High-Performance and Energy-Efficient Irregular Graph Processing on GPU architectures

Albert Segura Salvador

Doctor of Philosophy
Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors:
Jose-Maria Arnau
Antonio González

July, 2020

Abstract

Graph processing is an established and prominent domain that is the foundation of new emerging applications in areas such as Data Analytics, Big Data and Machine Learning. Applications such as road navigational systems, recommendation systems, social networks, Automatic Speech Recognition (ASR) and many others are illustrative cases of graph-based datasets and workloads. Demand for higher processing of large graph-based workloads is expected to rise due to nowadays trends towards increased data generation and gathering, higher inter-connectivity and inter-linkage, and in general a further knowledge-based society. An increased demand that poses challenges to current and future graph processing architectures.

To effectively perform graph processing, the large amount of data employed in these domains requires high throughput architectures such as GPGPU. Although the processing of large graph-based workloads exhibits a high degree of parallelism, the memory access patterns tend to be highly irregular, leading to poor GPGPU efficiency due to memory divergence. Graph datasets are sparse, highly unpredictable and unstructured which causes the irregular access patterns and low computation per data ratio, further lowering GPU utilization. The purpose of this thesis is to characterize the bottlenecks and limitations of irregular graph processing on GPGPU architectures in order to propose architectural improvements and extensions that deliver improved performance, energy efficiency and overall increased GPGPU efficiency and utilization.

In order to ameliorate these issues, GPGPU graph applications perform stream compaction operations which process the subset of active nodes/edges so subsequent steps work on compacted dataset. Although this optimization is effective, we show that GPGPU architectures are inefficient at performing stream compaction due to the data movements it entails, and end up representing a significant part of the execution time. After identifying these issues, we propose to offload this task to a programmable Stream Compaction Unit (SCU) hardware extension tailored to the requirements of these operations, while the remaining steps of the graph-based algorithm are efficiently executed on the GPU cores. The SCU is a small unit tightly integrated in the GPU that efficiently gathers the active nodes/edges into a compacted array in memory. We further extend the SCU to perform pre-processing of the data by filtering and reordering elements processed. Besides the benefits achieved with offloading, this pre-processed SCU-prepared data improves GPU efficiency by reducing workload and achieving larger memory coalescing. We evaluate our SCU design with a wide variety of state-of-the-art graph applications and extended GPGPU architectures. The results show that for High-Performance and for Low-Power GPU systems the SCU achieves speedups of 1.37x and 2.32x, 84.7% and 69% energy savings, at the cost of a small area increase of 3.3% and 4.1% respectively.

Memory divergence remains an important downside for irregular applications which struggle to fully exploit GPGPU performance. Although memory divergence can be improved by carefully considering architecture features and devoting significant programmer effort to modify algorithms with complex optimization techniques, it is far from an ideal solution which in the end shifts

programmers priorities. We show that in graph-based GPGPU irregular applications these inefficiencies prevail, yet we find that it is possible to relax the strict relationship between thread and data processed to empower new optimizations. Based on this key idea, we propose the Irregular accesses Reorder Unit (IRU). The IRU is a novel hardware extension tightly integrated in the GPGPU pipeline that reorders data processed by the threads on irregular accesses which significantly improves memory coalescing. Additionally, the IRU is capable of filtering and merging duplicated irregular accesses which reduces the application workload. These optimizations improve overall memory hierarchy and deliver increased performance and energy efficiency. Programmers can easily utilize the IRU with a simple API, or compiler optimized generated code with the extended ISA instructions provided. We evaluate our proposal for a wide variety of state-of-the-art graph-based algorithms. The IRU achieves a memory coalescing improvement of 1.32x and a 46% reduction in the overall traffic in the memory hierarchy, which results in 1.33x and 13% improvement in performance and energy consumption respectively, while incurring in a 5.6% area overhead.

The increased need for greater data processing efficiency establishes high-throughput architectures, such as GPGPUs, as crucial devices in order enable efficient graph processing. Although efficient GPGPU graph processing faces many challenges, in our previous contributions we separately explore stream compaction offloading and improved coalescing of irregular access to ameliorate graph-based workload efficiency. While the SCU achieves significant speedups and large energy savings, it incurs in high contention in the Network-on-Chip (NoC). On the other hand, the IRU achieves more modest energy savings, but its optimizations are highly efficient at reducing contention in the NoC, showing up to a factor of 46% lower NoC traffic. We find that it is possible to leverage the strengths of both approaches to achieve synergistic performance improvements and higher graph processing efficiency. We do so by proposing a new unit, the IRU-enhanced SCU (ISCU). The ISCU employs the efficient mechanisms of the IRU to improve SCU stream compaction efficiency and throughput limitations. We first characterize the bottlenecks of our SCU contribution concluding that its major limitation is the large NoC traffic due to the pre-processing operations, i.e. filtering and grouping. With this insight we realize we can leverage the IRU hardware utilizing its strengths to efficiently perform the pre-processing. Consequently, the ISCU leverages both the powerful stream compaction offloading achieved by the SCU and the efficient filtering mechanism of the IRU employed to deliver pre-processing optimizations. We evaluate the ISCU for a wide variety of state-of-the-art graph-based algorithms and applications. Our final proposal achieves a 2.2x performance improvement and 90% energy savings which enable a 10x increase in energy-efficiency, improvements derived from a high reduction of 78% memory accesses, while incurring in 8.5% area overhead.

Keywords

GPGPU, GPU, Graph Processing, Graph Exploration, Graph Applications, Data Analytics, Stream Compaction, Stream Compaction Unit, SCU, Energy-Efficient, High-Performance, Irregular Applications, Irregular Accesses, Irregular Graph Processing, GPGPU Graph Processing, Irregular accesses Reorder Unit, IRU, ISCU, IRU-Enhanced SCU, Workload Filtering, Reordering, Accelerator, Hardware Accelerator, Custom Hardware, Tailored Hardware, Programmability, Memory Divergence, Memory Coalescing, Memory Contention, Sparse Accesses, Branch Divergence, Control-Flow Divergence, Graphics Processing Unit, Computer Architecture.

Acknowledgements

First and foremost, I would like to thank my advisors Professor Antonio González and Dr. Jose-Maria Arnau for their invaluable guidance through out my PhD study and research, their patience, motivation and many teachings. I would like to recognize their invaluable assistance, feedback and their involvement in my day-to-day work, for which I feel very grateful.

I would like to thank my defense committee; Tor M. Aamodt, Jordi Tubella Murgadas and Julio Sahuquillo Borrás, for their valuable feedback and comments. It is an honor to have such prestigious committee.

I wish to thank all the members and lab-mates of ARCO (the "The D6ers") which I was lucky enough to share many moments during my PhD and be part of this amazing group of people. Special thanks to Gem Dot, Enrique de Lucas, Martí Torrents, Josue Quiroga with whom I shared the beggining of my PhD. I was lucky to share the majority of my stay at ARCO with Hamid Tabani, Reza Yazdani, Martí Anglada, Marc Riera, Franyell Silfa and Syeda Azmath thank you for all the good moments that we shared at UPC. Finally, I would like to wish good luck to the newcomers Dennis Pinto, Diya Joseph, Pedro Exenberger, Jorge Sierra, Raúl Taranco and Mehdi Hassanpour, I am sure will do an amazing work.

I would also like to thank university colleagues, campus mates and friends with whom I shared many years, projects, difficulties and successes over the course of my stay in the University. Special thanks to Pedro Benedicte, Constantino Gomez, Cristóbal Ortega and David Trilla (the "Cache Miss" group) for the many experiences shared. And thanks to the many others with which I have cross path during this endeavor.

Finally, I would like to thank close friends and my family for their unconditional support and encouragement. To my parents for their love and support enabling me from the very beginning to pursue my passion for computers, electronics and education. To my brother and sister for the encouragement and for enduring me. And to my grandparents and extended family for their love, support and endorsement that has undoubtedly led me to where I am today. Finally, the last words I have saved for my partner with whom I closely shared many joys over the duration of this thesis, but also without doubt her love, encouragement and support have helped me overcome the inescapable periods of hardship that a PhD thesis entails.

Declaration

I declare that the work contained in this thesis has not been submitted for any other degree or professional qualification except as specified. Some of the proposed techniques and results presented in this thesis has been published in the following papers:

- "SCU: A GPU Stream Compaction Unit for Graph Processing".
  Albert Segura, Jose-Maria Arnau, and Antonio González.
  International Symposium on Computer Architecture, June 2019 (**ISCA '19**).
  DOI: https://doi.org/10.1145/3307650.3322254

- "Irregular Accesses Reorder Unit: Improving GPGPU Memory Coalescing for Graph-Based Workloads".
  Albert Segura, Jose-Maria Arnau, and Antonio González.
  This work has been submitted for publication.

- "Energy-Efficient Stream Compaction Through Filtering and Coalescing Accesses in GPGPU Memory Partitions".
  Albert Segura, Jose-Maria Arnau, and Antonio González.
  This work has been submitted for publication.

Albert Segura Salvador

Contents

List of Figures

List of Tables

1 Introduction

This chapter reviews GPU architectures strengths and introduces current trends and challenges of graph processing on modern GPU architectures, thus establishing the motivation behind this work. It also explores the specific problems addressed in this thesis, reviews how they are approached by state-of-the-art works and finally introduces the novel proposals and contributions of this work.

1.1 Current Trends

Over the last decade, Graphic Processing Units (GPU) have produced a dramatic change in computer systems steaming from the increased demand for higher performance. Current GPU prevalence is mainly fueled by emerging workloads that exhibit high data parallelism. Big Data analytics [129], Neural Networks [57] and Deep Learning [82, 135, 168] are clear examples of extremely popular applications that benefit from the high degree of parallelism offered by GPUs. Other applications include the adoption of newer, more demanding video decoding and multimedia standards (i.e 4K and 8K resolutions [156]), the increased demand for graphics fidelity coming from the gaming industry as seen for Virtual Reality systems (VR) [158] and image detection [78], speech recognition [22] and, finally, the automotive industry with self-driving cars [16, 71]. These examples, among many other applications, have hugely increased the demand for high processing throughput and huge amounts of parallelism, both of which GPU systems have been able to provide by leveraging many simpler and lower frequency core counts, in contrast with single or few high performance core counts of traditional CPU systems. This approach has established GPUs as an essential and very successful component of modern computing systems.

