virtex6-760	150	N/A	64	76.8

Table 4.3: System Configuration of different architectures used for evaluation.

4.2.1 Throughput Evaluation

The GPU delivers the best raw performance across different predicate values and selectivities, followed by MTP that performs better on low selectivity values and smaller

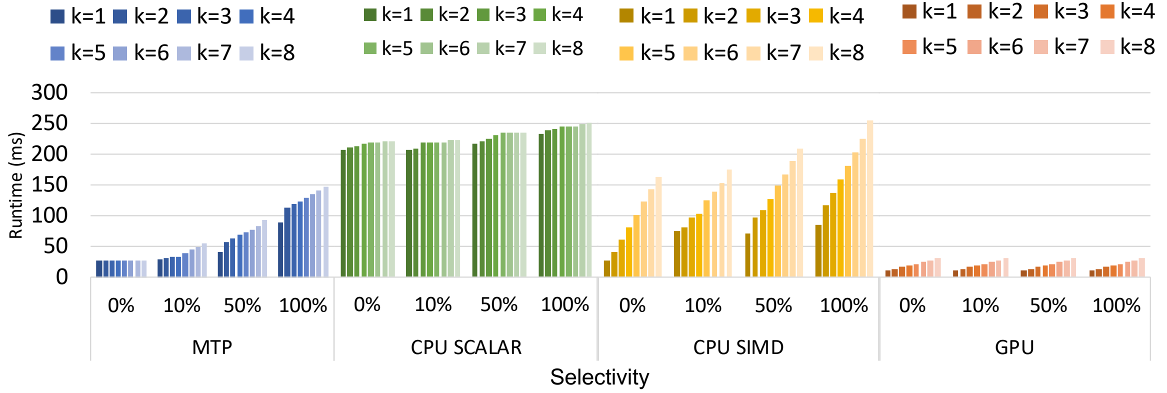


Figure 4.6: Query evaluation runtime measured on MTP, CPU and GPU with varying selectivity and number of predicates

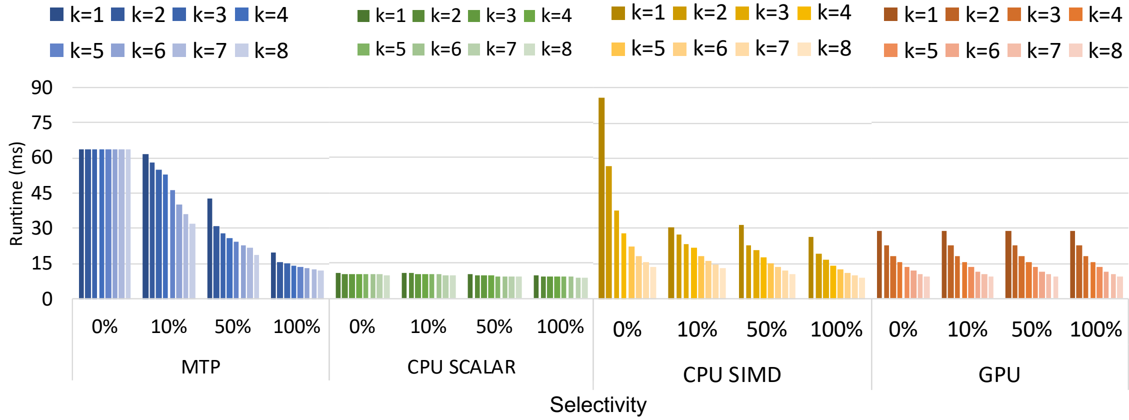


Figure 4.7: Throughput achieved by MTP, CPU and GPU implementation normalized to their respective bandwidth. Note that the legend description is same as that of Figure 4.6.

values of predicates. The CPU SIMD implementation comes next and finally the CPU Scalar implementation achieves the highest runtime among all other architectures.

Adhering to the branching-scan characteristics, the performance of MTP is sensitive to predicate probability. The predicate probability of a conjunctive query increases with the selectivity (S) as well as with the number of predicates (k) as it can be seen in Figure 4.3. As a consequence, the query evaluation can terminate early on the lower value of selectivities but builds up for higher values. Additionally, for high values of S, the writeback work(W_{wr}) increases too, resulting in a quick drop in throughput.

Furthermore, in our experiments we define S using p and k assuming that all predicates have the same probability. On the other hand, real-world queries might have a different combination of predicate probabilities for the same total selectivity. If a query optimizer performs a good job of arranging predicates in the order of their likelihood of being false (true) for conjunctive (disjunctive) queries, MTP will work independently of the number of predicates in a query and is only limited by the number of memory accesses required for query evaluation. This can be seen in Figure 4.6, where the MTP runtime does not change with the number of predicates for 0% selectivity.

Unlike the MTP implementation which is based on early termination, all other platforms implement a variant of the *No-Branch* algorithm. The CPU Scalar graphs clearly show that its execution time is independent both from the number of predicates and from the query selectivity. This independence is an expected behavior because in a row-major storage format selection is a memory-bounded computation. Each access to a particular tuple will bring from memory the values for *all* its columns, whether they will be evaluated later or not.

On the other hand, the runtime of the CPU SIMD implementation grows linearly as we increase the number of predicates in a query from 1 to 8. Again this behavior is explained by the fact that in a columnar storage format we are fetching only the values that will be later used for evaluating the predicate. For the queries with 8 predicates, the runtimes of Scalar and SIMD converge because they perform the same amount of memory accesses. However we can also see another trend for the vectorized implementation: its runtime grows as we move from 0% selectivity to 100%. This is explained by the increasing

amount of qualifying recordIDs which need to be written to the output buffer. The SIMD implementation is susceptible to growing W_{wr} because it requires additional permutations in order to retrieve IDs of the rows that were qualified from a SIMD lane.

We should also note that for the CPU implementations, the runtime is a function of the main memory bandwidth utilization, not the penalty of fetching data into CPU cache. In both experiments, the data access patterns (contiguous load for Scalar or load with constant strides for SIMD) are easily recognized by the CPU prefetcher, which was verified by preliminary experiments where we had disabled the prefetcher.

Similarly, the GPU implementation evaluates all predicates despite their different selectivities, resulting in more evaluation work for the respective query. This translates to a higher number of memory fetches that quickly dominate the total execution time, as their cost is several magnitudes higher than that of evaluating the predicate conditions. Therefore, an increase in the number of predicates corresponds to increasing runtime as indicated by our experimental results.

4.2.2 Throughput Efficiency

To better capture the memory-bounded nature of the selection and provide a direct comparison between widely different architectures we normalize the MTP, CPU, and GPU throughput to the memory bandwidth available on each architecture. As discussed in Section 4.2 the Convey HC-2ex has 4 FPGAs with cumulative bandwidth of 76.8 GB/s, the CPU system has a memory bandwidth of 51.2 GB/s, and the GPU system has a memory bandwidth of 480 GB/s. The normalized results are shown in Figure 4.7.

The CPU SIMD implementation is remarkably efficient for $k = 1$ and $S = 0\%$. Since only *one* predicate is evaluated, the columnar data layout makes the cache access extremely effective in this case.

Moreover, with 0% selectivity there is no result materialization overhead. However, the CPU SIMD throughput drops quickly as S and, especially k are increased.

In contrast, since the MTP design is only susceptible to the predicate probability, it takes better advantage of early termination on lower selectivity values and processes more tuples/sec per bandwidth. It can be seen from Figure 4.7 that for $k > 1$ and $S = 0\%$, MTP is 1.2x - 5x more bandwidth efficient. However, as selectivity increases, we start seeing the effect of the writeback pressure on the MTP design too. For instance, for 50% selectivity, an effective speedup of only 1.2x - 1.7x is achieved over the CPU SIMD implementation. Overall the MTP design remains 1.6x - 4.7x more efficient in comparison to the CPU Scalar implementation within the wide range of selectivities ($0\% \leq S \leq 50\%$).

A similar trend is also observed while comparing to GPU. For lower selectivity values, say, $S = 0\%$ the MTP design is 1.8x - 5x more efficient than GPU. However, as we increase the selectivity, the GPU throughput remains unaffected while the MTP sees a performance drop to 1.5x - 1.8x for $S = 50\%$.

Finally, for $S = 100\%$, the MTP throughput is similar to the GPU, CPU SIMD and has a 2x edge over the CPU Scalar. At this selectivity, the probability of each predicate is also 100%. For a conjunctive query, this leads to more predicate evaluations, resulting in the highest value of W_e. Additionally, with the maximum value of selectivity, writeback

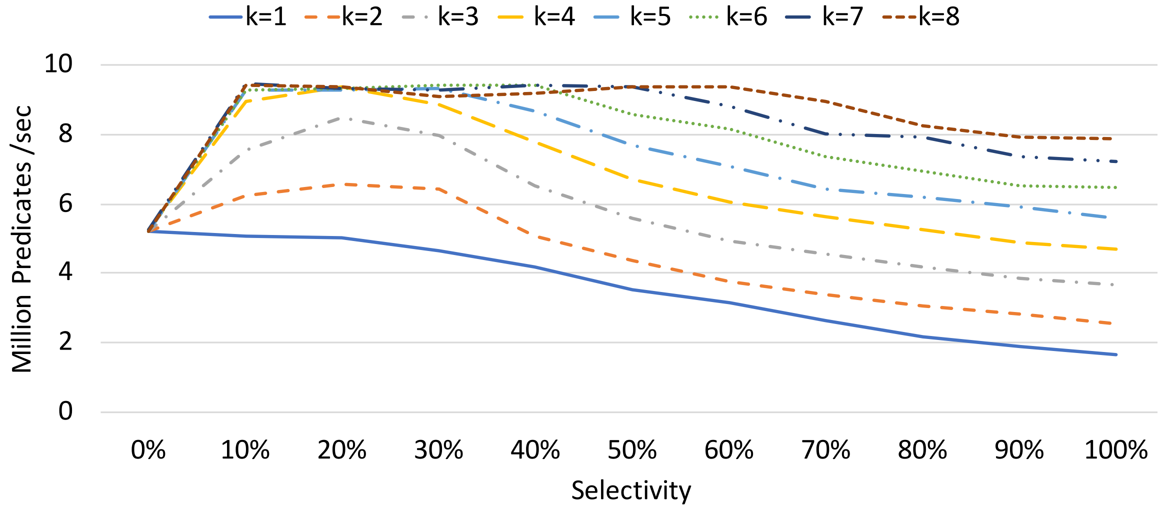


Figure 4.8: Absolute attributes evaluated as the selectivity and number of predicates are varied for (a) conjunctive

work is at its peak. Therefore, at this stage, the MTP design sees diminishing returns from the advantage of early termination as writeback becomes the bottleneck.

4.2.3 Multithreaded Execution of Selection

As discussed in Chapter 2.2.2 there are three states of multithreaded execution: (1) build-up state (2) steady state and the (3) drain state. In the context of selection operator, the build up state consists of requesting the attribute required for evaluating the first predicate each row. In a steady state, *Predicate Evaluator* module (Figure 4.2) always have an attribute to evaluate. This attribute can belong to a new row or the recycled row. Finally, as more and more rows qualify/disqualify, number of threads start decreasing, leading to insufficient in-flight requests that can no longer mask the memory latency. Note that the unit of work is predicate evaluation per row and therefore we measure the absolute

number of predicates evaluated per second to discuss multithreaded nature of selection operator

Figure 4.8 describes the aforementioned behavior exhibited by conjunctive queries. At $k = 1$ only one predicate is requested from memory which is a part of the build-up state. The steady state is non-existent in this case. As requests are fulfilled, predicates are evaluated and the qualifying row ids are written back to memory, launching the drain state, a long *tail* . At this point, the computational efficiency of the selection engine is low. Additionally, because of the write channel bottleneck, the predicates/sec drops further. However, as k increases, W_e increases, and now we can observe a steady state in the form of a *plateau* region. This region signifies that the PU is busy. Also notice that with higher value of k, the length of the plateau increases and the tail reduces.

4.2.4 TPC-H Query Evaluation

To evaluate the performance of our implementations on a standard workload we considered the well-known TPC-H benchmark [5]. We have profiled all 22 TPC-H queries to understand the various characteristics (selectivity, number of predicates, predicate types) of the selection operator in this benchmark. Most queries in the TPC-H workload involve complex joins and group-by aggregations, so not all predicates in the WHERE clause might be used as filtering conditions in selection operators. Instead, we have considered optimized plans where selections are pushed down and executed before the joins and right after table scan operators. Figure 4.9 presents the selectivity(%) of different TPC-H queries. In this experiment, the average number of predicates in the selection was 2, with an exception of queries Q_6 and Q_{19} which have 5 and 8/12 (Part/Lineitem tables) predicates respectively.