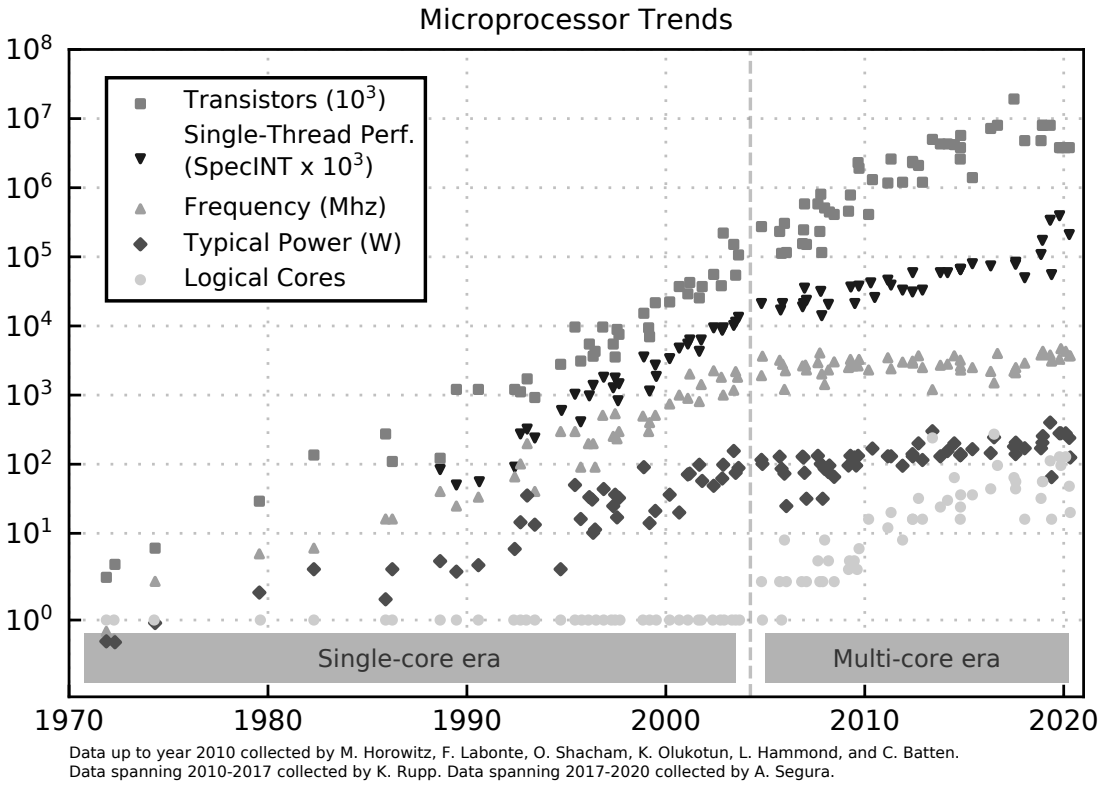


Figure 1.1: Trends in Microprocessor characteristics over the last decades [131]. Until the 2000s decade microprocessors transistors, single-thread performance (SpecINT), frequency and power increased steadily following Moore's Law and Dennard Scaling. Mid 2000s decade sees a clear shift in trends as increasing instruction-level parallelism (ILP) gets diminishing returns; frequency and power increase is halted while the increasing transistors numbers is dedicated to increase number of cores. Single-thread performance increase is reduced while systems provide higher performance by leveraging multi-threading.

Up to the 2000 decade, shrinking transistor technology nodes was a big contribution to delivering new computer systems with increased single core performance. Moore's Law, formalized by 1965 [101] and reformulated by 1975 [102], predicted the doubling of transistor density in new semiconductor systems every two years. Meanwhile, Dennard scaling [32], formulated by 1974, predicted power density to stay constant with new transistor nodes: by halving area, every new iteration would allow for 30% reduced circuit delays and reduced voltages, which would result in equal power dissipation with double the number of transistors yet allowing a 1.4x increase in frequency. Combined, Moore's Law and Dennard scaling would doubled computer chip performance every 18 months, a rule which drove industry forward for many decades.

In practice, many factors limited the scaling down of area, circuit delays and voltages. Area reductions where constrained due to issue logic and caches not scaling linearly with area in addition to the increase number of wires of newer computer designs [49]. Circuit delays reductions were not as aggressive as predicted by Dennard scaling, in part due to the increased number of wires [60]. Finally, voltages did not scale accordingly to Dennard scaling instead

decreasing by 15% every two years [49], consequently increasing power density. Overall, the performance increase observed up to the 2000 decade as seen in Figure 1.1 was not solely achieved by technology node scaling but in addition to computer architecture innovations. Novel innovations such as new deeper pipelines, improved branch predictors, out-of-order execution, new instruction set architecture (ISA), and improved memory organizations among many others were a huge contribution in performance improvements by reducing latency and improving instruction-level parallelism (ILP). Meanwhile, many new power-aware techniques such as clock gating [163], power gating [65] and the inclusion of dark silicon [36] reduced the amount of switching transistors which kept power density constrained [49].

By mid 2000's decade, architectural improvements reached a point of diminishing returns as increased frequencies drove power leakage up, thus increasing overall power dissipation. Consequently, the steady frequency increase on newer systems was halted, yet transistor counts kept increasing every year, as it can be seen in Figure 1.1. This meant that computer architects had at their hands more transistors without any way to fully utilize them all on single core systems. This disparity prompted the exploitation of thread-level parallelism (TLP) techniques, driving industry to design multi-core and multi-threaded systems with increasing CPU core and thread counts, while relying on parallelization to deliver performance improvements [150]. Nonetheless, parallelization is a complex problem. Amdahl's law [3, 59] dictates the maximum performance speedup achievable in relation with the parallelizable section of a program, thus revealing that to achieve significant speedups a large portion of the application has to be parallelizable, as depicted in Figure 1.2. Additionally, CPUs were not primarily optimized for parallelization. CPU systems were mainly optimized for single thread performance with complex execution units and pipeline optimizations, and low latency execution thanks to an extensive cache hierarchy. The end of single thread performance ILP scaling ushered in the popularization of programmable GPU architectures, or General Purpose GPU (GPGPU) [46], which in contrast to CPUs heavily focused on parallelization by providing huge core counts with lower performance, leveraging aggressive multi-threading (i.e. TLP) to tolerate main memory latency.

Meanwhile, the memory system had a different evolution. Technology improvements brought higher DRAM density thus facilitating capacity increases at a similar pace to Moore's Law [143], nonetheless memory performance did not scale accordingly. Despite technology scaling facilitating frequency increases, DRAM memory latency was largely dominated by wire delays; as such latency reductions depend on scaling down circuit delays, reductions which felt short on the Dennard scaling predictions, as discussed previously. Consequently, an increasing disparity emerged between computing and memory performance scaling, this disparity lead to the increased relevance of memory latency as the principal system performance limitation which had computers hitting the Memory Wall [165]. Over the years, architectural and technological innovations have persistently attempted to close this disparity and thus push the memory wall further into the future, despite the fundamental technology scaling limitations. More sophisticated caches and prefetchers, latency hiding techniques, deeper memory hierarchies, improved memory controllers, and 3D stacking [123] together with processing in memory [103] are among the constant stream of architectural innovations limiting memory latency negative performance impact. While multi-core CPU designs have invested significantly on these increasingly sophisticated memory improvements, GPGPU architectures leverage their vast core counts and unparalleled TLP to hide memory latency, thus pushing forward the memory wall performance limitations.

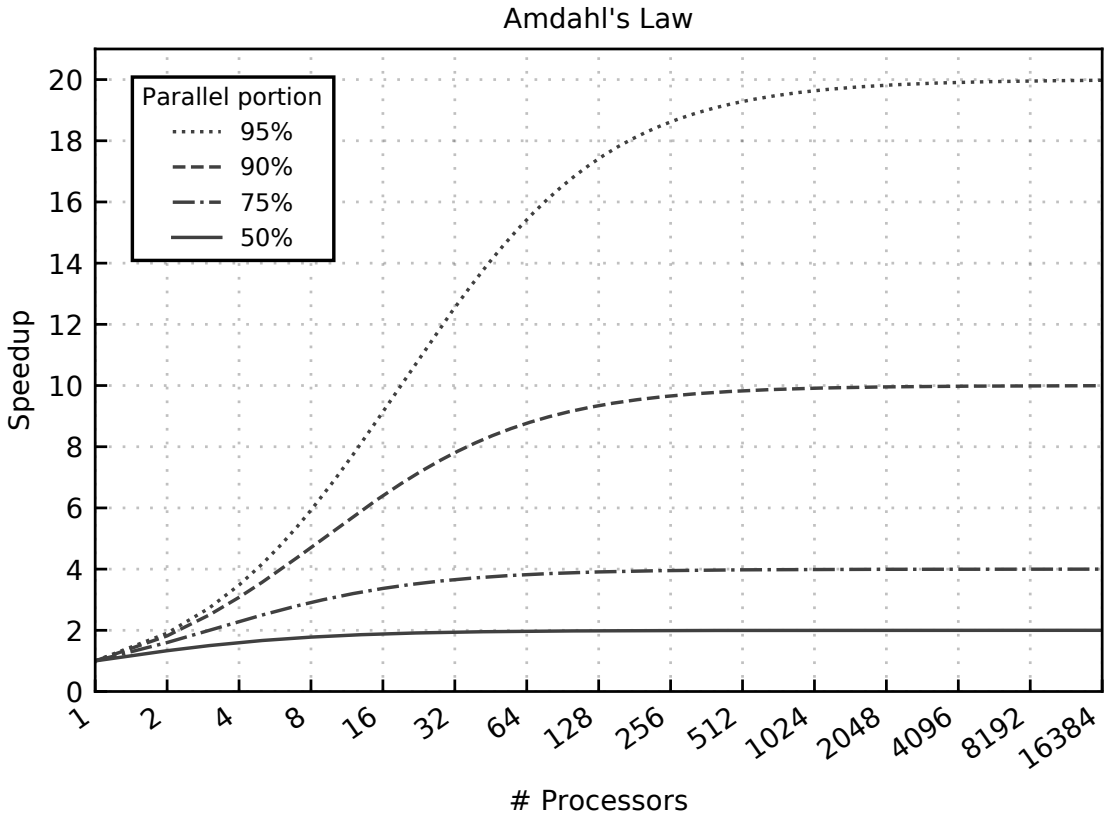


Figure 1.2: Maximum achievable performance speedup of a parallel application in increasingly parallel systems dictated by Amdahl's law [3, 59]. To achieve high speedup by means of parallel execution, a very high percentage of a program has to be parallel: even a high 95% parallel application maximum speedup does not exceed 20x. This clearly showcases the limitations of improving architecture performance by means of increased number of processors.

Over the last decade Moore's Law has started to slow down [155, 31]; industry manufacturers have been experiencing recurrent delays [64] of their production of newer technology nodes due to complexity, unreliability and overall increased cost of already few atoms-wide transistors. Further increased costs and complexity required to improve performance by shrinking the technology will put further pressure on computer architects to come up with new architectures and increasingly specialized hardware [31] as means to deliver performance improvements. The multi-core era and the GPGPU specialization ended up being very effective to drive further computer system performance increases; many GPGPU parallel applications [97] significantly outperform counterpart applications in traditional CPU systems which has facilitated new research domains and applications such as Deep Learning [100] and Autonomous Driving [15]. The prevalence of GPGPU has transformed traditional systems in more heterogeneous and specialized computing systems. A shift, which GPGPU architectures started, that has paved the way towards more diverse and specialized hardware, with the current inclusion of accelerators for specific applications and future widespread specialized hardware.

1.1.1 GPGPU Popularization

GPGPU architectures have managed to achieve broad adoption in the last decade despite being novel architectures which require considerable knowledge of the underlying hardware and new programming models to achieve high performance. It is interesting to take a look at their evolution over the years to understand the design choices and trade-offs that led to current GPGPU architectures.

Early GPU or video cards go way back in time to the beginning of the computer architecture, with contemporary designs first appearing in the 1990 decade. Early GPUs consisted of fixed function units for graphic operations which provided hardware acceleration to render 2D and 3D graphics for games and graphical tools. Prominent examples can be found in the console gaming industry, such as the Reality Coprocessor (RCP) of the Nintendo 64 [104] and the Sony GPU for the PlayStation [120], which first coined the term. On the other hand, the 3dfx Voodoo video card [1] is an example of early GPU in the PC market. The beginning of the 2000 decade saw the appearance of programmable GPUs together with graphical pipeline toolkits such as OpenGL [114] and DirectX [34]. In turn, this enabled the GPU to perform linear algebra, crudely implemented before by mapping data into textures and applying shader programs [79]. At the time, GPUs featured hardware acceleration closely correlated with the different stages of the rendering pipeline with the notable addition of programmable pixel and vertex shader units first introduced by the NVIDIA GeForce 3 NV20 [43]. This early programmability allowed simple pre-processing of pixels and vertices, but computer architects had to carefully balance the execution time and bottlenecks of the different stages of the graphics pipeline. The inclusion of newer stages in the DirectX and OpenGL pipelines marked a dead end of this fixed-function pipeline approach [133].