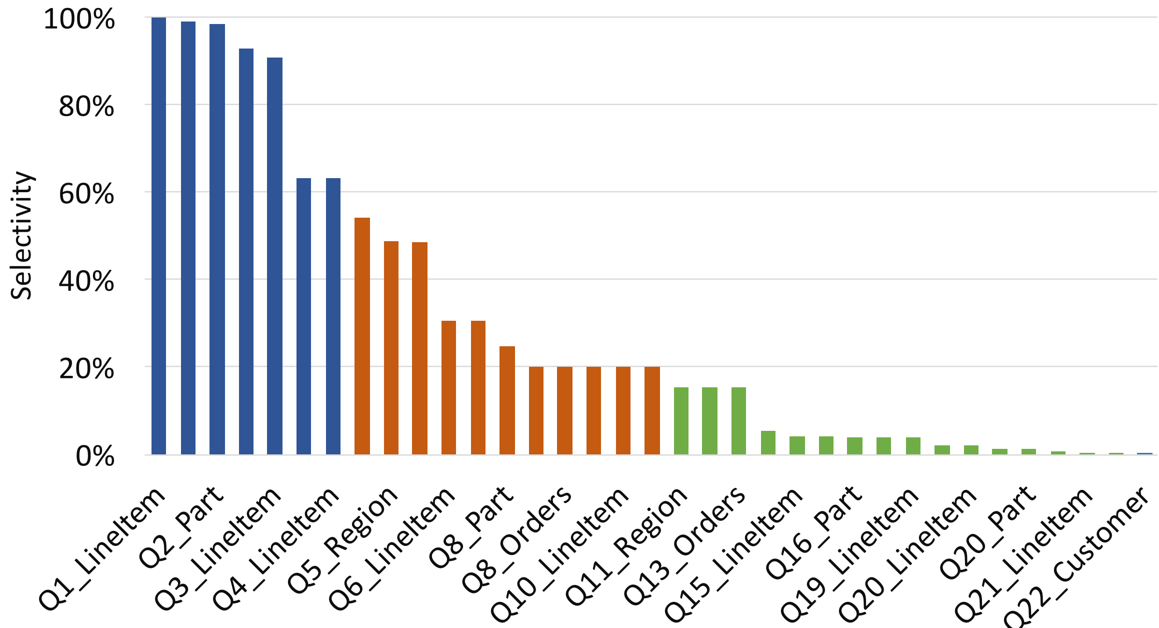


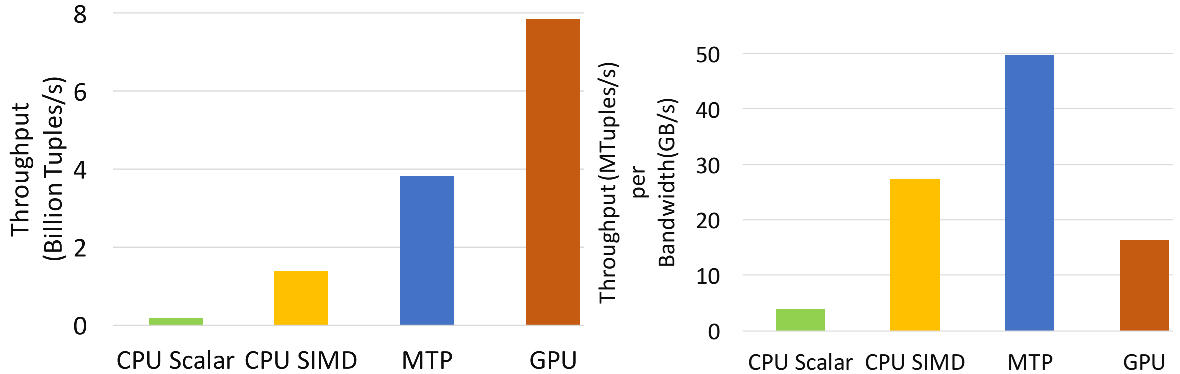
Figure 4.9: Selectivity of TPC-H queries. Each color marks the range of selectivity: (blue) above 60%, (orange) 50%-20%, (green) below 10%.

```
SELECT count(*)
FROM lineitem
WHERE l_shipdate >= date '1995-01-01'
and l_shipdate < date '1996-01-01'
and l_discount between 0.04 and 0.06
and l_quantity < 24;
```

Figure 4.10: TPC-H Query6

All of the selections in the benchmark were conjunctions, again excluding Q_{19} which has a mix of conjunctions and disjunctions.

In order to capture the real effect of the MTP design in the selection operation, we would like to isolate the effect of all relational operators in the query. This makes Q_6, shown on Figure 4.10, an ideal candidate for our evaluation. We run the query Q_6 on the TPC-H Lineitem table with the scale factor of 10. The measured selectivity of this query is 1.91%. Figure 4.11a presents the raw performance of query Q_6 executed by the various



(a) Raw performance comparison of CPU, MTP and GPU implementations

(b) Bandwidth normalized performance comparison of CPU, MTP and GPU implementation

Figure 4.11: TPC-H query Q6 performance evaluation.

	CPU Scalar	CPU SIMD	GPU	MTP
Memory Fetches(%)	100	18.75	18.75	11.4
Evaluations (per Row)	3	3	3	1.83
Effective Bandwidth speedup	0.47	3.44	2.01	6.13
Peak Bandwidth Utilization (%)	47.6	61.2	36.6	70.3

Table 4.4: Performance Evaluation of TPC-H query Q6 on CPU, GPU and MTP implementations.

architectures. We observe the same throughput trend as discussed before. Due to the high memory bandwidth GPU achieves the highest raw performance followed by the MTP design, the CPU SIMD, and the CPU Scalar implementation. However, when we compare the throughput efficiency in Figure 4.11b, the MTP implementation is 13x, 3x and 1.8x more bandwidth efficient than the CPU Scalar, GPU and CPU SIMD, respectively.

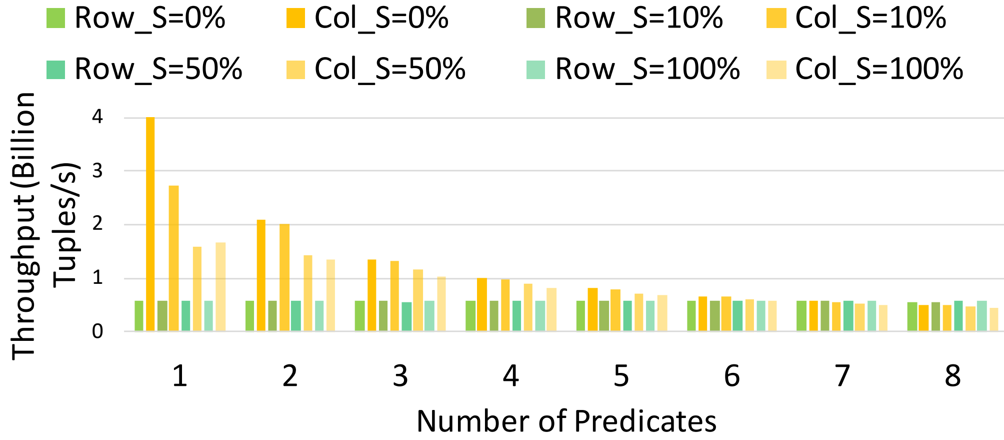
Furthermore, to confirm the advantage of our MTP design, we also measured the total number of predicate evaluations and memory fetches. The CPU and GPU implementations access the common columns ($l_{shipdate}$, $l_{discount}$) only once. However, MTP treats them as two independent attributes and fetches the same column again only if required for

further evaluation. The CPU Scalar implementation accesses all 16 columns per row of the table, therefore it is considered as a 100% memory access. The CPU SIMD and the GPU implementations need only 3 out of 16 columns to evaluate the query, contributing to 18.75% of memory accesses. We used counters to keep track of the number of memory fetches and the number of evaluated predicates for the MTP implementation. Memory fetches with MTP amount to 11.4% of total memory accesses. As suggested in [75], we compute the effective bandwidth speed-up by taking the ratio of the total size of the Lineitem relation processed per unit time and the peak bandwidth. Finally, we also report the actual peak bandwidth utilization in Table 4.4.

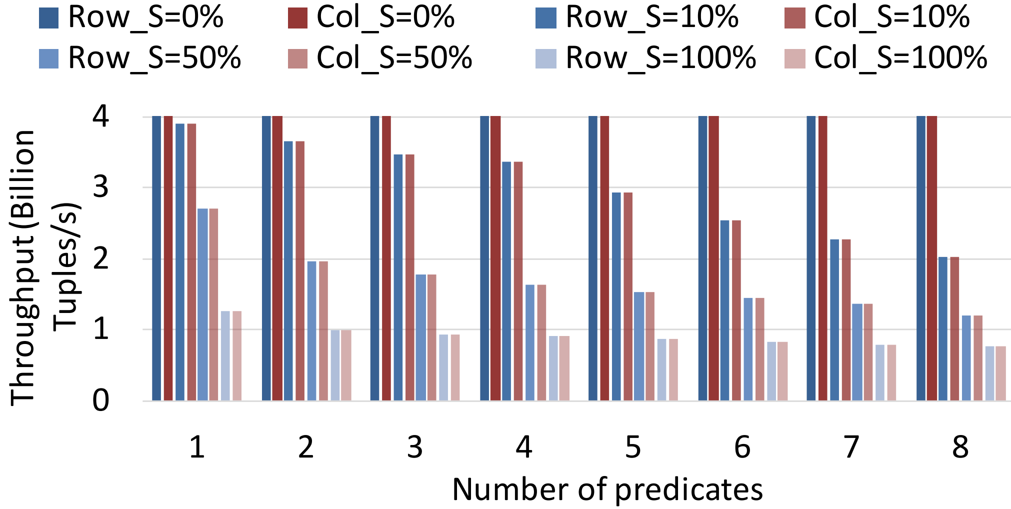
4.2.5 Datalayout Independence

We proceed with an experiment that tests the behavior of each approach for the row and column storage formats. The MTP design, utilizing the Convey HC-2ex memory subsystem, achieves performance that does not rely on any form of data alignment in memory. Its performance only depends upon accessing individual columns of a tuple for query evaluation.

On the other hand, both CPUs and GPUs are optimized for cache line accesses. As a result, we expect their performance to depend heavily on the different storage formats (row and column major). This is well known for GPUs as they depend on grouping the execution of threads into warps. This grouping is not only relevant to computation, but also to global memory accesses, making column access more advantageous. Related literature has unanimously promoted the use of column major [10, 39, 35, 74] data format for GPUs



(a) CPU throughput measured for row vs column data layouts



(b) MTP throughput for the row vs column layouts

Figure 4.12: Performance comparison of the CPU and the MTP implementations with row-major and columnar data layouts.

given its superior performance against the row data format. Hence we do not consider GPUs in the next experiment.

Figure 4.12a compares the performance achieved by the CPU implementations, namely, Scalar on row and SIMD on column store for varying selectivities and number of predicates. The columnar data layout leads to efficient cache access when only few

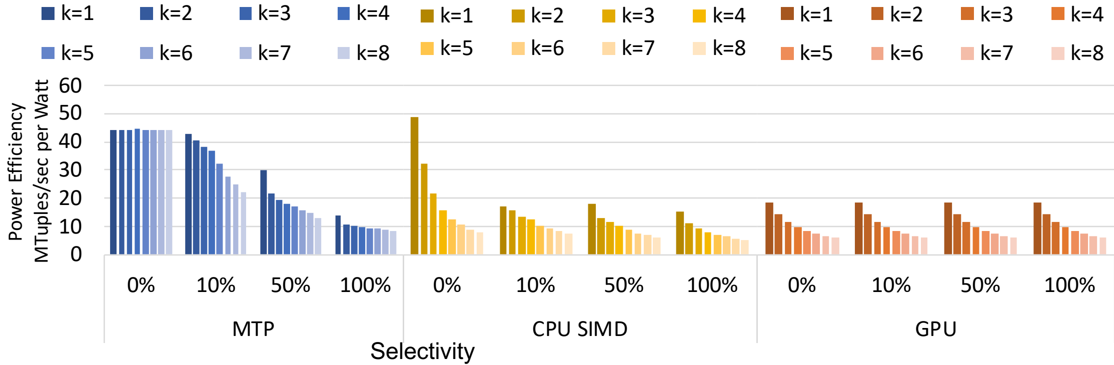


Figure 4.13: Comparison of Power Efficiency on MTP, CPU and GPU systems.

predicates are required for evaluation. This directly translates into high throughput which is 8x over the performance achieved by a row-store data layout. However, as the number of predicates increases, the amount of data accessed by both the row-store and the column store implementations converge and so does their performance. Figure 4.12b shows the MTP performance on row and column store data layouts. For a given number of predicates and selectivity, the number of memory accesses does not depend upon the type of data layout and therefore the MTP performance remains unaffected.

4.2.6 Power Utilization

To further justify our FPGA-based MTP design, Figure 4.13 presents the power efficiency results measured in Million Tuples/s per Watt. We compared the power efficiency of CPU SIMD, GPU, and MTP.

The measured on-chip power consumption on each FPGA of HC-2ex machine is 21 Watt and total power supply is 25 W, that gives us the total FPGA power consumptions as 109W (25 + 21*4). On CPU, we use the manufacturer TDP (thermal design power) rating of 105 W as prescribed in [65]. On GPU, the average power consumption was measured to be

Table 4.5: FPGA area utilization for the selection engine

# Engines	Registers	LUTs	BRAMs
1	304,966	248,149	307
	12.48%	20.31%	25.56%
4	325,217	263,639	343
	13%	21.58%	28.5%
6	347,217	279,639	380
	14.2%	22.89%	31%

155W for the design with a power supply of 600W [4]. We compare the power efficiency of all the devices in Figure 4.13. The power efficiency is coherent with our throughput evaluation. Since the MTP throughput drops with the selectivity and number of predicates, it directly affects the power efficiency too. It is 1.4x - 5.6x better than CPU-SIMD on 0% selectivity. However for $S = 100\%$ the power efficiency drops to 0.9x-1.7x. Similarly, in comparison to GPU we get 4.3x - 1.4x power savings at $S = 0$ but for $S = 100\%$, the efficiency is on par with GPU.

We use the same power statistics to evaluate the power efficiency of different architectures on query Q_6 of TPC-H benchmark. MTP design is 3.4x and 2.6x more power efficient than GPU and CPU-SIMD implementations on this query.

4.3 FPGA Area Utilization

Table 4.5 shows the area utilization (registers, LUTs, and BRAMs used). Many resources are shared between the engines as their number increases. For example, one selection engine uses 12.48% of the available registers, whereas 6 engines use only 14.2%. Also note that with increasing number of engines there is very minimal increase in the number of logic resources (LUTs). Overall, the space utilization on the FPGA is low, leaving

sufficient space to extend our design with various optimizations or operators (projections, aggregation, join). This also gives us insight into how well the design could scale on another platform with more, or less memory channels. These results include Conveys memory interface wrapper, which does not occupy a significant portion of the area. We see that even with 6 engines the Virtex-6 still has plenty of room for more engines; only 14% of the logic is utilized. The current design is therefore limited by the memory channels.

4.4 Conclusion

In this chapter, we presented a lightweight hardware multithreaded implementation of the *selection* operation for in-memory relational databases, the MTP. This design facilitates fine grained data access, thus is completely oblivious to a particular data layout (row or column) and avoids fetching irrelevant data.

We thoroughly evaluate it against the best implementations on CPU and GPU. Experimental results show that the MTP throughput varies only with the predicate probability and is bounded by the memory channels. The GPU achieves the best raw performance over the entire parameter space.

Due to the memory bounded nature of the selection operation, we attribute this performance to the high GPU memory bandwidth. Instead, this work achieves a speedup between 1.4x - 4.6x over CPU SIMD and 1.4x - 6.7x over GPU implementations for the query selectivity that ranges from 0% to 50% (77% of TPC-H queries). On higher selectivity values performance is bounded by the number of write channels.

We also evaluate MTP on the TPC-H query *Q6*. On this benchmark query, we achieve 13x, 3.2x and 1.8x normalized speedup over CPU Scalar, GPU and CPU SIMD respectively, while saving 89% of total evaluations.

Chapter 5

Sparse Matrix and Vector Multiplication

Sparse Matrix and Vector Multiplication (SpMV) is one of the most important kernels for numerous scientific applications. On one hand, low compute to memory ratio makes SpMV a memory bound problem and on the other hand, commercial off-the-shelf (COTS) architectures provide insufficient main memory bandwidth for available computation resources. Furthermore, SpMV requires random access into a memory space which is far too big for cache. Hence, it becomes difficult to utilize the main memory bandwidth which is already scarce. In this chapter, we exploit multithreaded architecture to mask the memory latency incurred while performing a non-sequential access on the input vector. Through our evaluation, we show that the proposed architecture can achieve as high as 95% of the expected upper bound performance. We compare our evaluation results to the latest GPU architectures namely K20, K40 and Titan GV100.

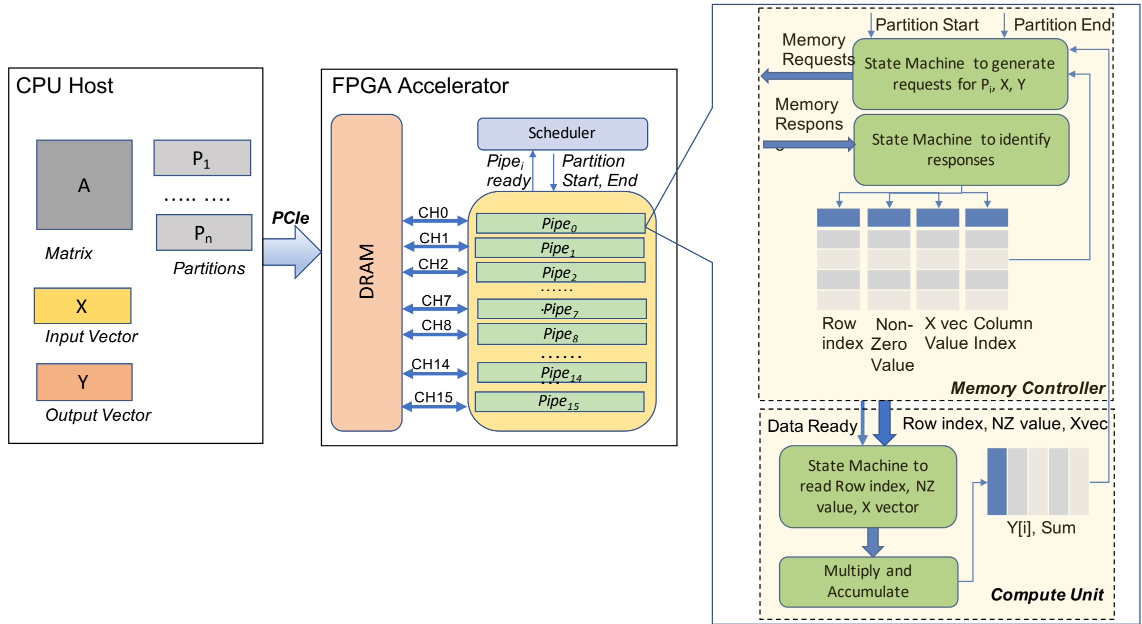


Figure 5.1: SpMV design on FPGA.

5.1 SpMV Architecture Overview

Figure 5.1 presents the system level accelerator model: an FPGA co-processor is used in conjunction with a multi-CPU host system. Therefore data are initially stored and preprocessed in the CPU DRAM. Data are transferred from CPU DRAM to accelerator DRAM via an interconnect such as PCIe. Once the data is transferred closer to the FPGA, the execution starts by fetching the partition pointer. Each partition is completely processed by a pipeline, P_i, hereafter referred as *pipes* or PEs. To ensure load balancing between *pipes*, dynamic scheduling is applied. Each pipe raises a *ready* flag as soon as it is done processing the previous partition. This type of scheduling ensures approximately equal division of work between *pipes*.

The multithreaded SpMV kernel focuses on double precision floating point operations. We assume the input is a 1D partitioned COO encoded sparse matrix, with double

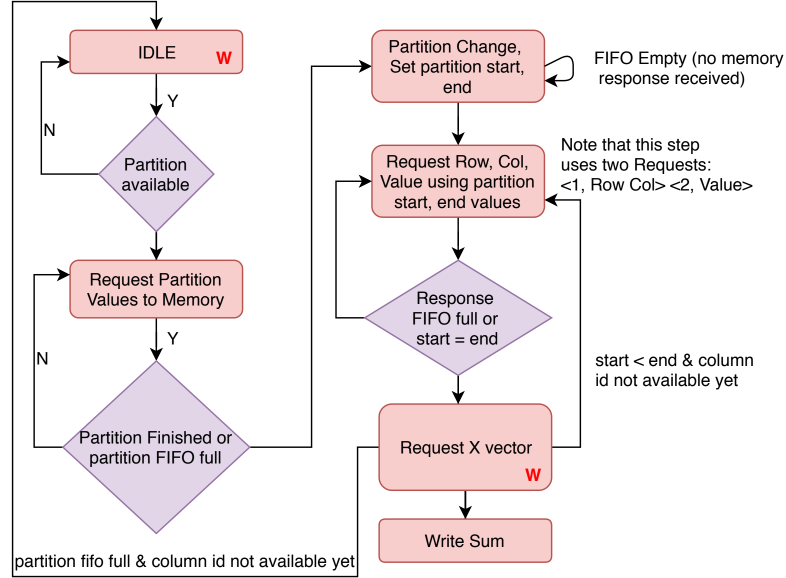


Figure 5.2: Memory controller state machine

precision floating point values and 32 bit index (col and row). Initially, we chose traditional 2D blocking technique, where a matrix is divided into sections called blocks or submatrices. However, preliminary experiments showed that for a smaller block sizes, blocking introduces 'empty' blocks, leading to unnecessary memory fetches of a block pointer.

We also view registering intermediate Y values superior to caching X vector values. Caching X vector can give some performance benefits, however, the gains are limited. This observation is also confirmed in [24] where the average amount of X vector reuse per X vector fetch from external memory is reported to be 8.8 for the block size of 128.

The preprocessing step involves splitting the input matrix into work items which can be parallelised and operated on independently. Since the COO format is used, partitioning can be achieved by row slicing: splitting the matrix into disjoint sets of adjacent rows. The complexity of preprocessing is $O(nnz)$.

The multithreading paradigm allows us to completely separate the memory and the compute units. Each row is a thread. Thread states must maintain the running sum, and the start/end positions for the memory requests and the partition ID. As requests are fulfilled the data is sent to a summation unit which produces the final sum-of-products. Each PE manages the requesting, multiplying, and summing for multiple threads (rows) concurrently.

A PE has its own *Memory Controller* and the *Compute Unit.* These two components work independent of each other. The *Memory Controller* has a state machine that balances the request type between different data structures (partition pointer, row column, value and X vector array). A high level description of this state machine is described in Figure 5.2. Note that the states marked with *W* are identified as wait states. During the execution, the state machine may halt at these states due to various reasons like lack of data response, FIFO empty etc. In these situations, instead of keeping the channel idle, it is used to write back the final sum to the memory. The main objective of this module is to keep the *Compute Unit* as busy as possible by fetching the required data as fast as it can. The Convey HC-2ex machine supports in-order memory requests. The physical accesses to memory are fulfilled out-of-order, but the HC-2ex uses a custom crossbar to reorder the data before returning it to the PE. Thread states can therefore be stored in FIFO buffers.

The *Compute Unit* comprises of: (1) a small control logic that streams row ID, matrix value and the X vector value from the *Memory Controller* and identifies the end of partition (2) floating multiplier, (3) the reduction circuit. This work uses a reduction circuit similar to [38]. The work presented in [38] suggests the use of two on-chip memories

to store the partial sums. However, as an interesting side-effect of partitioning the matrix row-wise, we require only one on-chip memory to store partial sums. And this memory is addressed using row IDs local to the partition. This reduction circuit handles multiple rows concurrently, and can read a new element every cycle. The circuit assumes all data for one row enters the datapath before any data from another row enters. This assumption holds for our kernel because of the HC-2ex's in-order requests. This reduction circuit is only needed if the kernel is compiled for floating point operations because addition requires multiple cycles.