NVIDIA released its first contemporary GPGPU system by late 2006 with the Tesla microarchitecture, the first release of their programmable unified shader architecture [153]. It was first featured in the NVIDIA 8800 GTX (Tesla G80) [44] which introduced a redesigned SIMT pipeline which is the base design of today NVIDIA GPGPU architectures. Similar unified shader architecture was promptly adopted by competitors such as the ATI Radeon HD 2000 [122] by 2007. This programmable unified shader replaced previous stages as it was capable of processing vertex, fragments and geometry kernels alike. Additionally, the first release of the Compute Unified Device Architecture (CUDA) [25] toolkit facilitated programmability for general purpose applications, enabling computing on an impressive 96 unified shader (Executing Units or Cores in NVIDIA terminology), specially compared to poor parallelism of CPUs at the time.

NVIDIA SIMT architecture consists of a number of Streaming Multiprocessors (SM) with a private L1 data cache and a scratchpad memory (Shared Memory), while the SMs in a GPU share an L2 cache. Each SM includes a number of cores (i.e. execution units), typically in multiples of 32, and execute instructions in lock-step in groups of 32 threads, i.e. a warp in NVIDIA terminology. GPGPU SM are typically clocked at a lower frequency and are less complex than counterpart CPU designs since they are designed with simpler in-order instruction execution pipeline. GPGPU architectures leverage thread-level parallelism (TLP) and memory-level parallelism (MLP) which are provided by the cores and the huge memory bandwidth of the architecture and the memory respectively, and counter the lower performing cores and higher memory latency.

GPGPU architectures specific design choices and reliance on thread and memory level parallelism limits the set of applications fit for GPGPU acceleration and reveal some inefficiencies of the architecture in particular scenarios. Since warp's threads are executed in lock-step, they progress simultaneously through the pipeline and execution units. Consequently to implement branch behaviour, threads in a warp can be activated and deactivated with the use of predication registers and a divergence SIMT stack. Thread deactivation becomes an important source of inefficiencies referenced as branch divergence issues which limits the TLP . Divergence on performing memory accesses are another major source of inefficiencies that arise as the consequence of un-collocated memory accesses performed by the threads in a warp which limits both thread and memory level parallelism.

Over the 2010 decade GPGPU architectures have increased performance and energy efficiency without deviating significantly from the SIMT unified shader pipeline, while incrementally incorporating new features in each new architecture. One of the initial additions was the upgrade to 32-bit executing units from previous 24-bit ones, a relic of a past as 24-bit were enough to process RGB pixels. Over time, half, single and double floating point made its way into the cores, as well as improved atomic operations with the aim to support more complex and general purpose applications. More recently, the addition of Tensor Cores have allowed fast Matrix Multiply and Accumulate operations for Neural Network applications. Overall, each new architecture has brought improved performance and energy efficiency while accommodating higher number of cores and parallelism, reaching 4608 CUDA cores for the NVIDIA Titan RTX [42] released late 2018.

GPGPU adoption has been very broad from personal computers, to gaming consoles, data centers, high-performance computing (HPC), supercomputers, as well as mobile platforms and self-driving cars. Functionality additions such as floating point operations allowed GPGPUs to be used for algebra computations and science field computations such as fluid dynamics[55], particle docking [97] and others [10, 87]. By November 2019, nearly 40% of the total compute performance of the TOP500 supercomputers list (626 petaflops) came from GPU-accelerated systems [72]. The high parallelism delivered has improved processing of Big Data applications, data base operations[7], and large graph processing[161] as well as Neural Networks [57] and Deep Learning models [82, 135]. The latter addition of Tensor Cores greatly improves training and inference of deep learning applications and neural networks, in turn improving image, vision and speech recognition [125], for applications such as image classification [35], automatic speech recognition (ASR) [136] or audio de-noising [110]. Finally GPGPU architectures have also found a place in mobile systems such as the Nintendo Switch [105] with a NVIDIA Tegra X1 [111] or in self-driving cars such as the NVIDIA Tegra X2 [111] and Pascal GPGPU found, up to 2014, in Model S cars[108, 154] from the Tesla manufacturer.

1.1.2 GPGPU Graph Processing

World data generation and total global data size is increasing exponentially. The International Data Group (IDG) [66] predicts a total of 175 ZB for 2025, an increase of 3.5x in five years [124], which is likely to continue in the future. In today's and future knowledge-based and highly interconnected society, a huge range of devices and new technologies are likely to generate this data. The advent of Internet-of-Things (IoT) [5], and the new implementation of 5G mobile

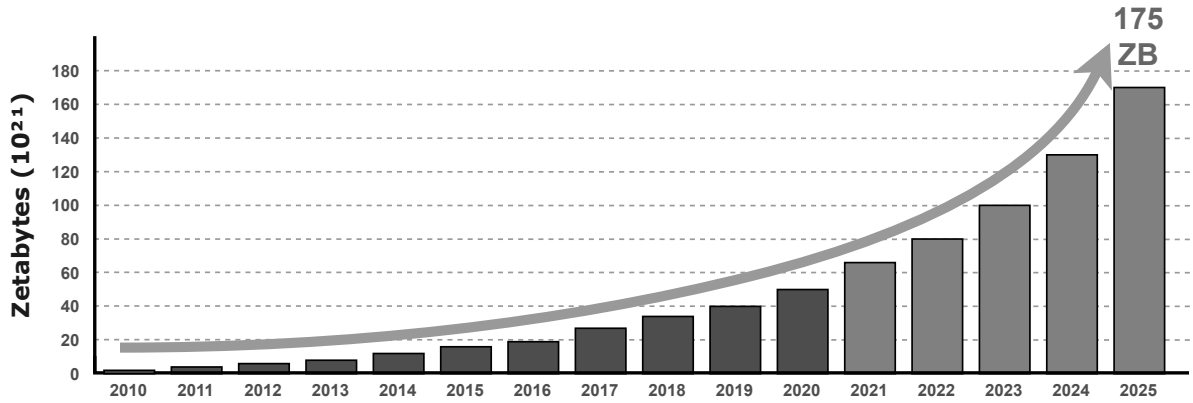


Figure 1.3: Global worldwide growth of data predicted by the IDC over the next years [124]. The depicted trend indicates that the increase in worldwide data is exponential, consequently as capacity increases we will need higher performance systems to be able to use the data to process and analyze it.

networks will bring huge amounts of interconnected smart devices generating real-time data to be processed and analyzed. New applications such as small sensor devices collecting health data, monitorization of smart home appliances, streaming-focused applications such as streaming gaming platforms, smart city devices and autonomous interconnected cars are some examples among many new devices and applications yet to come.

Large graph processing is crucial for data analytics as graph datasets allow for efficient data and relationship representation and facilitate many analytics applications. Graph datasets are employed in a broad set of applications, including social networks, Big Data analytics, web classification and ranking, recommendation systems, real-world navigation and pathfinding, elements in unstructured meshes, biologic and genetics sequencing and control of viral disease spreading.

GPGPU architectures allow for fast data processing and data analytics in graph datasets since they are capable of processing in parallel huge amounts of data as well as train deep learning models efficiently. Due to graph datasets large amount of independent element processing, GPGPU architectures allow for fast processing due to their high amount of parallelization and throughput in contrast to regular CPU architectures. Graph data representation tends to be irregular and unstructured and so efficient resource utilization on GPGPU architectures is not trivial, which has led to the introduction of graph parallelization frameworks such as NVIDIA nvGRAPH [107] and Gunrock [161], which facilitate programming efficient graph-based applications for GPUs.

Efficient Graph Processing and Analytics remains a hot topic in the research community and industry due to GPUs many inefficiencies at processing graph-based applications, and will likely continue to do so as our world generates increasing amounts of data, requiring improved data processing capabilities. Despite GPGPU architectures shortcomings, their throughput and parallelization capabilities enable fast data processing for graph applications and will continue to do so in the foreseeable future.

1.2 Problem Statement

GPGPU architectures provide high performance for highly parallel applications. Applications showing regular execution and regular memory access patterns are able to efficiently utilize GPGPU architectures, whereas more irregular applications that benefit from high parallelization struggle to achieve high efficiency and utilization. Graph-based applications can easily benefit from parallel exploration of graph datasets, but experience irregular execution due to the characteristics of both graph datasets and graph algorithms: low computation to memory access ratio [9], data-driven workloads, unstructured and irregular datasets and poor data locality. This section explores the consequences of the irregular execution of graph-based applications on GPGPU architectures, which defines the problem statement and optimization targets of this thesis. We explore in detail two main issues: Memory Divergence 1.2.1 and Workload Duplication 1.2.2.

1.2.1 Memory Divergence

Graph-based applications experience significant memory divergence on GPGPU architectures. High memory divergence occurs when the threads in a wave-front or warp have un-collocated memory accesses, i.e. individual threads within a warp access different cache lines, which negatively impacts performance. On the contrary, if all memory accesses are collocated, i.e. they access the same cache line, minimal divergence is achieved and the application experiences high memory coalescing. Graph applications notoriously experience significant memory divergence, due to characteristics of both graph datasets and algorithms. First, parallel graph traversal requires threads to traverse the graph data structures and, consequently, many irregular accesses are performed due to the unstructured and irregular nature of the data represented in a graph, which identifies relationships between elements. Second, graph exploration shows poor data locality for several reasons. First, the amount of data represented by graph datasets largely exceeds the capacity in the caches. Second, due to connectivity of the graph, node connections might not be visited again, leading to poor temporal locality. Third, neighbor nodes in the graph can be stored at large distances in memory, leading to poor spatial locality. Figure 1.4 explores the memory coalescing on several datasets and graph algorithms. On average, the memory coalescing achieved is of 4 accesses per warp. As stated earlier, the dataset characteristics greatly influence the memory coalescing experienced.

Memory divergence has important performance implications for GPGPU systems and applications. First, memory divergence increases the saturation of the memory hierarchy architectural resources, increasing the utilization of the Load/Store (LD/ST) unit and the latency to issue all memory accesses of a warp instruction. Additionally, the increased accesses per warp instruction increase pressure on resources to handle misses on caches, such as miss status holding registers (MSHRs) and entries in the miss queue. This problem is aggravated by the fact that GPU L1 ratio of cache entries per thread is significantly lower, putting even more pressure on the caches which negatively impacts data locality, increasing contention and miss ratio. Aside from increased pressure to the data caches, the elevated miss ratio leads to increased use and contention of the Network-on-Chip (NoC), negatively impacting latency to access L2 cache. Second, high bandwidth is required to provide enough data to feed the functional units.

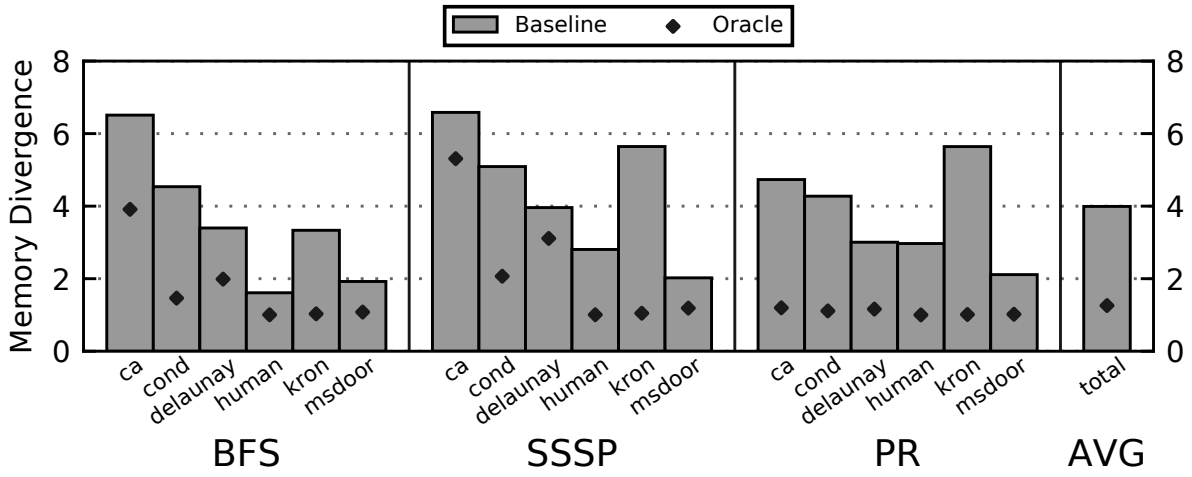


Figure 1.4: Memory coalescing over different benchmarks and applications for the Baseline and an Oracle. The memory coalescing is measured as the number of L1 cache accesses per warp-level memory instruction, the higher this value the worse the memory coalescing an application has. It ranges from 32 accesses to 1 access per warp instruction. The measured coalescing is represented by the Baseline, while the Oracle shows potential memory coalescing.