5.2 Performance Bound

The performance of sparse matrix and vector computation depend on parameters like memory bandwidth and the on-chip resource availability. We use a classical roofline model [85] to describe the upper bound on performance of the SpMV. This model suggests that the maximum floating point operations is given by the following formula:

$$\begin{aligned}\texttt{Attainable Ops/sec} = \; & \texttt{Min} \; \{ \; \texttt{Peak Floating Point Performance,} \\ & \texttt{Memory Bandwidth x Operational Intensity} \; \}\end{aligned} \tag{5.1}$$

A similar model is also proposed in [56]. In this work the sparse matrix is partitioned into vertical and horizontal blocks. However, the purpose of the vertical partitioning technique is not well understood since the input vector is not cached and is fetched from the memory as and when required. Similarly, the design of a processing engine along with the issue of load imbalance between engines is not addressed in this work.

This analysis assumes that the data is stored on off-chip memory. An aggregated bandwidth of `b` is available between off-chip memories and the FPGA. The on-chip FPGA memory can hold upto `m` words of the partial results. Compared to DRAM bandwidth, on-chip memory bandwidth is assumed to be infinite.

The compute unit, MACC, performs one multiply and accumulate per cycle. It operates at a frequency of `f` Hz. We assume that there are `R` resources available on FPGA and each MACC uses `r` of them, giving us $k = \left\lfloor \frac{R}{r} \right\rfloor$ MACC units. Without loss of generality, matrices are assumed to be nxn in size. We denote the density of non-zero elements in a sparse matrix α, i.e. if the number of non-zero elements is NNZ and the total number of elements is N, then $\alpha = \frac{NNZ}{N}$. Finally, we assume large matrices such that $n, m \gg k > 1$.

It is intuitive that the computational intensity of FPGA is bounded by the availability of hardware resources. The roofline equation 5.1 suggests that the number of operations performed by FPGA are given by Equation 5.2. Each MACC generates one result each cycle by performing 1 add and 1 multiply operation.

$$P_{comp} = 2kf \tag{5.2}$$

Recall that the matrix elements are read from DRAM and therefore the overall performance is still bounded by the memory bandwidth. The limited on-chip memory also restricts the amount of data that can be fetched from the DRAM. This behavior can be analytically represented by Equation 5.3 and is also called a memory bound. The right hand side of this equation is also known as the operational intensity of an application.

$$P_{mem} = b.\{\ O_{op}/O_{mem}\} \tag{5.3}$$

Since, the FPGA performance is bounded by both memory and the compute, Equation 5.1, 5.2, 5.3 presents the maximum performance P_{max} that can be achieved on FPGA

$$P_{max} = min\{\ P_{comp}, P_{mem}\} \tag{5.4}$$
$$= min\{\ 2kf, \frac{b.O_{op}}{O_{mem}}\}$$

The SpMV always perform number of operations equal to:

$$O_{op} = 2\alpha n^2 \tag{5.5}$$

This number is independent of any memory constraint (on/off chip). On the contrary, the number of memory accesses, O_{mem}, depend upon different ways of scheduling requests and also the on-chip memory m. In our technique, we use a simple COO matrix format and partition the matrix row-wise. Each partition is of size $p_r * n$. This partition size is chosen in such a way that partial sums can be stored on-chip. Correspondingly, the output vector y is of size $p_r * 1$. Number of non-zero elements in each partition is given by $\alpha * p_r * n$. For each non-zero element, the corresponding x vector is also fetched from the off-chip memory. The probability that there is at least one non-zero element in a column of a partition is given by $\boldsymbol{P_{non_zero_per_column} = 1 - (1 - \alpha)^{p_r}}$. The expected number of x vector elements required for one partition are $\boldsymbol{n * P_{non_zero_per_column}}$.

Additionally, output vector y is written back at the end of computation. It is possible to schedule this stage at the end of each partition. The probability that there is at least one non-zero element in y is given by $\boldsymbol{P_{non_zero_per_n} = 1 - (1 - \alpha)^{n}}$. The expected number of write operations is $\boldsymbol{P_{non_zero_per_y} * n}$. The total $\boldsymbol{O_{mem}}$ for all

partitions is given by the Equation 5.6.

$$\boldsymbol{O_{mem} = [2\alpha p_r n + n * P_{non_zero_per_column}]\frac{n^2}{p_r n} + n * P_{non_zero_per_y}} \tag{5.6}$$

Note that on Convey HC-2ex, each request is 8 bytes. It takes 2 requests (1: $\boldsymbol{< row, col >}$ 2: $\boldsymbol{< value >}$) to fetch a non zero value. The maximum size of a partition is bounded the on-chip memory $\boldsymbol{m}$. From Equations 5.7 5.5 5.6, we get

$$\boldsymbol{P_{max} = min\{\ 2kf, \frac{2.b}{c}\}} \tag{5.7}$$

$$\text{where } \boldsymbol{c = 2 + \frac{1}{\alpha}[\frac{P_{non_zero_per_column}}{m} + \frac{P_{non_zero_per_y}}{n}]}$$

Since SpMV is known to be a memory bound operation, the limiting factor is on-chip memory and the memory bandwidth. Based on Equation 5.7 , theoretical peak performance of FPGA-based SpMV operation is given as follows:

$$\boldsymbol{P_{max} = \frac{2.b}{c}} \tag{5.8}$$

5.3 Experimental Results

5.3.1 Experimental Setup

The Convey HC-2ex has 4 Virtex 6 LX760 FPGAs, called application engines(AEs). This machine supports 16 channels per FPGA, allowing us to put 16 independent and deep pipelines that can run in parallel. In this work, we put 4 PEs per AE and replicate them across four AEs at runtime, allowing us to run total 16 PEs in parallel. Control registers in the AE specify the number of threads (i.e. rows) needed by the SpMV kernel. They are

also used to specify the base addresses for each memory array used by the kernel. When using multiple AEs the values are partitioned to balance the workload.

Integrating a kernel into the HC-2ex requires all memory requests to communicate with Convey's memory interface. Designs are placed and routed varying the number of pipes from 1 to 4 and the size of a from 128 to 1024. Area utilization (including the wrapper) for a single AE is shown in Table 5.1. The design consumes marginally more registers and LUTs as the block size is increased. For instance, the *pipe* with 128 block size uses 10% of registers and 19% of the LUTs . The amount of same resources increases to 11% and 22% respectively for a block size of 1024. On the other hand, memory consumption grows considerably from 8 to 18% when we change the block size from 128 to 1024. Regardless, the only purpose of blocking a matrix is to store partial sums on-chip and it does not affect the overall performance (see Section 5.3.3). Also, note that increasing the number of pipes, minimally (1 -2%) increases the area per block size.

The GPU results are obtained on Nvidia K40 and Titan GV100. K40 has 2496 cores and operates at a frequency of 704 MHz. The peak bandwidth achieved on this machine is 288 GB/s. Nvidia Titan GV100 is the latest GPU enhanced with Volta architecture. It has 5120 cores, operating at a frequency of 1132 MHz. It provides a peak bandwidth of 870 GB/s. The implementation uses NVIDIA cuSPARSE[64] library. The cuSPARSE [64] was developed at Nvidia for sparse linear algebra and graph computations. It supports multiple sparse matrix formats including all those highlighted in this thesis. The library's performance benefits over multi-core CPU architectures were shown in [16]. Out of various formats supported by the cuSPARSE library, we use CSR representation as suggested

Table 5.1: FPGA resource utilization for SpMV engines.

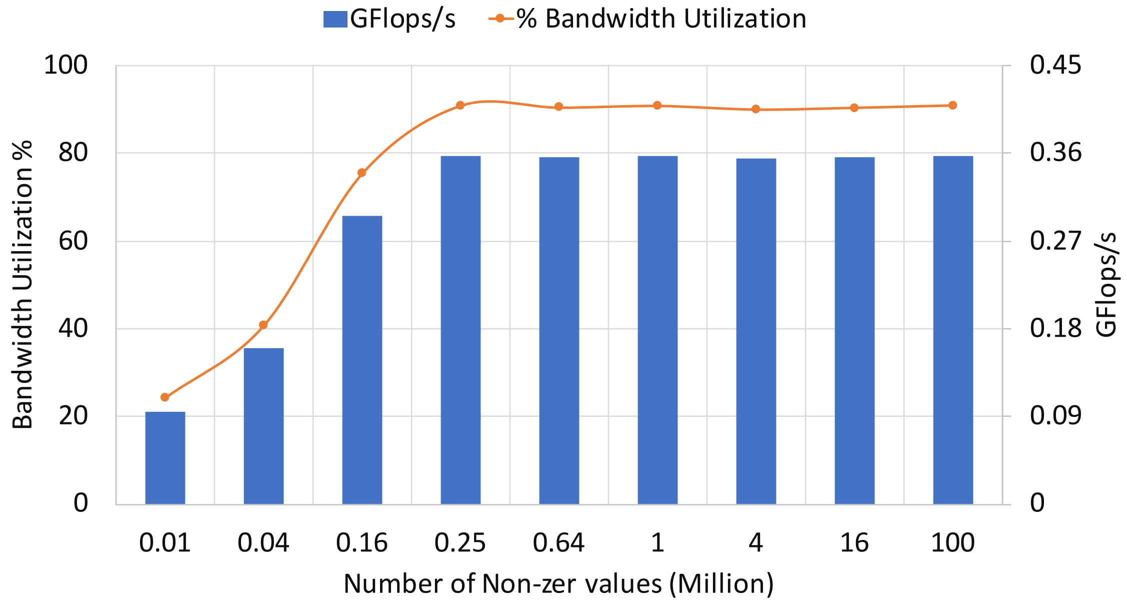
Block Size	# of Pipe	Registers	LUTs	BRAMs
128	1	81,200 (8%)	68,636 (14%)	40 (5%)
	2	89,120 (9%)	79,062 (16%)	47 (6%)
	4	99,843 (10%)	94,526 (19%)	61 (8%)
512	1	86,352 (9%)	72,932 (15%)	18,089 (13%)
	2	98,443 (10%)	82,092 (17%)	19,348 (15%)
	4	106,443 (11%)	102,650 (21%)	23,229 (17%)
1024	1	86,879 (9%)	75,092 (15%)	18,779 (14%)
	2	95,662 (10%)	85,595 (18%)	21,348 (16%)
	4	108,043 (11%)	105,650 (22%)	25,109 (18%)

by [58]. The ELL format is only applicable to small row lengths, and is also more likely to encounter memory allocation failure compared to CSR [58]. Results on the Tesla K40 use CUDA version 6.0, and GCC version 4.6.3. Titan GV100 use CUDA version 9.1 and GCC version 4.9.1.

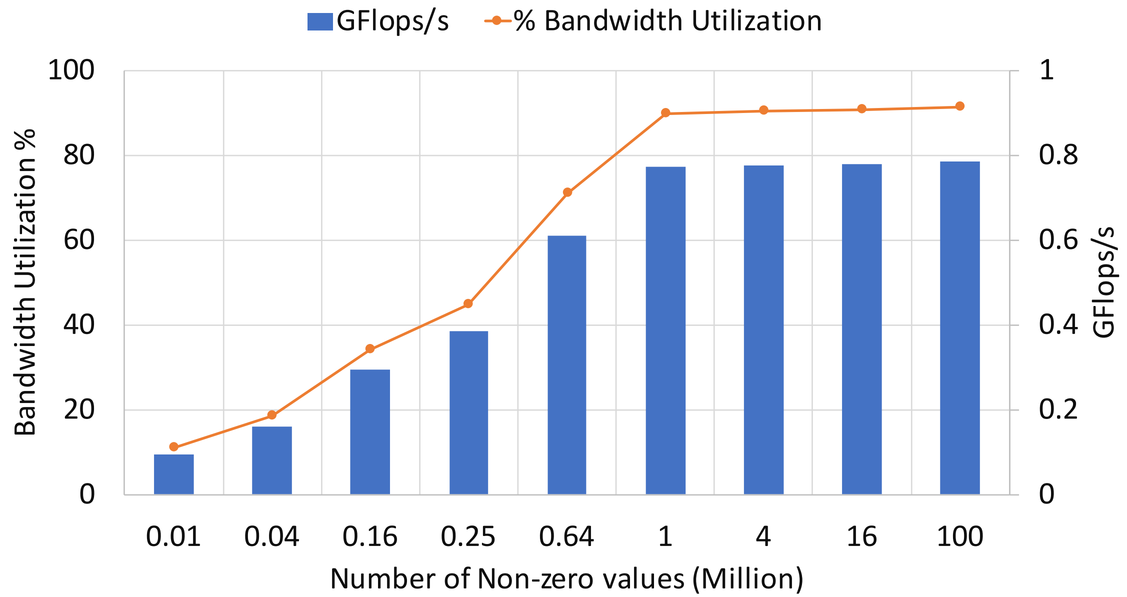
To measure performance, we have used the billion FLOPs per second (GFLOPS) metric, which is computed as $\frac{2 * NNZ}{t * 10^9}$ for the equation y = Ax, It should be stressed that the runtime does not include the host-side data preparation time for both the sparse matrix and the two vectors.