High memory divergence reduces the memory hierarchy capacity to deliver enough data to the functional units, which then are under utilized due to the pipeline stalls awaiting data. An increase of memory divergence from 1 to 2 accesses to service a warp doubles the memory pressure. In the worst case, no memory access is collocated, which requires an access per each thread in the warp (typically 32 threads). Consequently, memory divergence significantly increases memory contention compared to an application with regular access patterns. Finally, the above mentioned problems translate to higher energy cost. On one hand, the slowdown of the application increases static energy consumption while, on the other hand, more dynamic energy is required due to the increased activity throughout the memory hierarchy.

Graph-based applications are negatively impacted by the consequences of high memory divergence. Memory coalescing that results from the graph dataset and algorithm characteristics previously reviewed reaches 4 accesses per warp on average. Consequently, the high memory divergence has a negative impact on the memory hierarchy: an L1 data miss ratio of 83% on average, as well as a 64% L2 data miss ratio. The performance implications also translate outside of the memory hierarchy. The increased memory hierarchy congestion due to the divergence aggravated by the low computation to data ratio of the graph-based application results in significant stalls in the functional units leading to low issue rate efficiency of 14% and 228.8 out of a maximum of 2048 Instructions per Cycle (IPC).

Data compaction algorithms are employed by graph-based applications to reduce and limit the amount of memory divergence when traversing a graph dataset. The use of data compaction involves adding a previous step to the graph exploration in which the data to be accessed is previously read and written to a single compacted data array, then subsequent accesses on the compacted dataset exhibit a more regular memory access pattern with improved data locality

and increasing bandwidth utilization. This additional step has a significant execution time overhead, yet overall results in a positive effect due to the improvement of memory accesses. Nonetheless, GPU architectures are not well suited for data compaction for several reasons. First, data compaction consists of sparse memory accesses with poor locality that fetch the elements to be compacted. Sparse memory accesses result in very low memory coalescing, producing intra-warp memory divergence and reducing GPU efficiency by a large extent. Second, data compaction has an extremely low computation to memory access ratio, as it primarily consists of load and store instructions to move data around in main memory, consequently the functional units are underutilized during data compaction operations. Finally, the parallelization of data compaction algorithms incurs in costly synchronization overheads. These reasons lead us to propose hardware techniques to perform data compaction operations instead of performing them by software on GPU architectures.

Finally, although memory divergence is high on graph-based applications, we observe potential to improve the memory coalescing. Figure 1.4 shows baseline memory coalescing against potential evaluated memory coalescing. The potential memory coalescing is measured with a simple oracle which sorts in increasing order the addresses generated and computes the resulting memory coalescing. This approach is not optimal but provides a good approximation. Figure 1.4 indicates that memory coalescing can improve on average from 4 to 1.25 accesses per warp which would alleviate the memory hierarchy congestion.

1.2.2 Workload Duplication

GPGPU graph-based applications leverage the parallelism available to process in parallel several elements of the graph. Graph exploration on GPGPU architectures is typically implemented with frontiers (i.e nodes or edges frontiers), which are elements that are active or to be processed on a given iteration of the algorithm, after they have been processed their adjacent elements are inserted in the new frontier to be processed. The parallel nature of graph processing allows for duplicated workload to occur. Several threads might be processing edges which lead to the same adjacent node, or they might be processing duplicated elements in a frontier. This work might often be benign, thus not altering the final computation, but might incur in significant workload overheads: a given duplicated element explored might generate duplicated workload itself, which can quickly increase the application workload exponentially. Many factors contribute to increase the potential of workload duplication, such as the connectivity of the elements in the graph, the specific graph dataset characteristics and the properties of the graph exploration algorithm evaluated.

In order to achieve efficient GPGPU performance and avoid to exponentially increase the workload, a mechanism to detect and suppress this workload duplication has to be employed. Nonetheless, to properly detect and eliminate workload duplication is a non-trivial task on parallel architectures. Either costly synchronization mechanisms have to be used which add significant overheads to the execution, as every thread is required to evaluate its data uniqueness or, otherwise, imprecise best-effort mechanisms are used which avoid synchronization overheads while providing partial detection and elimination of duplicated elements. Each of these software mechanisms has its own trade-offs that the application has to carefully balance in order to execute efficiently. Additionally, elimination of duplicated elements has an impact on branch

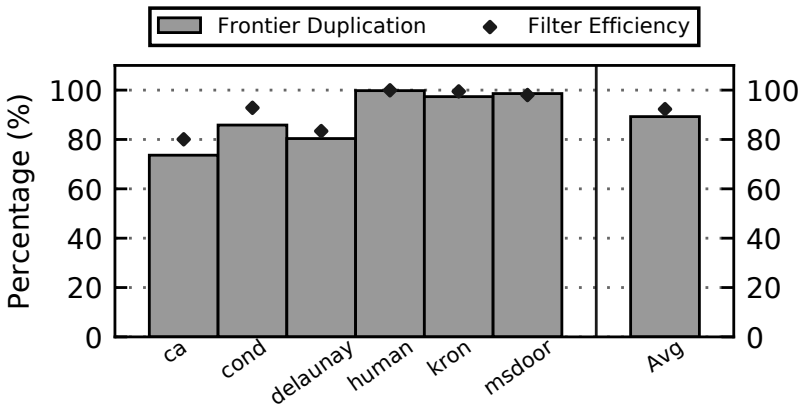


Figure 1.5: Frontier workload duplication and software filtering efficiency for BFS algorithm. The frontier duplication indicates the amount of duplicated elements within the frontiers, averaging a 89% for BFS. Meanwhile the filtering efficiency, which averages 92% for BFS, indicates the percentage of workload eliminated either because of detected duplication within the frontier, or due to elements that had already been processed previously. Filtering efficiency is higher due to this suppression of workload across frontiers (i.e. already processed elements).

divergence as some of the threads in a warp might be disabled due to their data being duplicated or their work being superfluous, thus under-utilizing the functional units of the GPU.

The overheads of software approaches are significant, either in synchronization or in missed duplicated workload. Figure 1.5 shows for the BFS algorithm the percentage of frontier duplicated elements, which averages 89%, and the effectiveness of its best-effort workload filtering approach, averaging a 92%, while on algorithms using precise synchronization it reaches 100%. We identify that on average, for several graph datasets and algorithms, as much as 94% of elements are duplicated and need to be eliminated by costly software mechanisms. We identify this duplication detection mechanism as a potential optimization factor with the objective to either entirely avoid or reduce the overheads observed while reducing the amount of duplicated workload, thus improving efficiency and utilization of graph processing on GPGPU architectures.

1.3 State-of-the-art in GPGPU Irregular and Graph workloads

Graph-based and Irregular programs on GPGPU architectures face many challenges that often result in low GPU utilization and poor performance. Several previous works have thoroughly analyzed the causes of these inefficiencies, that boil down to memory accesses divergence and irregularity [18, 112, 94, 166, 20]. Nonetheless, if these issues are overcome, irregular applications can greatly benefit of the high parallelism that GPU architectures offer. Consequently, improving GPGPU architecture limitations such as memory divergence and memory contention has attracted the attention of the research community during the last few years. In the same way, many research works have focused on improving GPGPU graph processing proposing many software, microarchitectural and newer architectures solutions.

1.3.1 Memory Divergence

The occurrence of thread divergence when performing memory accesses can significantly impact performance, as well as reduce data locality and increase memory contention. Is for this reason that several works have profoundly explored this topic seeking deeper understanding and better performing GPGPU architectures.

A straight-forward approach to handle memory divergence that has been explored by several works is to directly modify by software the programs' data structures with the intent to minimize diverging accesses [48, 106, 161], whereas other works such as HALO [47] propose instead to modify the program input data to improve data locality, in this particular case to provide a static offline reordering of a graph dataset. In a similar line, several works propose software frameworks for programmers to do memory divergence optimization to realize performance improvements [173, 157]: either offline by analyzing and performing source-level restructuring to obtain GPU kernels with varying thread granularity [157], or online by analyzing dynamic irregularities in GPU computing and performing data reordering and thread remapping improvements [173].

Specialized approaches instead focus on a selected application and exploit the characteristics of it. Works on Non-deterministic finite automaton (NFA) propose to dynamically employ the GPU shared memory to store frequently used sizable lookup tables [91]. Many specialized works have focused on GPU execution of irregular Sparse Matrix Vector Multiplication (SpMV) and Matrix Matrix Multiplication (GEMM) by proposing software approaches that reorder the matrices dataset [119], algorithms tailored for specific data characteristics of the matrices [127], and row reordering techniques [69] to improve data locality among processed rows. Finally, other works propose approaches to realize the best data layout possible to minimize divergence and set to prove the NP-completeness of a process to find, through data repositioning, that data layout concluding with software algorithms to attain it [162].

Despite the extensive literature on memory divergence, the current solutions do not appear very straight-forward and require important involvement of the programmer to build their application around this problem through hardware conscious decisions, use of frameworks and analysis tools, or specific solution for specific applications. Consequently, memory divergence remains one of the key challenging aspects for applications to achieve efficient GPGPU execution. In this work, we propose novel solutions to improve memory coalescing for data compaction workloads in graph processing (SCU) and in irregular access workloads (IRU). We develop efficient offloading of data compaction that in turn brings memory coalescing improvements on graph-based workloads and we facilitate simple and efficient memory coalescing improvements of irregular accesses.

1.3.2 Memory Contention

Memory divergence has bast consequences on the overall efficiency of GPGPU architecture, negatively impacting data locality and in turn memory contention. MLP is a key aspect for efficient GPGPU applications and thus memory contention, i.e. penalties caused by memory accesses, impedes high bandwidth exploitation and reduces overall performance.

Several works have explored memory contention improvements that propose microarchitectural improvements transparent to the programmer, whereas a few other works propose some involvement of the programmer to achieve the desired result. Extensive research has been done on flexible cache solutions [80, 85, 51] which realize the different accesses granularity of irregular memory accesses and propose dynamic cache organizations that adapt for fine-grained and coarse-grained accesses, with the objective of improving L1 cache utilization and finally reducing memory contention. Other works pursue memory contention improvements by resorting to cache bypassing mechanisms [21, 86]. More microarchitectural approaches have also been explored in works that leverage the warp scheduler to modify caches entries lifetime based on the warp scheduler prioritization [160], or entirely adapt the warp scheduling to minimize cycles data reuse, and thus potentially reduce the cache capacity required. Additionally, works such as LAMAR [126] explore sizable GPU architecture and memory hierarchy modifications to detect and provide fine and coarse grained accesses throughout the memory system.

Other solutions to memory contention propose improving data dependency with hybrid software and hardware approaches that enable data dependent aware dynamic scheduling [167] or provide prefetching of irregular accesses [81] to registers to avoid early data eviction. Finally, works such as D2MA [67] and Stash [77] set to provide mechanisms to manage global data allocation to shared memory with the objective to increase capacity close to the cores and improve memory hierarchy and overall performance.