5.3.2 NNZ Vs Sustained Bandwidth

As described in Chapter 2.2.2 , multithreaded paradigm achieves performance by masking memory latency. The custom SpMV kernel masks latency by generating multiple outstanding requests. The build up phase is dominated by memory requests until enough threads (rows) are buffered by the kernel. The performance is therefore dependent upon the matrix, and the architectures memory latency. The build up cost negatively affect smaller

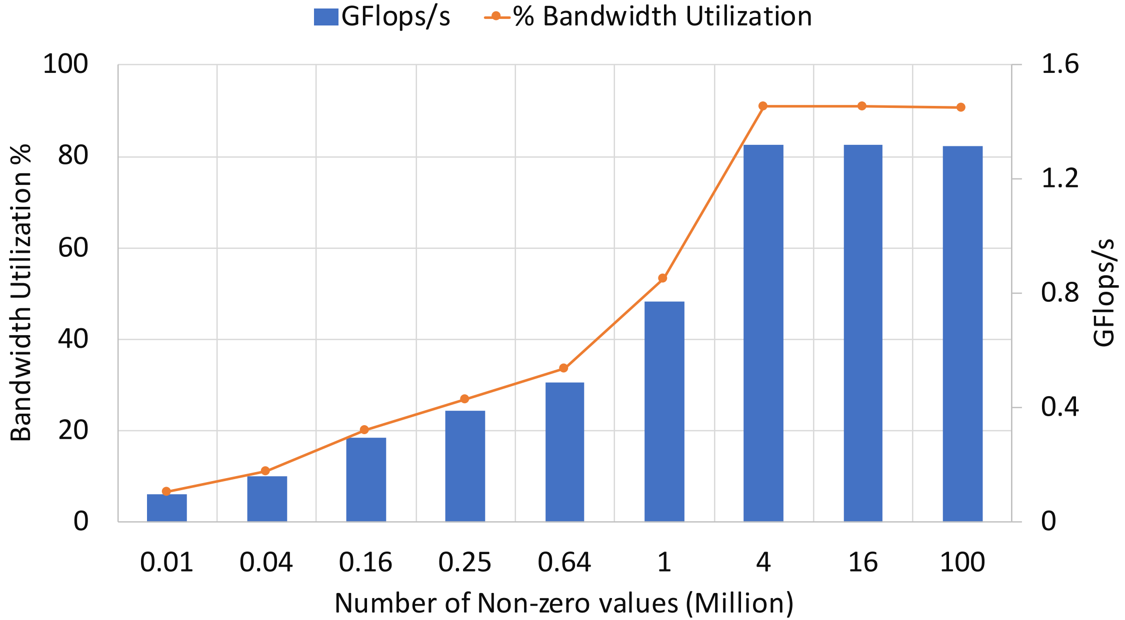


(a) Minimum number of non-zero values required by 4 engines



(b) Minimum number of non-zero values required by 8 engines

matrices. We run a series of test on the Convey HC-2ex to determine what size of datasets are “large enough” to mask the initial startup costs.



(c) Minimum number of non-zero values required by 16 engines

Figure 5.3: Minimum number of non zeros as number of engines (channels) are increased

In this experiment we use dense matrices stored in a partitioned COO format. By doing so we remove all irregularity, and therefore get a better estimate of the startup cost. With no irregularity any optimizations or caching implemented Convey's memory crossbar will yield their best performance, and in turn give us the minimum dataset size needed to mask the startup costs. We performed this experiment with 4, 8 and 16 PEs. Each of these configurations are connected to 4, 8 and 16 memory channels, providing a sustained bandwidth of 4.8 GB/s, 9.6 GB/s and 16.2 GB/s respectively.

The results are shown in Figure 5.3. The number of non zero values in a sparse matrix should commensurate with available bandwidth. For instance, to saturate a bandwidth of 4.8 GB/s on four PEs, 250,000 NZEs are required. Similarly, on 8 and 16 PEs, matrices with minimum 1 and 4 Million non zero elements obtain maximum performance.

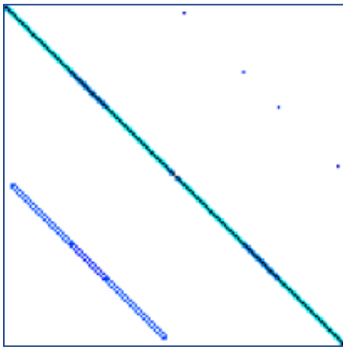
Based upon this experiment, we divide our benchmarks into two distinct groups.

Table 5.2: Benchmarks used for performance evaluation of multithreaded SpMV design

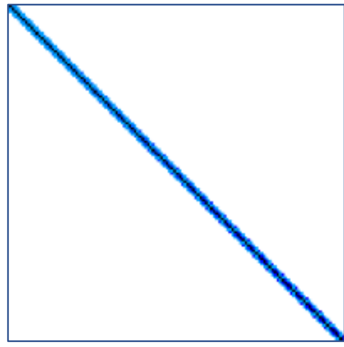
Benchmark	Rows	NNZ	NNZ per Row	Density
Group 1				
d8192	8192	41,746	5	0.00062
t2_d9	9,801	87,025	9	0.00090
raefsky1	3,242	294,276	91	0.028
psmigr_2	3,140	540,022	172	0.054
scircuit	170,998	958,936	6	3.2E-05
Group 2				
torso2	115,967	1,033,473	9	7.68E-05
Economics	206,500	1,273,389	6	2.99E-05
pdb1HYS	36,417	4344765	119	0.0032
FEM/Cantilever	62,451	4,007,383	64	0.00102
pwtk	217,918	11,524,432	53	0.000242
consph	83,334	6,010,480	72	0.00086
cage15	5,154,859	99,199,551	19	3.73E-06
nlpkkt120	3,542,400	96,845,792	27	7.72E-06
Serena	1,391,349	64,531,701	46	3.33E-05
kron g500-logn2	2,097,152	182,082,942	87	4.14E-05
nlpkkt200	16,240,000	448,225,632	28	1.70E-06
Queen_4147	4,147,110	329,499,284	80	1.91E-05
HV15R	2,017,169	283,073,458	140	6.95E-05
Bump_2911	2,911,419	127,729,899	44	1.5E-05
Flan	1,564,794	117,406,044	75	4.7E-05

- *Group1:* where NNZ $<$ 1 Million. These were used in [44] [81], [52] [63].
- *Group2:* NNZ $>$ 1 Million, listed in [44] [81]. Top five benchmarks with the largest value of NNZ are also included to evaluate the performance of the accelerators. These benchmarks are more indicative of some of real scientific, economic and engineering workloads.

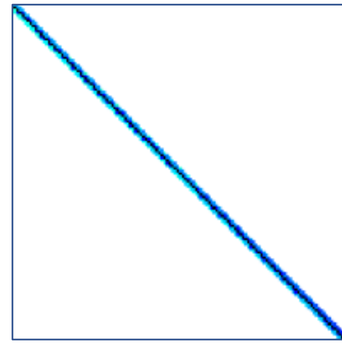
The benchmarks evaluated in this paper shown in Table 5.2 and Figure 5.4, are obtained from the SuiteSparse Matrix Collection [29].



dw4096, $n_r = 8192$
$\propto = 0.0006$

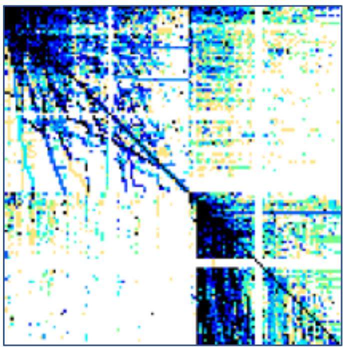


t2d_q9, $n_r = 9801$
$\propto = 0.0009$

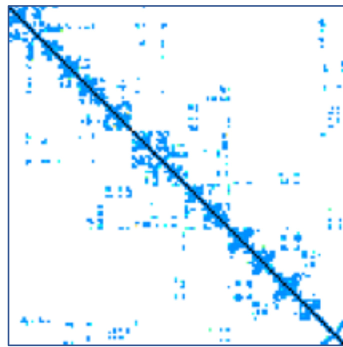


epb1, $n_r = 14734$
$\propto = 0.0004$

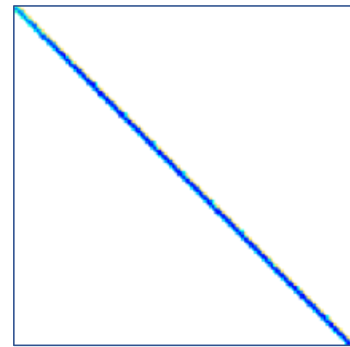
(a)



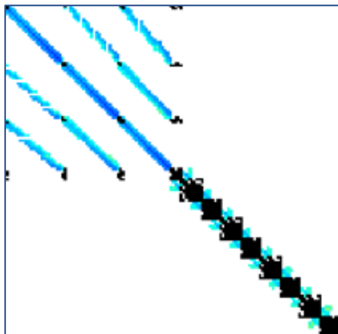
webbase, $n_r = 1000005$
$\propto = 0.000003$



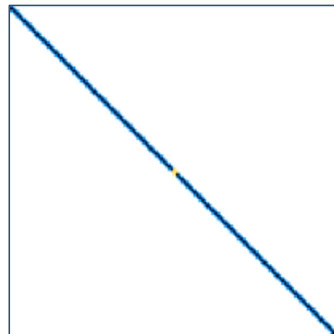
pdb1HYS, $n_r = 36417$
$\propto = 0.0032$



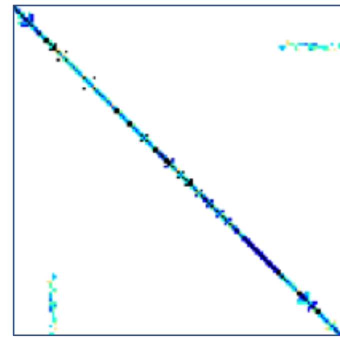
cant, $n_r = 62451$
$\propto = 0.0010$



Consph, $n_r = 83334$
$\propto = 0.00086$

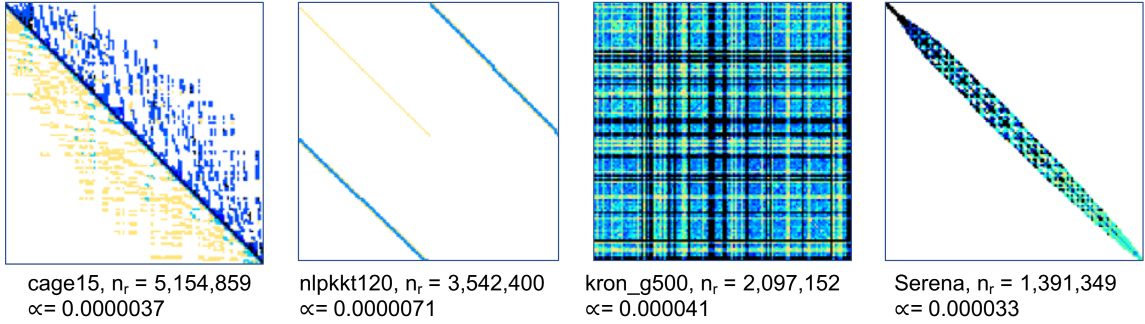


mac_econ_fwd500, $n_r = 206500$
$\propto = 0.000029$



pwtk, $n_r = 217918$
$\propto = 0.0002$

(b)