High memory bandwidth is key for high performance GPGPU application, memory contention derived from memory divergence is a very limiting factor to achieve efficient applications. Many works have proposed a wide selection of approaches; many at the architectural level and a few with hybrid software approaches, that deal with memory contention and its consequences. Instead, in this work we approach the memory contention problem with novel proposals that deal with the cause, not the consequence of memory contention which is the high memory divergence. Our works provide effective means to improve memory coalescing which in turn bring important reductions in the overall contention of the memory hierarchy.

1.3.3 Stream Compaction

Stream compaction is an algorithm used to allocate data in compact and contiguous memory space from memory accessed in sparse, disperse and irregular way. Stream compaction is an efficient algorithm used as a primitive building operation to implement many algorithms and programs, consequently it has attracted a lot of attention over the last decade to optimize its performance on GPGPU architectures.

Many of the works on stream compaction are based on the parallel prefix-sum algorithm [14] for parallel architectures, prior to the popularization of GPGPU architectures. Initial work on parallel prefix sum algorithms [63, 142, 128, 56, 141] propose GPGPU adapted software optimized implementations which focus on depth-optimal and work efficient algorithms. Following works on this area propose further optimized software implementation which focus on maximizing concurrent execution and minimizing synchronization [13, 169], avoiding divergent execution [61], bandwidth efficient implementations [52], single-pass methods [98] and non-order preserving methods [8].

Many works have approached stream compaction from a software optimization perspective with the aim to improve aspects of the algorithm which make it inefficient in GPGPU architectures such as excessive synchronization overheads. In this work we realize that stream compaction is an big part of graph processing and consequently we propose a novel and efficient GPGPU hardware extension to perform stream compaction operations without the overheads of GPGPU architectures.

1.3.4 Graph Processing

Graph processing is an important and ascendant area, and using GPUs for it is challenging as traditionally GPUs have been designed for throughput-oriented applications featuring streaming and regular memory accesses. Unlike these applications, graph processing algorithms are highly irregular and input dependent parallelism that makes it challenging to run efficiently, particularly irregular memory accesses present a major challenge. High performance and energy efficient graph processing has attracted the attention of the architectural community in recent years. Several papers have studied their limitations and constrains on GPGPU architectures [20, 166, 164, 144], and many more have proposed solutions from different software, hardware and standalone accelerator approaches.

Software Solutions

An approach thoroughly explored is the proposal of graph processing frameworks for GPGPU architectures which aim to facilitate the use of certain programming models and deliver optimizations and efficient execution. Medusa [175] enables the use of sequential C++ to program graph processing for GPU architectures with the use of a runtime. TOTEM [48] allows graph partitioning schemes, while GasCL [19] provides an API to program graph applications in a Gather-Apply-Scatter (GAS) programming model. Goffish [148] framework integrates shared memory usage, and MapGraph [37] dynamically chooses different scheduling strategies based on frontier sizes. GraphReduce [140] adopts a combination of edge and vertex-centric implementations of the GAS programming model, while HPGA [170] provides abstractions used to map vertex programs to generalized sparse matrix operations. Finally, the popular Gunrock [161] implements as well a data-centric abstraction centered on operations on the vertex or the edge frontier.

Multiple works propose solutions to specific graph processing shortcomings such as low utilization or data locality and memory bandwidth. Garaph [95] proposes a vertex replication scheme that tries to maximize GPU utilization and provide work balanced edge-based partitions. Handling of memory overflows from GPU is also explored with a MapReduce-based out-of-core GPU memory management technique [147] for processing large-scale graph applications, overlapping CPU-GPU data transfers and computation. Finally, Subway [132] proposes to only load active edges of the graph to the GPU memory to improve the efficiency of the memory bandwidth used.

Interesting adaptive solutions which modify the graph data representation to improve fitness for the GPGPU architecture have also been explored. CuSha [74] proposes two different graph dataset representations: G-Shards and Concatenated Windows, with the aim to improve utilization and memory coalescing, and follow works which propose formats such as Warp Segmentation [73]. Similarly, Tigr [106] transforms the graph representation offline to generate a more regular dataset, making it more amenable for GPGPU architectures.

Graph algorithm specific solutions are similarly an interesting performance improvement direction. Among the graph algorithms explored we can find traversal algorithms such as Breadth-First Search (BFS) [99] implemented with fine-grained task management with efficient prefix-sum, parallel-friendly and work-efficient methods to solve Single-Source Shortest Paths (SSSP) [29] and recommendations using PageRank (PR) [45] on graph datasets. Additionally, works such as Frog [145] and others [88] explore GPGPU graph coloring algorithm, i.e. assignment of colors to elements of a graph based on constrains such as dependency updates.

Finally, big graph datasets require large quantity of memory to process the data efficiently and so, multi-GPU graph processing solutions have been proposed. Lux [68] proposes a distributed multi-GPU locality-aware system that exploits the aggregate memory bandwidth to process large graph datasets, while mGPU [117] puts forward a graph processing library that enables to extend single GPU graph algorithms.

Overall, the large quantity of works over the last decade on GPU accelerated graph processing clearly indicate the increase popularity and relevance of graph processing on GPGPU architectures. Several approaches are explored at the software level solution; graph processing frameworks, schemes to deal with specific shortcomings such as low utilization or data locality, improved data representation formats, algorithm specific implementation and finally multi-GPU proposals. In this work we realize that GPU architectures, although highly effective, have inherent difficulties for efficient graph processing. Consequently, we propose the SCU a novel hardware GPGPU extension which efficiently performs the inefficient aspects of GPGPU graph processing. On top of it, we also propose the IRU, a novel hardware solution able to improve the memory coalescing shortcoming of irregular applications in general and graph processing in particular.

Microarchitectural Solutions

A few works on graph processing propose GPGPU microarchitectural solutions. Some previous works address the low utilization problem for graph processing by focusing on microarchitectural details of GPGPU architectures such as branch divergence [54] proposing to delay diverging threads in branches until later iterations of a loop. Other works [62] point to warp size and warp-centric programming to improve utilization. The work in [146] presents mechanisms to reduce address translation overheads for irregular applications such as graph processing.

Register prefetching [81] is exploited to improve memory access latency tolerance for graph applications, done with additional hardware that detects load pairs common in graph algorithms and injects instructions to prefetch the corresponding data. Other works propose a work redistribution hardware scheme to improve load balancing [75] for graph algorithms. Finally, GraphOps [113] proposed a modular hardware library for quickly and easily constructing energy-efficient accelerators for graph analytics algorithms, specially targeting FPGA architectures.

Overall, most previous works are not specifically targeting graph processing but more broad irregular applications or targeting specific problems such as branch divergence which are relevant as well for graph processing. Some interesting architectural solutions are proposed such as prefetching and workload redistribution. In our work, we propose novel hardware solutions both specifically for graph processing with the SCU and more broad irregular applications with the IRU.

Accelerators Solutions

There is a plethora of works that propose to replace entirely the GPU with special purpose accelerators custom-made for graph processing, which set aside the GPU due to fundamental limitations of GPU irregular program execution. The main approach of these works focus on exploiting deep knowledge of graphs data structures. This insight into the graph data organization is exploited by accelerators closely integrated with memory, using new memory technologies or near data processing approaches.

Proposals include standalone accelerator approaches such as SGMF [159], TuNao [177], AccuGraph [171] and others [115]. A different approach is the exploration of accelerators for specific memory technologies such as dram-based Graphicionado [53], ReRAM memory based GraphR [149], SOT-MRAM based accelerator GraphS [4], SSD-based GraphSSD [96], and flash-based GraFBoost [70]. Meanwhile, the popularization of near-data-processing and new 3D-stack memory technologies has ushered in many processing-in-memory (PIM) graph specific solutions: GraphH [28], GraphQ [178], Tesseract [2], ExtraV [83], G2 [176] and in-memory and cloud proposal.

Due to the GPU original design that aims at throughput-oriented applications with regular memory patterns, achieving efficient graph processing is challenging. This challenges have prompted a wide variety of custom accelerators purpose-made for graph processing. In our work, we realize that although GPU architectures are not very efficient for specific parts of graph applications, they are indeed well suited for graph processing enabling high parallelism. Consequently, in our work we set to improve the suitability of GPGPU architectures for graph processing instead of entirely replacing them. In this way we propose to offload inefficient stream compaction operations to tailor made hardware additions to the GPU, as well as improve the GPU capabilities of processing irregular accesses present on graph applications.

1.4 Thesis Overview and Contributions

The goal of this thesis is to propose novel and effective techniques to improve performance and energy efficiency of graph processing and irregular applications in GPGPU architectures. Graph processing on GPGPU systems faces control flow and memory divergence obstacles which impede full utilization of the parallelism and system resources available. Our main contributions are the Stream Compaction Unit (SCU), Irregular accesses Reorder Unit (IRU) and the improved graph processing GPGPU architecture that results of the integration of both previous hardware proposals. We evaluate all our proposals for graph processing algorithms on a high performance

GPGPU system, as well as on a low-power GPGPU system. The following sections outline the description of the problems approached, the proposed solutions which comprise this thesis contributions and finally their comparison with related work.

1.4.1 Energy-Efficient Graph Processing by Boosting Stream Compaction

Graph processing algorithms are key in many emerging applications in areas such as machine learning and data analytics. Although the processing of large scale graphs exhibits a high degree of parallelism, the memory access pattern tend to be highly irregular, leading to poor GPGPU efficiency due to memory divergence. To ameliorate this issue, GPGPU applications perform stream compaction operations on each iteration of the algorithm to extract the subset of active nodes/edges, so subsequent steps work on compacted dataset. Our numbers indicate that stream compaction operations represent between 25% to 55% of the total execution time, but GPGPU architectures are not efficient for stream compaction workloads.

Contribution

In first place, we analyze and characterize the performance and energy consumption of stream compaction operations of graph processing applications on GPGPU architectures. We conclude that it can surpass 50% of the execution time and it performs poorly due several reasons. First, it consist of sparse memory accesses with poor locality that fetch elements (nodes/edges) to be compacted, consequently it results in very low memory coalescing that produces intra-warp memory divergence and reduces GPU efficiency by a large extent. Second, stream compaction has an extremely low computation to memory access ratio, as it primarily consists of load and store instructions to move data around in main memory. Since GPGPU SMs are optimized for compute-intensive workloads, the functional units remain largely underutilized during the stream compaction stage.

In second place, we propose the SCU, a novel hardware unit tailored to the requirements of stream compaction operations which integrated with existing GPGPU architectures is able to provide 2x energy reduction and 1.5x speedup for graph applications on both high-performance and low-power GPUs. The SCU features a set of fundamental stream compaction operations which enable to offload compaction operations from the GPU to the SCU. This allows to modify existing graph processing algorithm to offload the stream compaction efforts to the SCU while leveraging the high parallelism of the GPU to perform the graph exploration computation.

In third place, we extend the proposed SCU with additional pre-processing to the compacted data so that the GPU can process it in a more efficient way. We propose two extensions with this goal: filtering and grouping. Filtering detects duplicated elements which can be discarded. This elements can be removed due to the nature of the graph exploration with parallel algorithms, which sees multiples edges explored simultaneously with the possibility of repeated elements. Filtering is a very effective technique which manages to remove 75% of the original GPU workload. Additionally, grouping reorders data to maximize the effectiveness of memory coalescing.

Finally, we evaluate the final SCU system with the proposed filtering and grouping pre-processing functionality which improve its energy efficiency and performance significantly. Overall, the high-performance and low-power GPU designs including our SCU unit achieve speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively on average for several graph-based applications. This work has been published in the proceedings of the International Symposium on Computer Architecture (ISCA '19) [139].

Related work

The related research proposals that tackle memory divergence do so in a direct way by means of providing load balancing mechanism [99], changing the microarchitecture [39] and scheduler [17] to modify individual thread execution paths or directly modifying the program data structures [48, 106, 161]. In contrast, the SCU main aim is to offload stream compaction operations, yet it also manages to improve GPGPU control and memory divergence in a less intrusive way. It indirectly improves divergence with the pre-processing filtering and grouping. GPGPU utilization is improved by eliminating duplicated elements that would cause warps to underutilize the execution units and more memory accesses.

The majority of the works on graph processing focus on a software perspective with the aim to provide new frameworks [175, 19, 161] to facilitate programming efficient graph applications or provide new data representation formats [95, 147, 132]. Among the proposals that explore hardware solutions we can find register prefetching [81] and load balancing [75], yet the majority of the works directly propose an standalone accelerator solution [177, 53, 2, 4]. Additionally, many of the works on GPGPU stream compaction require solutions to problems arising from the parallelization of the scan algorithm such as increased space constrains or synchronization overheads [13, 169].

Our SCU solution is different from previous proposals since we extend the GPU with a small programmable unit that boosts GPU performance and energy-efficiency for graph processing algorithms. We observe that, although the GPU is inefficient for some parts of graph processing (e.g. stream compaction), the phases that work on the compacted dataset can be efficiently executed on the GPU cores. Therefore, we only offload stream compaction operations to the SCU, meanwhile we leverage the streaming multiprocessors for parallel processing of the compacted data arrays. Finally, unlike previous hardware-based solutions, our SCU has a very low cost in area for high-performance and low-power GPU respectively, and it does not require any changes in the architecture of the streaming multiprocessors.

1.4.2 Improving Graph Processing Divergence-Induced Memory Contention

GPGPU architectures are the preferred system to achieve performance by means of parallelization, which allows performance improvements of orders of magnitude. However, irregular applications struggle to fully realize GPGPU performance as a result of control flow divergence and memory divergence due to irregular memory access patterns. To ameliorate these issues,

programmers are forced to carefully consider architecture features and devote significant efforts to modify the algorithms with complex optimization techniques, which shifts programmers priorities yet struggle to quell the shortcomings. We show that in graph-based GPGPU irregular applications these inefficiencies remain, yet we find that it is possible to relax the strict relationship between thread and data processed to empower new optimizations.

Contribution

We first characterize the degree of memory coalescing and GPU utilization of irregular applications, specifically modern graph-based applications. Our analysis shows that memory coalescing can be as high as 4 accesses per warp and GPU utilization as low as 13.5%. We conclude that GPGPU programming models impose restrictions that hinder full resource utilization of irregular applications for several reasons. First, irregular programs such as graph processing algorithms consist of sparse and irregular memory accesses which have poor data locality and result in bad memory coalescing, producing intra-warp memory divergence and reducing GPU efficiency significantly. Second, these issues are, in the best of cases, hard to improve without significant programmer effort to modify algorithms and data structures in order to better utilize the underlying hardware. Ultimately the programmer has to take into consideration ways to rearrange the data or change the mapping of data elements to threads to achieve better memory coalescing and higher GPU utilization, even if the relation of which threads process what data might not even be a restriction imposed by the algorithm, since the threads are primarily the means to expose parallelism.

Then, we propose the IRU, a novel hardware unit integrated in the GPGPU architecture which enables improved performance of sparse and irregular accesses achieving 1.33x speedup and 1.13x energy reduction for graph-based processing applications. IRU's key idea is to relax the strict relation between a thread and the data that it processes. This allows the IRU to reorder the data serviced to the threads, i.e. to decide at run-time the mapping between threads and data elements to greatly improve memory coalescing. The IRU mapping improves the effectiveness of the memory coalescing hardware resulting in better coalescing and cache data locality, with subsequent improvements in the entire memory hierarchy and higher GPU utilization for irregular applications. In addition, the IRU performs simple pre-processing on the data (i.e filtering repeated data), which reduces useless resource utilization of the GPU and allows for better utilization and further performance and energy improvements.

We also propose an ISA extension and high-level API for the IRU, and we show how modern graph-based applications can easily leverage the IRU hardware. Efficient irregular GPGPU programs require complex thread to data assignment or additional pre-processing (e.g. reordering) to be performed by the programmer. Unfortunately, at the cost of additional algorithm complexity and computational cost. Most importantly, this is a complex burden for the programmer which usually are not willing or able to handle as it clearly shifts the programmers effort from the algorithm to a hardware conscious programming, requiring sound knowledge of it and hampering code portability. Instead, programmers can easily utilize the IRU to achieve efficient execution of irregular programs with our proposed simple API, or with compiler optimized generated code with the extended ISA instructions provided.

In conclusion, the IRU delivers efficient execution of irregular programs on modern GPGPU architectures. It achieves a memory coalescing improvement of 1.32x and a 46% reduction in the overall traffic in the memory hierarchy, which results in 1.33x speedup and 1.13x energy reduction for graph-based processing applications. The IRU optimizes irregular accesses while requiring minimal support from programmers. This work has been submitted for publication [137].

Related work

The majority of the related works that tackle memory divergence propose software methods that require extensive programming effort to change algorithm behavior [73, 99], data structures [48, 106, 161], and require profound hardware knowledge and compiler assistance to properly achieve reordered and optimized efficient solutions [157]. Other works propose solutions that rely on the specific characteristics of the irregular applications targeted such as the Sparse Matrix Multiplication [119, 127].

Related to our solution some works propose to perform thread remapping approaches. Software approaches propose to perform reordering with label-assign-move (LAM) with CPU-assisted analysis and remapping frameworks [174, 173]. Thread remapping microarchitectural approaches propose changes to the SIMT stack [39], reconstructing warps at divergent points [38] and support extension of warp schedulers [17].

In contrast, our IRU solution requires very lightweight changes of the algorithms and does not require profound knowledge of the inner working of the GPU memory hierarchy nor is targeting specific characteristics of algorithms to improve memory coalescing and resolve contention issues. Of the thread remapping approaches explored, the majority specifically target branch divergence and utilization improvements while overlooking the effects of remapping on memory divergence. The software remapping framework [173] does target memory divergence but it requires complex algorithm changes. Our proposal is a hardware method which performs the reordering optimization specifically targeting the improvement of memory divergence and it does so in a close to transparent way without the requirements of CPU assisted analysis or frameworks nor complex extensions to the GPGPU microarchitecture.

Related works approach memory contention by proposing flexible cache solutions [80, 85, 51], cache bypassing mechanisms [21, 86] and data reuse latency by means of warp prioritization with the warp scheduler [160]. The aforementioned related works leverage hardware solutions that work around or ameliorate the consequences of low memory coalescing by providing mechanisms to lower memory contention. In contrast, our IRU work provides tools to amend the cause, not the consequence, of the high memory contention which is poor memory coalescing that leads to high memory contention, and our IRU significantly improves the memory contention of the GPGPU with irregular workloads.

Finally, in contrast to graph accelerators which require systems to add special custom hardware, the IRU leverages the popularity of GPU architectures and provides enhancements and a generic solutions that bring performance and efficiency improvements of the GPU architecture for irregular programs.

1.4.3 Combining Strengths of SCU and IRU

Current and future data gathering requirements in our knowledge-based society demand great data processing efficiency, essential for emerging domains such as data analytics or machine learning. High-throughput GPGPU architectures are key to enable efficient graph processing. Nonetheless, GPGPU are afflicted by poor efficiency caused by high memory divergence and contention, which arise from graph-based applications irregular and sparse memory access patterns. In our previous work, we point to stream compaction and irregular access divergence to improve GPGPU graph processing efficiency and performance. We find that it is possible to leverage the strengths of both approaches to achieve synergistic performance improvements and higher graph processing efficiency.

Contribution

We first characterize the limitations and bottlenecks of our initial work on the SCU. We observe that its major limiting factor is the large amount of data movement between the L2 cache. This high movement is caused by the SCU pre-processing operations and its accesses to an in-memory hash, which contribute to 57% of Network-on-Chip (NoC) traffic. The in-memory hash enables the pre-processing of filtering and grouping operations, delivering massive filtering percentages of duplicated elements. After the pre-processing, the majority of accesses performed are limited to the in-memory hash, and so, they represent an important performance and energy bottleneck.

Afterwards, we identify synergies between the SCU and the IRU and show that they perfectly complement each other. While the SCU achieves significant speedups and large energy savings, it produces high contention in the NoC. On the other hand, the IRU achieves more modest energy savings, but its optimizations are highly efficient at largely reducing contention in the NoC to a factor of 46% lower NoC traffic.

Based on this observation, we propose the ISCU, a novel GPU unit that combines both system strengths. The ISCU leverages the powerful stream compaction offloading achieved by the SCU and the efficient filtering mechanism of the IRU employed to deliver pre-processing optimizations. The proposed system improves overall graph processing efficiency.

Finally, we evaluate our proposal to improve graph processing on top of a modern GPU architecture and for a diverse set of graph-based applications. Our experimental results show that the ISCU delivers a high reduction of 78% memory accesses, which result in a reduction of 90% in energy consumption and 2.2x speedup on average. Compared to the SCU, our ISCU improves performance by 63%, while achieving 66% energy savings.

In conclusion, the ISCU leverages the strengths of our previous work on improved graph processing by offloading stream compaction operations, and our work on improved irregular accesses on GPGPU architectures which deliver synergistic improvements in efficient graph processing. This work has been submitted for publication [138].

Related work

The majority of the hardware proposals for graph processing propose custom build accelerators [177, 171, 53] with many of them closely exploring memory technology synergies to improve graph processing [4, 28, 2]. In contrast, we propose extensions to the already popular and ubiquitous GPGPU architectures which make them more versatile to support graph-based workloads for which they face many challenges. Our complete solution leverages the strengths of our earlier works on stream compaction offloading and irregular accesses reordering which results in a high energy efficiency, increased performance and flexibility for graph processing on GPGPU architectures.

1.5 Thesis Organization

The reminder of this document is organized as follows:

Chapter 2 describes the relevant background on GPGPU architectures, as well as concepts of Irregular Applications and Graph Processing Algorithms on this architecture. We describe the characteristics of Irregular Applications and a detailed explanation of the Graph Algorithms explored on this thesis.

Chapter 3 goes through the experimental methodology. First, we describe the different tools and the parameters used in our evaluation of performance, energy and area. Second, we introduce the hardware systems and simulators we employ, including our own simulator and the configuration parameters utilized. Finally, we describe the graph processing datasets used in our evaluation.

Chapter 4 proposes a programmable GPGPU hardware extension Stream Compaction Unit (SCU). We first characterize the performance and energy of stream compaction operations on modern GPU architectures and then we propose the SCU as an efficient hardware unit tailored for the requirements stream compaction. We show how it can be integrated with existing graph applications and demonstrate the efficiency of the SCU to filter duplicates and improve overall memory coalescing.

Chapter 5 presents the Irregular accesses Reorder Unit (IRU), which improves irregular accesses on GPGPU architectures. We first characterize the degree of memory coalescing and GPU utilization of modern irregular graph-based applications. We then propose the IRU, a novel hardware unit integrated in the GPGPU architecture that enables improved performance of sparse and irregular accesses by reordering data serviced to each thread and allowing duplicated data filtering. Finally, we describe its API which enables seamless integration with existing graph applications.

Chapter 6 describes the synergistic integration of our previous hardware extension proposals into a final improved graph processing GPGPU system, the IRU-enhances SCU (ISCU). We first characterize the limitations of our initial SCU design, identifying the main bottlenecks. We then propose the adaptation of the IRU hardware extension with the SCU which improves execution of stream compaction operations addressing the bottlenecks identified. Finally, we conclude

our work by achieving synergistic improvements in energy-efficiency and performance for graph processing on GPGPU architectures.