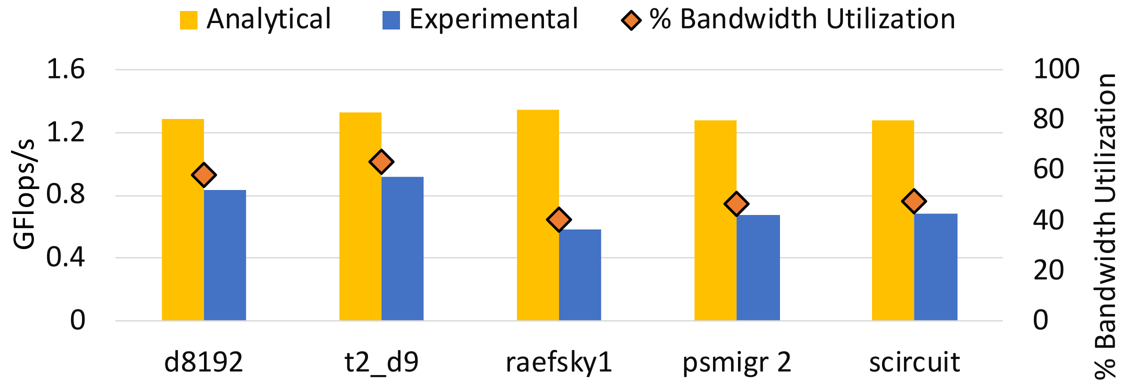
(c)

Figure 5.4: Sparse matrix structure for (a) *Group 1* and (b,c) *Group2*

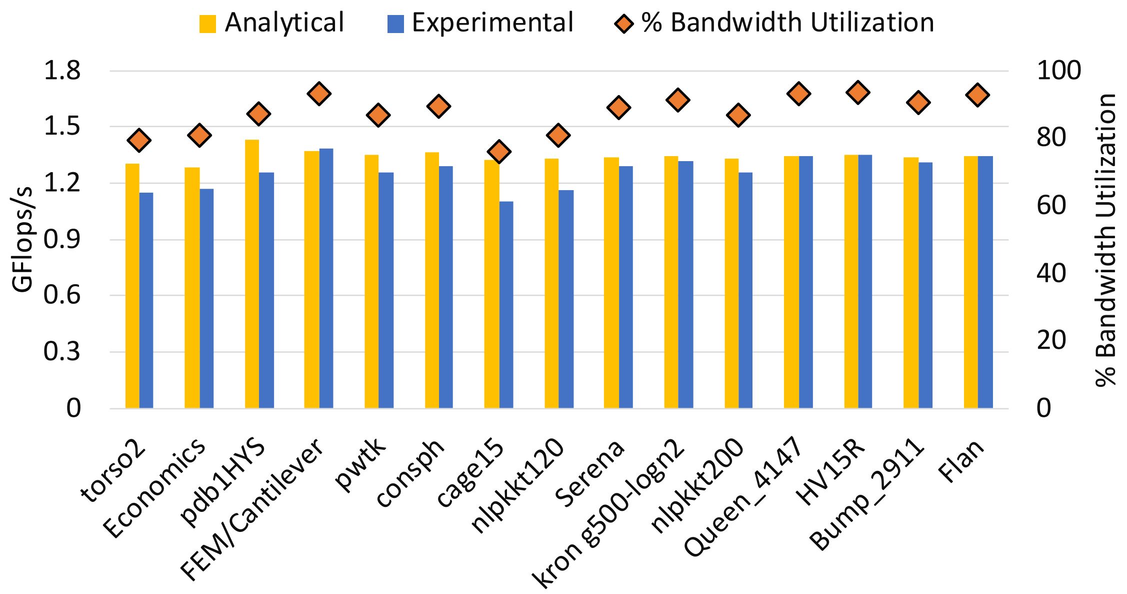
5.3.3 Performance Evaluation of Multithreaded SpMV

In this experiment we validate the analytical model described in Section 5.2. FPGA-based SpMV experimental results are obtained by setting a block size to 128 and running 16 PEs in parallel. Figure 5.5 compares the GFlops obtained by the proposed model and our experiments. Analytical GFLops reported in Figure 5.5 are calculated using Equation 5.7. Inputs to this model are matrix density, number of rows and columns, and the block size. Using these inputs we calculate the value of a constant $\boldsymbol{c}$ as shown in Equation 5.7. Finally, theoretical peak performance is calculated as $\boldsymbol{P_{max}} = \frac{\boldsymbol{2b}}{\boldsymbol{c}}$, where $\boldsymbol{b}$ is the available peak bandwidth and $\boldsymbol{c}$ is the total words read and written to DRAM. This equation is very similar to the one used in [55, 57]. However, the constant value $\boldsymbol{c}$ provides a more fine grained details about the DRAM data access.

We compare the overall performance of *Group 1* and *Group 2* separately. Group 1 was used in [81], [44] and [63] and consists of smaller matrices. The largest matrix in this group is less than 1 million NZEs. Group 2 is used in [44] [81] and consists of matrices that have NZEs greater than 1 Million. The size ranges from 1 to 448 million NZEs. It



(a)



(b)

Figure 5.5: Analytical and Experimental Performance Comparison for Group 1 and Group 2 sparse matrices

should be noted before these comparisons that GPUs have both a higher clock frequency than the FPGA, and a larger memory bandwidth. It is also important to note that the peak performance numbers between the FPGA and GPU should not be compared at face value given that GPUs have much higher bandwidth (16.2 GB/s, 288 GB/s (K40), 870 GB/s (Titan GV100)).

As shown in Figure 5.5a, *Group 1* achieves 44% - 73% of the theoretical GFlops/s. This behavior is expected as the minimum start up cost required to attain maximum performance is not met by these matrices. Additionally, the model is not biased towards any such matrix or design dependent properties and predicts much higher performance than what could be achieved by the design. This can also be verified by the bandwidth utilization across these matrices that varies from 40% - 63%. This observation is in accordance to Figure 5.3c that clearly shows that the bandwidth utilization of matrices containing close to 1M NZE ranges from 40% - 75%.

Figure 5.5b compares the theoretical and the experimental GFlops/s for *Group2*. It can be seen that *Group 2* achieves far better accuracy than *Group 1*. This behavior is a direct consequence of running bigger matrices (NNZ $>$ 1M). Because the matrix data is capable of completely saturating the available bandwidth, the design achieves much better performance. It achieves as high as 82% - 99% of accuracy when compared to the model.

We also experiment with different block sizes to identify its effect on the overall design performance. In this experiment, we restrict the number of PEs to 8 to reduce the timing pressure on hardware circuits caused due to high on-chip memory usage. We use 128, 256, 512 and 1024 as our block sizes. The results are shown in Figure 5.6. It can be observed that the performance approximately remains the same across different block sizes. This behavior is obvious as changing the block size only alter the number of partitions which in turn can lead to a different distribution of NZEs across PEs. This change, however, does not affect the total number of reads (total NZE) and writes (output vector) for a given matrix. A bigger block size will reduce the number of partitions which

may lead to insufficient workload across PEs. Nevertheless, as long as number of partition $\gg$ number of PEs, the aforementioned situation will not affect the overall performance. For instance, consider the matrix *cage15* that has 5,154,859 rows. For a block size of 128, 256, 512, and 1024, we get 40,274, 20,137, 10,069, 5,039 partitions for this particular matrix. The number of partitions for all block sizes is $\gg$ 8 (number of PEs). Therefore, the performance of our SpMV design, as shown in Figure 5.6, fairly remains constant across different block sizes.

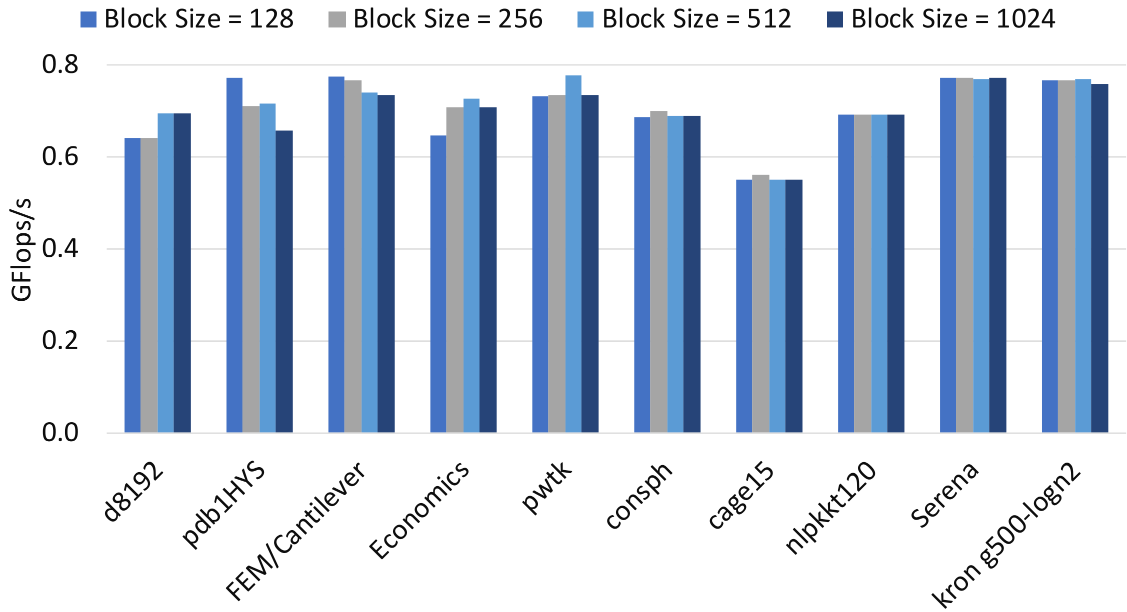


Figure 5.6: Effect of block size on the achieved SpMV performance when ran with 8 PEs. As long as number of partitions $\gg$ than number of PEs, the block size does not affect the performance.

5.3.4 Performance Comparison to GPUs

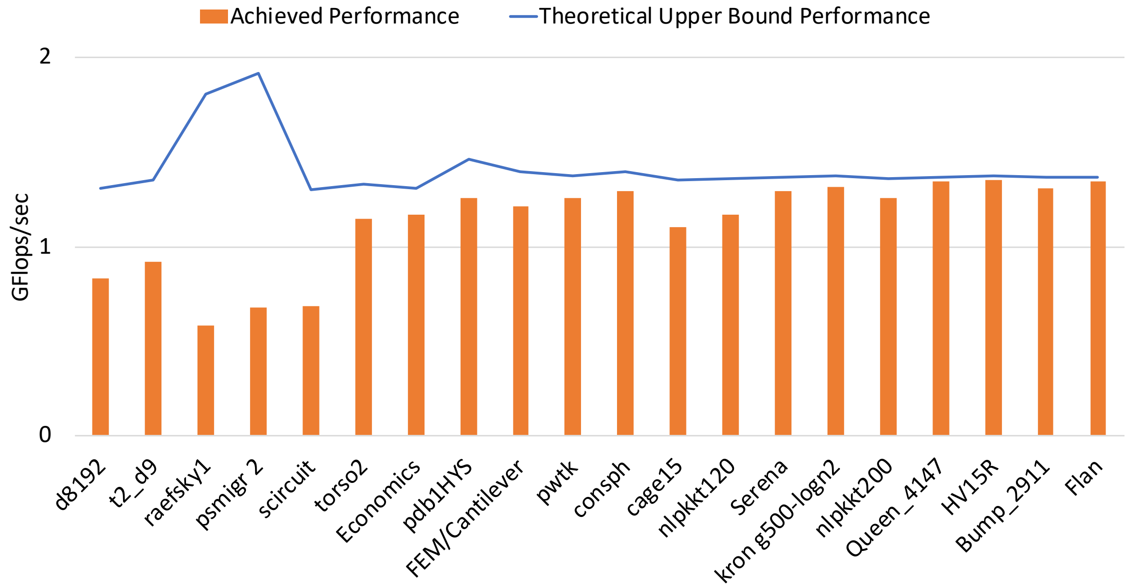
Both FPGA and GPU platforms run at different frequency and memory bandwidth. For a fair comparison, we compare the achieved performance of each of these archi-

tectures to their respective peak upper bound performance. As discussed in Section 5.2, the upper performance bound for the multithreaded SpMV is governed by Equation 5.8 and is equal to 1.35 GFlops/s. Similarly, on GPU, the upper bound on the performance is calculated by finding the a lower bound of data transfers in bytes. In a standard CSR format, a square matrix of size N*N requires 4N + 12NNZ bytes of storage for a double precision (DP) SpMV product. The input and output vector take up 16N bytes, leading to a total of 12*NNZ + 20*N bytes. The number of floating point operations (Flops) for an SpMV is at most 2*NNZ. With this information, we can calculate Flops/byte and, moreover, derive an upper performance bound, $\boldsymbol{P_{max_GPU}}$ for the SpMV in Equation 5.9.