Chapter 7 summarizes the main conclusion of this thesis, lists the contributions achieved and outlines some open research work areas.

2

Background

This chapter reviews the relevant background that serves as the foundation to contextualize and understand the work presented throughout this thesis. This chapter first explores the architecture of a GPGPU and showcases its programmability. Afterwards, potential sources of inefficiency in GPGPU architectures are discussed, with special attention on execution of irregular applications. Finally, the chapter explores algorithms for graph processing on GPGPU architectures.

2.1 GPGPU Architecture

The specific characteristics of GPGPU architectures are worth exploring to better understand their strengths, competences and limitations. This section describes the GPGPU philosophy and its main components, from execution units to the memory hierarchy, and concludes by detailing GPGPU programmability to properly utilize hardware resources.

2.1.1 Overview

GPGPU architectures are tailored for massively parallel, compute intensive applications. To support high thread counts, a GPGPU devotes higher die area to replicate Execution Units (EU) while fewer area is dedicated to complex control and caches, compared to traditional CPU designs and seen in Figure 2.1. This fundamental decision defines GPGPU approach and trade-offs to deliver performance; hugely increased counts of significantly less efficient threads. The more characteristic trade-offs required to support as many threads without in turn significantly increasing control area are: an in-order pipeline with a back-end executing threads in lock-step, and the reduced decoding complexity per thread as a consequence of the SIMT approach that

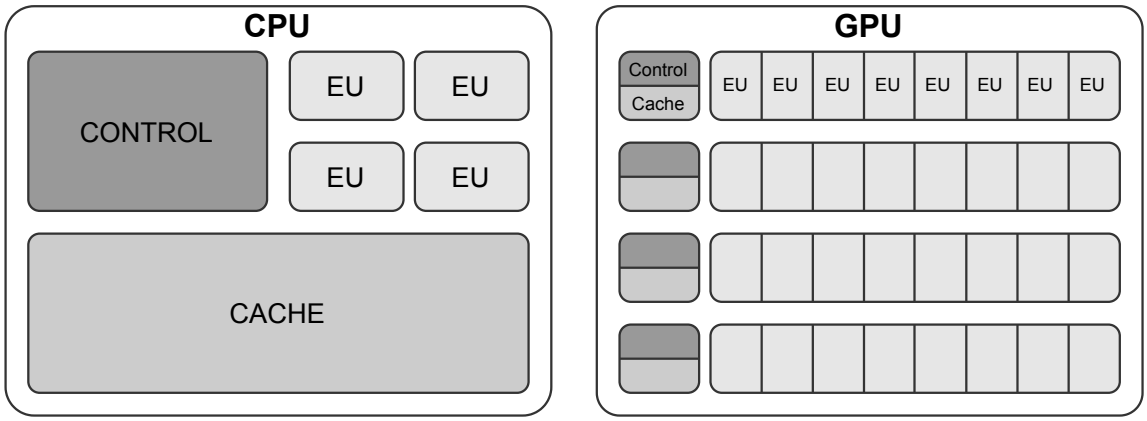


Figure 2.1: Diagram of the transistor area distribution of a CPU versus a GPU architecture, showcasing the design principles of each architecture. CPU designs dedicate comparatively more area to improved control logic and caching, whereas GPUs rely on simpler control logic and caches while featuring much more execution units [76].

decodes a single instruction for multiple threads. The resulting increased thread latency is hidden with the increased throughput and parallelism, consequently GPGPU excels at running applications with high Instruction-Level Parallelism (ILP) and Memory-Level Parallelism (MLP). Many scientific, calculus and visualization applications fit these characteristics which allow to maximize the utilization of the EUs and hide the increased latency.

The overview of the components of a GPGPU architecture is shown in Figure 2.2. A single GPU contains multiple cores or Streaming Multiprocessors (SM) with the characteristics previously mentioned. The GPU die contains a number of SM and Memory Partitions (MP), which are connected with an interconnection Network-on-Chip (NoC), such as a crossbar or a butterfly network, and the Memory Partitions (MP) contain a partition of the shared last-level cache and provide accesses to multiple off-chip main memory channels.

2.1.2 Streaming Multiprocessor (SM)

The general architecture of a GPGPU core is similar to the architecture of a regular CPU multiprocessor with some additions and specializations. GPGPU architectures feature SIMT in-order pipelines where the front-end is capable of scheduling and issuing multiple SIMT instructions per cycle (e.g. typically 2 or 4), while the back-end is specialized and pipelined in different functions; arithmetic or special function (ALU), memory (LD/ST), and the newly added tensor operations (Tensor units). The pipelines are warp-wide (typically 32 thread-wide) with predicated execution managed with masking registers, enabling or disabling the execution of threads in a warp.

Since the pipeline is SIMT, a single instruction operates on multiple data effectively doing the work of multiple instructions on a regular pipeline, and being equivalent to the execution of a thread per data. This approach benefits from requiring simpler instruction decoding logic per regular instruction executed and consequently achieves reduced energy decoding cost per

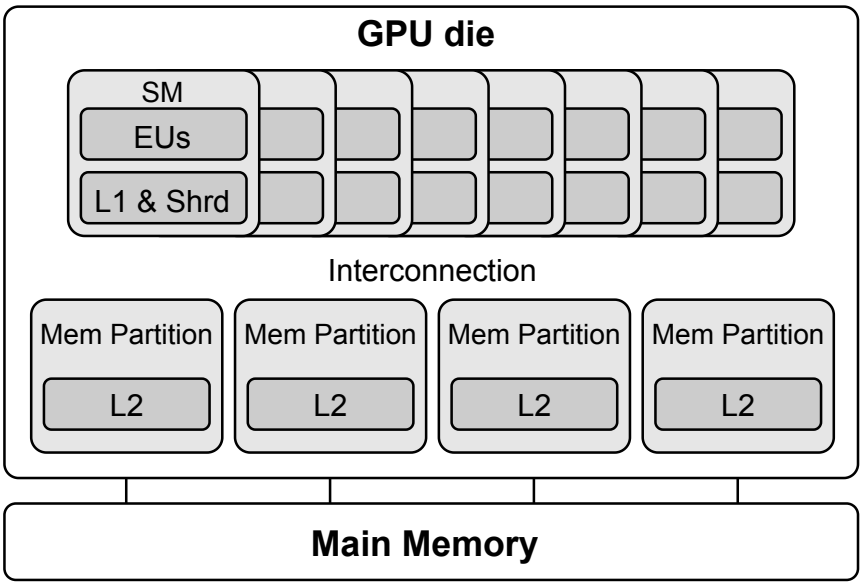


Figure 2.2: Overview of a GPGPU architecture showing the GPU die with 16 Streaming Multiprocessors (SM) interconnected with 4 Memory Partitions (MP) and the main memory located outside of the die. The most relevant internal components are showcased, for the SM are the Execution Units (EUs), L1 data cache and shared memory, whereas for the MP is the L2 data cache.

instruction executed. The decoding logic resulting from an instruction governs the threads in a warp, which are executed in lock-step being effectively scheduled, issued, executed and retired from the pipeline jointly.

A SIMT instruction operates on the same register for each thread but with different data per thread and as such, a register entry in a GPGPU architecture is several times larger than a regular register. Consequently, in order to fully utilize the multiple pipelines, the register file (RF) has to be able to provide much more data for the operands of the multiple threads in a scheduled warp. The increased RF's data throughput can be delivered with a multi-ported RF or specialized hardware such as an operand collector [92] which uses an arbiter and multiple single-ported RF to provide the desired throughput with fewer area and energy overheads.

Additional hardware resources are required to manage branch divergence, typically resolved with the use of a SIMT stack. On a GPGPU architecture, branch divergence occurs when a branching instruction generates different control flow paths for the threads in a warp. This diverging control flow is specified with the use of predication register serving as an active mask. The SIMT stack is used for each warp to keep track of the control flow information such as the next program counter (PC), the active threads mask and reconvergence PC. At a divergent instruction, this information is pushed into the stack with the new active mask and reconvergence PC, whereas upon reconvergence the information is poped effectively managing branch divergence. The threads issued to the back-end and their assigned functional units are deactivated with the corresponding active mask in the stack for that warp. Notice that, unlike traditional architectures, potentially both taken and not-taken paths instructions are executed based on the warp active mask.

2.1.3 Caches and Memory Hierarchy

The memory hierarchy of a GPGPU system is critical to achieve high performance. Significant memory bandwidth is required in order to maintain the huge parallelism of GPGPUs. The thousands of threads occupying the functional units require to read and write data to maintain high utilization. As explored earlier, the RF is provisioned to sustain the high performance, but the whole memory hierarchy needs to be sized accordingly as to provide the cores with enough data as well. The SM houses several memory components worth exploring featuring several caches, memories and specialized hardware. Similar to a traditional architecture, the SM features an instruction cache and a data cache with the addition of a memory accesses coalescer to reduce memory contention. Additionally, two other caches are present: the constant cache, employed for parameters and program constant values, and the texture cache specialized for holding data of texture mappings. Finally, an scratchpad memory local to the SM is used to facilitate in-core memory communication between threads.

Streaming Multiprocessor Caches and Memory Resources

Data cache size for the SM are characteristically small compared to traditional computer architectures, and the ratio of cache entries per thread is even smaller. This design choice guides which kind of application can effectively use the architecture, e.g. streaming (read, process, and output) and high computation applications, and limit the data reuse possibilities for applications processing many data and irregular patterns. A warp processing a memory instruction requires to read or write significant amounts of data from or to the caches, consequently several techniques and hardware are used to improve performance and reduce contention. First, cache entries have bigger granularity which allows to service a complete warp for word-sized memory instructions (i.e. a total of 128 bytes assuming a warp size of 32 threads). Second, cache policies are typically set to evict data on a write with no write allocation, since they most benefit streaming applications. Third, processing of misses with additional hardware resources such as the Miss Status Holding Registers (MSHRs) used to store information of processing misses and merge new misses to the same entries, and Miss Queues to avoid blocking new accesses when outputting misses. Finally, coalescing hardware is used to further reduce contention by merging multiple memory requests within a warp.

The accesses coalescing hardware is a significantly important addition to the GPGPU memory hierarchy. Its purpose is to reduce memory accesses to the L1 data cache, improve utilization and consequently effectively reduce accesses to the entire memory system. Every memory request that originates from a warp memory instructions is processed by the coalescing hardware before being issued to the L1 data cache. The accesses coalescer merges accesses that correspond to the same memory block and keeps track of the data that each individual access (i.e. thread) requested in it. Given multiple accesses performed by the warp that target the same memory block, the coalescing is capable to merge them to a single access to the L1 data cache, whereas if they access different memory blocks the coalescing hardware is of no benefit as each one has to be performed individually. Consequently, the accesses coalescing hardware provides a large reduction in the accesses to the L1 cache, up to 32x reduction if the 32 threads within a warp target the same cache line.

The last relevant component inside the SM architecture is the scratchpad memory. The scratchpad or shared memory is a randomly addressable memory which has very low latency compared to main memory. Its main purpose is to facilitate communication of data and synchronization execution between threads located to that particular SM. The scratchpad accesses are not coalesced, instead the scratchpad is banked which allows divergent accesses to be performed simultaneously on to the separate banks. The data allocated to the scratchpad memory is managed directly by the application, as it is a separate address space from the main memory space.