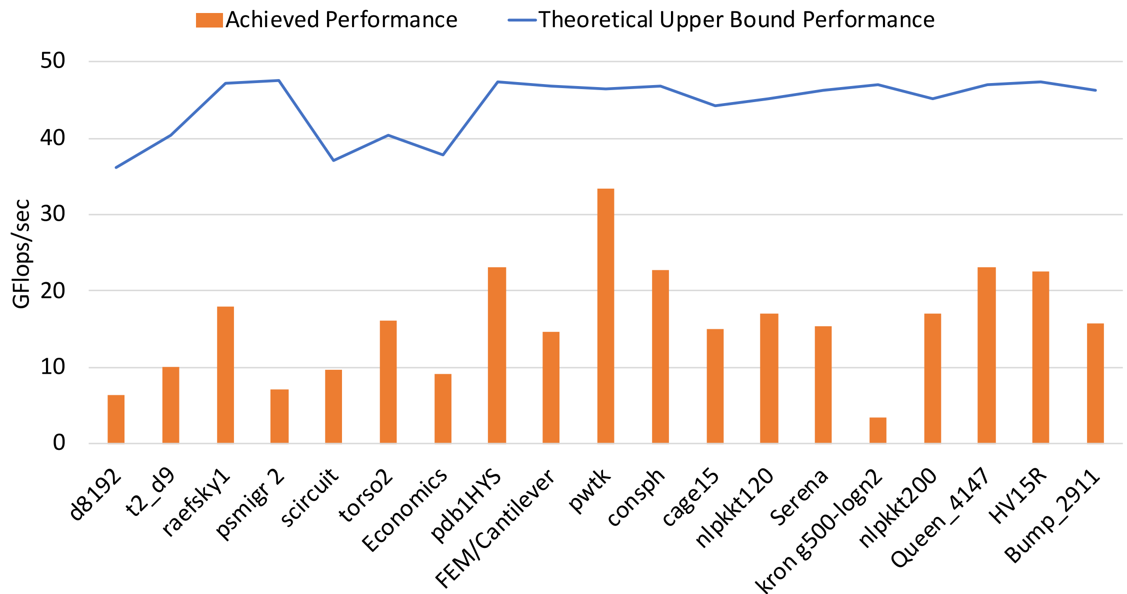
$$\boldsymbol{P_{max_GPU}} = \frac{\boldsymbol{2 * NNZ * Peak\ Bandwidth}}{\boldsymbol{12 * NNZ + 20 * N}} \tag{5.9}$$

Using Equation 5.9, we calculate the upper bound performance on K40 and Titan GV100 for each matrix presented in Table 5.2. Prior work [63, 55, 57, 9] uses a similar analysis to calculate the upper bound on the SpMV performance.

Figure 5.7 compares the achieved performance of mutithreaded SpMV, K40 and Titan to their respective peak performance. The FPGA-based SpMV achieves 35%-67% of peak flop efficiency on *Group 1* while K40 achieves 17.23%-35.12% whereas Titan achieves 10%-25%. In *Group 2*, multithreaded SpMV achieves much higher efficiency compared to *Group 1* as well as GPU-based SpMV. It is 80%-91% flop efficient whereas K40 achieves is 37.7% -47.4% and Titan achieves 40%- 66% of the peak achievable flops. Note that both GPU machines achieve much better raw performance than FPGA-based implementation. However, this behavior is obvious considering the high GPU bandwidth.

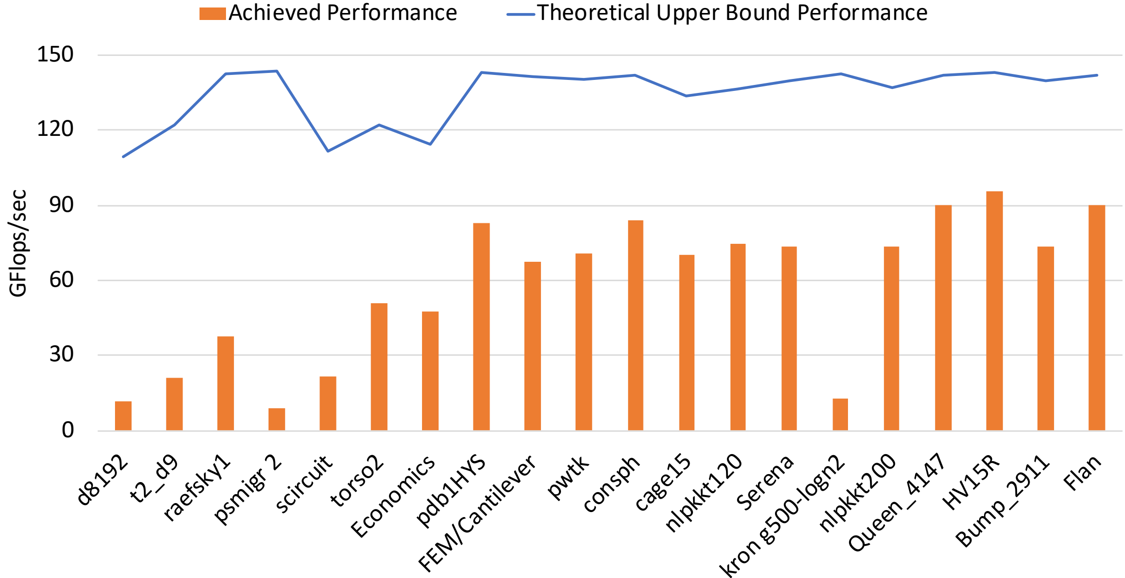


(a)



(b)

It is also important to note that the performance of multithreaded SpMV design is more predictable than GPU-based implementation. Knowing the NNZs and the available bandwidth, the performance of an FPGA-based SpMV can be easily deduced from the



(c)

Figure 5.7: Comparison between the achieved and the theoretical upper bound performance for (a) multithreaded SpMV (b) Tesla K40 (c) Titan GV100 .

observations made in Section 5.3.2. However, the same is not true for GPU implementation, where the cuSPARSE library implementation is unknown to the user.

We also compare the DRAM bandwidth utilization of each of these architectures in Figure 5.8. This experiment is particularly interesting because it conveys different information about each of these architectures. Higher DRAM bandwidth utilization for the FPGA-based SpMV imply that memory channels are utilized efficiently. Overall, multithreaded SpMV implementation achieves higher bandwidth utilization across all matrices. It is particularly more efficient on very large matrices (*Serena, kron_g500_logn21, HV15R and Bump_2911*). On an average, FPGA-based SpMV achieves 51% of bandwidth utilization on *Group 1* and 85.7% on *Group 2.*

On the other hand, lower bandwidth utilization on GPU imply that caches are more effective and there is less traffic to DRAM. This behavior also indicate that the GPU

performance is also limited by the architectural constraints like non-coalesced memory access while fetching the input X vector and the thread divergence encountered while handling variable size rows. As shown in Figure 5.8, as the size of a matrix increases, we observe higher DRAM bandwidth utilization on GPUs. On an average K40 uses only 25.5% of the DRAM bandwidth on *Group 1* and 38.67% on *Group 2* and Titan utilizes 18.8% of the available DRAM bandwidth on *Group 1* and 47.65% on *Group 2*.

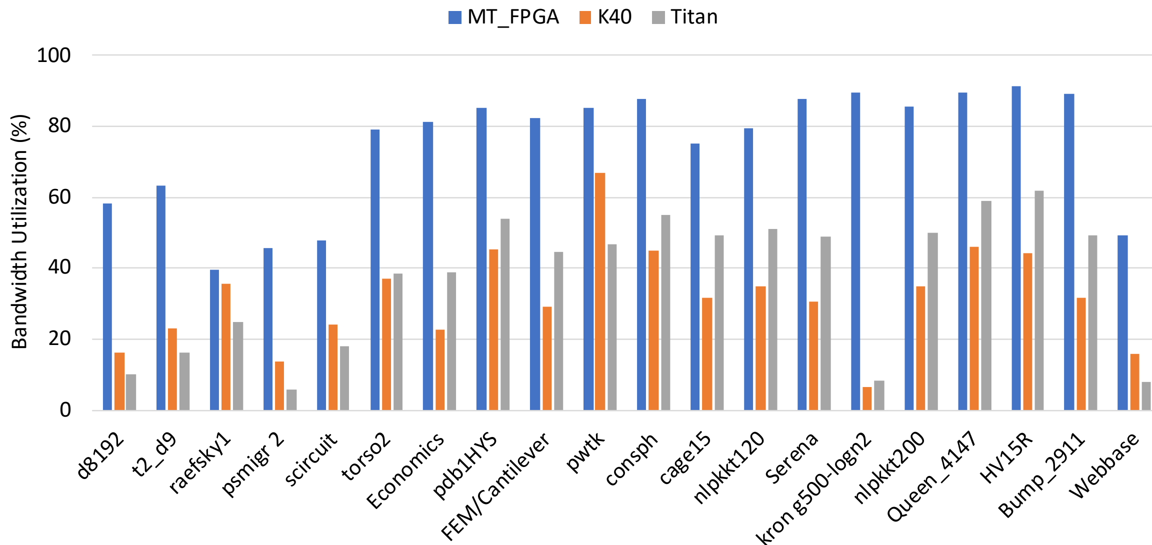


Figure 5.8: Comparison of bandwidth Utilization of multithreaded and GPU-based SpMV implementation

5.3.5 Scalability

An important issue that needs attention while parallelizing sparse applications is a good load balancing. A sparse matrix consists of variable length rows, leading to uneven distribution of non-zeros between partitions. As a consequence, many PEs may remain idle and the overall design may not scale, leading to a lot of unused hardware resources. In our

design, we apply dynamic scheduler that distributes partitions among PEs almost equally. This directly translates into a linearly scalable design.

Figure 5.9 presents the performance of our SpMV design on 2,4,8 and 16 PEs. It scales linearly with the number of PEs. This also imply that the scheduler is able to achieve fairly good load balancing between PEs. However, in smaller matrices where there are not enough partitions to distribute between PEs, the design will not scale.

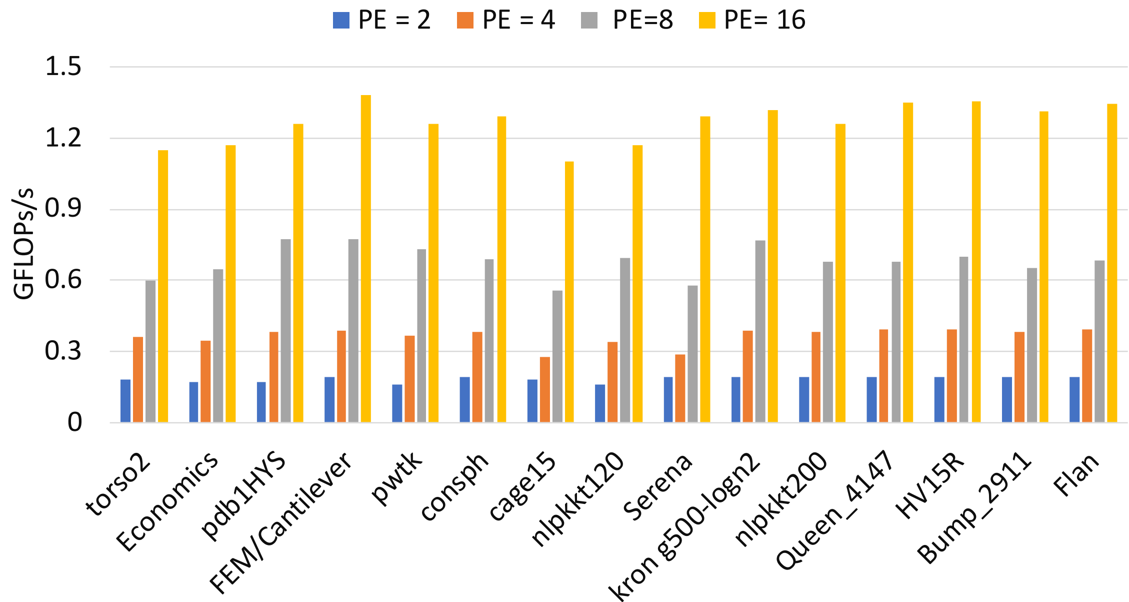


Figure 5.9: SpMV performance scaling with number of PEs

5.3.6 Existing FPGA-based SpMV Accelerators

Table 5.3 compares prior work on FPGA-based SpMV implementation to the proposed multithreaded SpMV. Because SpMV is a memory-bound operation, we consider memory bandwidth utilization as a more important metric than than peak performance alone. This metric also captures the overall efficiency of the architecture. Several of these

Table 5.3: Prior FPGA-based SpMV designs and implementation. '*' and '**' presents Group 1 and Group 2 data respectively.

	DRAM B/W Util				
Work	Average	Min-Max	Traversal	Storage	Comments
[81]	91%	91-96%	Column	Interleaved	Output Vector stored on-chip
[93]	76 %	64-98%	Blocked	Semi-Interleaved	Input Vector stored on-chip and replicated, restricts minimum non-zeros per row to be 8
[44]	*27.5% **35%	*21.7%-32.7% **37%-43.5%	Row	CSR	
[52]	14%	4-19%	Row	Custom CVBV	
[42]	71%	13-74%	Column	Custom SPAR	On-chip cache for x and y. Cache miss is handled by a hardware interrupt, leading to dead cycles.
MT SpMV	*51% **85.3%	*39-58 % **80-91%	Row	Partitioned COO	Suited for large matrices.

works use alternative representations to compress matrix data. Kestur et al. [52] utilize the bit-level manipulation capabilities of the FPGA to compress the matrix nonzero pattern using delta encoding. Our scheme does not include any compression or blocking and can be further enhanced with these techniques to offer higher throughput. Gregg et al. [42] use column major traversal and one DRAM chip per parallel processing. Zhang et al. [93] interleaves row-major values and column indices.

5.4 Conclusion

In this chapter, we used a Sparse Matrix Vector multiplication as a proof of concept to show that the multithreaded design can outperform cache based FPGA kernels and the GPU-based cuSPARSE implementation in terms of overall bandwidth utilization. Especially on larger matrices, multithreaded SpMV achieves as high as 91.17% and GPU achieves up to 69.96% of bandwidth utilization. Multithreaded design, unlike other FPGA-

based SpMV, does not make any assumptions on the size of an input or output vector and is therefore suited for more real world applications. Compared to other FPGA-based approaches, multithreaded design achives 82% bandwidth efficiency on the entire test suite. We also show that the multithreaded design scales linearly as long as the matrices are large enough (NNZ $>$ 1M for 16.2 GB/s) to saturate the memory bandwidth. Provided the scalable nature of our design, it can scale with the new technology and upcoming FPGA-based architecture with higher bandwidth.

Chapter 6

Conclusions

In this thesis, we show that multithreading as a FPGA paradigm is valid for irregular applications. Many real world applications like graph processing, query processing, sparse linear algebra etc are moving towards more random memory access. It will be difficult for the cache-centric hardware architectures to achieve maximum system performance. However, there is a push by the hardware industry towards heterogeneous architectures like Convey HC-2ex, Intel HARP and the latest Stratix10 MX integrated with HBM2. Many companies are actively focusing on FPGA accelerators that easily connect through PCIe, and provide APIs to easily integrate them with existing software. These developments allow researchers to prototype hardware designs without relying on caches to cope with long memory latencies. Because of the memory wall, the processor performance has outpaced memory performance, causing memory bandwidth to be the bottleneck for many applications

Instead of relying on the caches, multithreaded architectures, introduced in early 90's, focused on masking the memory latency. They perform well on applications with suf-

ficient parallelism, but identifying parallelism is a non-trivial task. Some chip manufactures like Oracle, with its UltraSPARC T series, have partially adopted this platform into their CPU designs. However, the emerging heterogeneous platforms can offer the best of both worlds. Caches work well for regular applications, but the irregular applications can be offloaded to accelerators with custom multithreaded kernels.

In this thesis, we introduce a custom hardware multithreaded execution that is an amalgamation of exiting multithreaded techniques (SMT and temporal). A custom hardware implementation requires a very small context that can be maintained on-chip on FPGA. Additionally, since the hardware design pipelined, it allows each thread to enter the pipeline every cycle. As a proof of concept we apply this technique to both type of irregularities: dataflow (group-by aggregation and SpMV) and control flow (selection operator).

In Chapter 3 we present a portable FPGA based in-memory aggregation algorithm for relational databases. The design uses custom CAM logic to enforce memory locking, and ensure synchronization while creating the hash table. Multithreading, along with multiplexing on memory channels can achieve upto 700 - 500 MTuples/sec depending on the dataset distribution and key cardinality, with a speedup up to 10x.

In Chapter 4 we present a detailed comparison study of the selection operator on multi-core CPU, GPU and multithreaded FPGA implementations. Multithreaded selection operation achieves a speedup between 1.4x - 4.6x over CPU SIMD and 1.4x - 6.7x over GPU implementations for the query selectivity that ranges from 0% to 50% (77% of TPC-

H queries). Unlike other architectures, multithreaded selection is independent of the data layout (row or column) and is very well suited for transactional workloads.

In Chapter 5, we explored multithreading on a well known sparse application, Sparse Matrix Vector Multiplication (SpMV). We theoretically derive the peak performance of our SpMV kernel and compare it to our experimental results. We achieve 65% - 95% accuracy depending on the NNZ in a matrix. We also discuss a heuristic to find the least NNZs to saturate the available bandwidth, above which our multithreaded kernel achieves maximum performance. This kernel is highly scalable and achieves as high as 93% bandwidth utilization. The results are compared against a widely used cuSPARSE library on K40 and the Titan GV100 GPU. The absolute GPU performance is higher than FPGA-based SpMV across all matrices. This behavior is a direct consequence of high bandwidth available to GPUs (Titan: 870 GB/s, K40: 288 GB/s Vs FPGA: 16.2 GB/s). Regardless, multithreaded SpMV achieves up to 95% of its upper bound performance where GPU achieves up to 69% of its peak.

Bibliography

[1] Cray. http://www.sdsc.edu/~allans/cug.html.

[2] Kickfire. http://www.teradata.com/.

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

[4] Nvidia. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[5] (TPC). TPC-H Benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

[6] 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.

[7] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *International Confonference on Supercomputing*, Supercomputing '92, pages 188–197, 1992.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *International Conference on Supercomputing*, Supercomputing '90, pages 1–6, 1990.

[9] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. On the performance and energy efficiency of sparse linear algebra on gpus. *The International Journal of High Performance Computing Applications*, 31(5):375–390, 2017.

[10] Peter Bakkum and Srimat Chakradhar. Efficient data management for gpu databases. *High Performance Computing on Graphics Processing Units*, 2012.

[11] 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.

[12] 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.

[13] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast data stream algorithms using associative memories. SIGMOD'07, pages 247–256, 2007.

[14] Ronald Barber, Guy Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1246–1252. IEEE, 2015.

[15] Andrew Bean. *Improving memory access performance for irregular algorithms in heterogeneous CPU/FPGA systems*. PhD thesis, Imperial College London, 2016.

[16] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, Nov 2009.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] David Broneske, Andreas Meister, and Gunter Saake. Hardware-sensitive scan operator variants for compiled selection pipelines. In *BTW*, pages 403–412, 2017.

[23] 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.

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

[25] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. VLDB'07, pages 339–350, 2007.

[26] John Cieslewicz and Kenneth A. Ross. Data Partitioning on Chip Multiprocessors. DaMoN'08, pages 25–34, 2008.

[27] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[28] John D. Davis and Eric S. Chung. Spmv : A memory-bound application on the gpu stuck between a rock and a hard place. 2012.

[29] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[30] Michael deLorimier and André DeHon. Floating-point sparse matrix-vector multiply for fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 75–85, New York, NY, USA, 2005. ACM.

[31] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28, April 2013.

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

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

[34] 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.

[35] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, and Jin Wang. Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology, 2012.

[36] Amr El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang, and George A. Mihaila. Column-oriented query processing for row stores. In *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*, DOLAP '11, pages 67–74, New York, NY, USA, 2011. ACM.

[37] 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.

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

[39] Pedram Ghodsnia et al. An in-GPU-memory column-oriented database for processing analytical workloads. In *The VLDB PhD Workshop. VLDB Endowment*, volume 1, 2012.

[40] 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.

[41] 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.

[42] 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.

[43] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. FPGA-based multithreading for in-memory hash joins. In *7th Biennial Conference on Innovative Data Systems Research (CIDR 15)*, 2015.

[44] Robert J. Halstead and Walid A. Najjar. Compiled multithreaded data paths on fpgas for dynamic workloads. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 1–10, 2013.

[45] Robert Joseph Halstead. *Using Multithreaded Techniques to Mask Memory Latency on FPGA Accelerators.* PhD thesis, UNIVERSITY OF CALIFORNIA RIVERSIDE, 2015.

[46] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[47] 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.

[48] Cray Inc. Cray xmt. `http://www.craysupercomputers.com/downloads/crayxmt/crayxmt_datasheet.pdf`.

[49] S. Jain-Mendon and R. Sass. A case study of streaming storage format for sparse matrices. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–6, Dec 2012.

[50] Chris Jesshope, Mike Lankamp, and Li Zhang. Evaluating cmps and their memory architecture. In *International Conference on Architecture of Computing Systems*, pages 246–257. Springer, 2009.

[51] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[52] S. Kestur, J. D. Davis, and E. S. Chung. Towards a universal fpga matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, April 2012.

[53] 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.

[54] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar. Next generation intel x00ae; micro-architecture (nehalem) clocking architecture. In *2008 IEEE Symposium on VLSI Circuits*, pages 62–63, June 2008.

[55] D. Langr and P. Tvrdk. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):428–440, Feb 2016.

[56] C. Y. Lin, H. K. H. So, and P. H. W. Leong. A model for matrix multiplication performance on fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 305–310, Sept 2011.

[57] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[58] Y. Liu and B. Schmidt. Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 82–89, July 2015.

[59] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[60] 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.

[61] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. VLDB'09, pages 229–240, 2009.

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

[63] K. K. Nagar and J. D. Bakos. A sparse matrix personality for the convey hc-1. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8, May 2011.

[64] NVIDIA. *CUDA CUSPARSE Library*, August 2010.

[65] Meikel Poess and Raghunath Othayoth Nambiar. Energy cost, the key challenge of today's data centers: A power consumption analysis of tpc-c results. *Proc. VLDB Endow.*, 1(2):1229–1240, August 2008.

[66] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1493–1508, New York, New York, USA, 2015. ACM Press.

[67] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware - DaMoN '14*, pages 1–6, New York, New York, USA, 2014. ACM Press.

[68] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 109–120, New York, NY, USA, 2002. ACM.

[69] 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.

[70] Simone Secchi, Antonino Tumeo, and Oreste Villa. A bandwidth-optimized multi-core architecture for irregular applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 580–587. IEEE, 2012.

[71] Nikita Shamgunov. The MemSQL in-memory database system. In *IMDM@ VLDB*, 2014.

[72] Yi Shan, Tianji Wu, Yu Wang, Bo Wang, Zilong Wang, Ningyi Xu, and Huazhong Yang. Fpga and gpu implementation of large scale spmv. In *Proceedings of the 2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, SASP '10, pages 64–70, Washington, DC, USA, 2010. IEEE Computer Society.

[73] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, Jan 2015.

[74] Evangelia A. Sitaridi and Kenneth A. Ross. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware - DaMoN '13*, page 1, New York, New York, USA, 2013. ACM Press.

[75] 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.

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

[77] 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.

[78] Kevin Townsend. *Computing SpMV on FPGAs*. PhD dissertation, Iowa State University, 2016.

[79] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 8. ACM, 2013.

[80] Antonino Tumeo, Simone Secchi, and Oreste Villa. Designing next-generation massively multithreaded architectures for irregular applications. *Computer*, 45(8):53–61, 2012.

[81] Y. Umuroglu and M. Jahre. An energy efficient column-major backend for fpga spmv accelerators. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 432–439, Oct 2014.

[82] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[83] 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.

[84] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009.

[85] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[86] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[87] 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.

[88] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[89] Q. Xu, H. Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149, Oct 2014.

[90] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. DaMoN'11, pages 1–9, 2011.

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

[92] 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.

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

[94] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[95] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 63–74, New York, NY, USA, 2005. ACM.

[96] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. ACM.