Memory Partitions and Out-of-Chip Memory

Contrary to CPU architectures, the SM itself does not include a private L2 cache as GPGPU architectures are not as concerned to reduce latency to memory and increase data reuse. Instead the L2 data cache is the last level cache (LLC) and is shared, banked, partitioned and located close to the memory controllers, in the memory partitions of the GPGPU architecture. Accesses from the SM are directed to the corresponding L2 partition by means of the interconnection NoC which can feature designs such as a Crossbar or a Butterfly network depending on the latency, area and energy constrains of the system. The L2 features the same hardware contention resources such as MSHRs and miss queues, but no accesses coalescing, as the warp accesses are already coalesced before accessing the L1 data cache. Contrary to the L1, the L2 uses write-allocate as the write allocation policy.

The GPGPU architecture does not feature any memory coherence protocol for the L1 data caches. To correctly access data, synchronization mechanisms such as atomic operations are supported by the architecture. The memory partitions include the corresponding functional units to perform the atomic operations and guarantee atomicity.

Finally, the main memory is located off-chip and it typically features technologies such as Graphics DDR (GDDR 5/6) [40, 41] and High Bandwidth Memory (HBM 1/2) [58], which stand out for their characteristics and trade-offs: very high bandwidth, high parallelism MLP, and less optimized higher latency and energy overheads.

2.1.4 Programmability

Efficient hardware is of no use if it cannot be easily programmed and used by programmers and applications, consequently powerful and easy to use programming languages and toolkits are of high importance. Over the years, product vendors have had in their own interest to provide accessible and easy to use tools to promote their platforms, offering tools such as debuggers, profilers and highly optimized libraries for relevant application domains like machine learning or scientific computing. GPGPU main programming languages and toolkits are CUDA and OpenCL, which are extensions of C/C++ with added functionalities to interface and program GPGPU architectures.

These toolkits provide a way to program GPGPU architectures as well as mechanisms to compile and integrate it with regular CPU applications. Figure 2.3 gives an overview of the nvcc

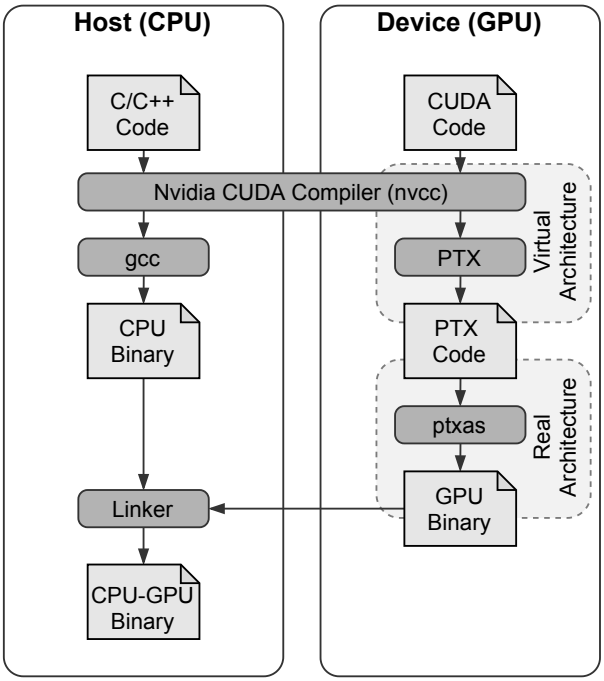


Figure 2.3: Simplified compilation diagram showcasing the process to generate and integrate a GPU kernel with a CPU application with a CUDA toolkit. The CUDA code is compiled with the nvcc into generic Parallel Thread Execution (PTX) [118] assembler code, that is later processed into GPU binary Streaming Assembler (SASS) assembler code. The same process compiles the regular CPU code and integrates both binaries in a final FAT binary that can contain multiple SASS versions to execute on different GPU architectures.

compilation of CUDA code. Similar to a regular computer system, a GPGPU executes assembly code, consequently the high level code has to be translated to assembly. High level CUDA code is translated to Parallel Thread Execution (PTX) [118] assembly, which is a generic system independent assembly language. PTX code is further translated to architecture dependent assembly code Streaming Assembler (SASS). This process is automated by the toolkits which compile and link the code for several GPU architectures creating a final binary where GPGPU machine code is integrated into a regular host or CPU binaries programs. With the use of drivers and the toolkit libraries provided, upon execution of the binary, the program running on the host platform, i.e. CPU, is able to configure, prepare the input data and ultimately launch the execution of the GPGPU program, i.e. execution of a kernel function on the GPU.

The configuration of a kernel is done before launch from the host code. Figure 2.4 gives an overview of the interaction between the Host (CPU) and the Device (GPU). At configuration, the host specifies the amount of threads to use and the size of thread block (i.e size of a group of threads allocated to an SM). Note that the thread block is independent from the architectural concept of a warp; the concept of a warp (i.e. group of lock-step executing threads) is not a concept of the programming language but a specific detail of the architecture. The launch of a kernel is done with a function call to a specially marked function that contains the GPGPU code, additionally, parameters can be provided to this function with constant values and pointers to input data to process. In the case of discrete GPUs, host and device memory spaces are

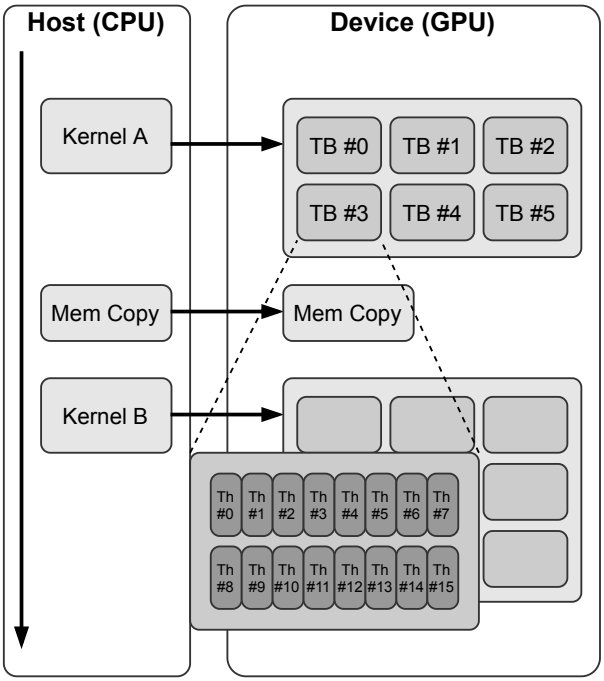


Figure 2.4: Interaction between a Host (CPU) and the Device (GPU). The CPU is responsible to perform data movements to the GPU memory, configure the parameters and initiate the launch of the GPU Kernel.

separated and data movements have to be handled manually with specific memory operations between CPU and GPU as shown in Figure 2.4. In case of an integrated GPU, host and device can share the same address space and, hence, memory copies are avoided reducing the overhead of offloading code to the GPU.

GPGPU programs are specified with kernel functions which are regular functions specially marked for GPGPU execution. These functions are programmed with either CUDA or OpenCL which provide keywords and additional values such as the thread id and size of the thread block of the instantiated kernel, which are used to distribute the work among threads. The parallelism is implicit as the code that is programmed in regular C/C++ runs on all the threads of the instantiated kernel, but with different variables per each thread. The toolkits provide synchronization barriers and atomic functions to manage and implement parallel algorithms efficiently. Finally, the different keywords provided allow the programmer to use and manage the different address spaces of the shared memory and the caches such as the constant and texture caches, while regular data is implicitly allocated on the GPU main memory.

Finally, efficient GPGPU programming requires deep understanding of the underlying architecture. For this reason, to facilitate GPGPU programmers to develop fully fledged applications, many vendors and researchers develop additional toolkits and libraries. These toolkits provide additional functionality, primitives and qualified algorithms that are highly optimized and tuned to execute efficiently on several GPGPUs without requiring extra optimization effort by the programmers, which aids in the integration, improvement and development of GPGPU accelerated applications. Examples of such toolkits and applications cover many different areas:

cuBLAS [24] for Linear Algebra, cuFFT [26] for Fast Fourier Transform, cuSPARSE [27] for processing sparse matrices, nvGRAPH [107] or Gunrock [161] to perform graph analytics and PyTorch [121] or TensorFlow [152] for machine learning and deep learning applications.

2.2 High Performance GPGPU Code and Common Bottlenecks

GPGPU systems are complex architectures which require thoughtful programming and deep understanding of the underlying architecture to achieve high performance. This section gives an overview of the aspects which positively and negatively impact performance.

2.2.1 High Performance GPGPU Code

The first aspect to achieve high performance GPGPU applications is to maintain high utilization of the functional units, which results in high IPC. The threads in a warp execute in a lock-step manner and with predication, and so to completely utilize the functional units it is necessary to minimize non-diverging path. Divergent control flow disables the functional units of the divergent threads, which underutilizes the functional units of the GPGPU. Regular applications which have regular control flow contribute to maximize gains resulting from architectural decisions such as savings in decoding and control logic that result from decoding a single instruction per multiple treads.

In order to sustain the high utilization of the functional units, enough data has to be provided by the memory system, thus requiring significant memory bandwidth. Applications that maximize memory bandwidth show more regular memory access patterns which enable high MLP and facilitate the memory system to provide high bandwidth. Additionally, regular access at the warp level significantly reduce the LD/ST pipeline latency and minimize accesses and contention to the data cache. Additionally, applications have available the shared memory which alleviates the pressure on the memory system for high reuse data which is accessed with low latency.

Overall, applications which achieved the highest performance on GPGPU architectures show regular control flow and memory access patterns and maximize coalescing. Contention of the memory hierarchy is highly important to maintain high utilization and consequently achieve high performance, such as streaming applications which have low reuse of data or applications that exhibit high computation to memory access ratio.

2.2.2 GPGPU Bottlenecks

Many choices constrain the efficient use of GPGPU architectures. Applications that show irregular behavior with unpredictable and irregular memory access patterns are likely to underachieve in a particular aspect which can in turn severely effect the performance. For these applications GPGPU architectures are unable to provide enough memory bandwidth due to a huge portion of the threads generating uncoalesced memory accesses, while performance

is additionally hampered when all functional units are not utilized due to divergent thread execution. Nonetheless, irregular applications can still benefit from the high parallel processing of data.

A warp processing a memory instruction that has irregular accesses is likely to achieve poor memory coalescing, since the accesses performed by the threads in that warp will likely not be collocated into the same memory block. In this case, an individual memory request will have to be issued for every uncoalesced thread, one per thread in the worse case. Hence, the overhead for irregular applications represents up to 32x more memory requests (assuming a typical warp size of 32 threads), which increases both utilization of the LD/ST unit and instruction latency and puts higher pressure on the L1 and the whole memory hierarchy. In addition, every warp instruction requires more resources to handle misses on L1, such as miss status holding registers (MSHRs) and entries in the miss queue, a problem aggravated by the fact that GPU L1 capacity is typically smaller than that of CPUs and the ratio of cache lines per thread is significantly lower as well. All these factors combined increase significantly the contention on the L1 and its miss ratio due to conflict and capacity misses. Finally, the interconnection traffic congestion increases, L2 observes similar problems to the ones described for L1, and main memory accesses increase as a consequence of increased L2 misses.

On the other hand, the SIMT architecture of GPGPUs enables savings in hardware costs and energy but also constrains the flexibility of the program. Irregular applications that have divergent control flow within the warps end up under utilizing the functional units in the pipeline. Upon executing a branch divergent operation, the architecture has to handle the control logic to create reconvergence points and manage the disabling of divergent executing threads. The potential under-utilization of the functional units can be as high as a factor of 32x, when only one functional unit in the pipeline is utilized, severely lowering the performance of the architecture. Additionally, the decoding trade-off is negatively impacted as well since the energy used by the decoding of a warp instruction is serving less threads and achieving less performance. Finally, branch divergence requires both divergent execution paths to be evaluated, resulting in slower overall performance.

Overall, irregular applications can and do benefit from the high performance delivered by the huge parallelism of GPU architectures, but the architecture has many pitfalls when it comes to enabling high performance and utilization of irregular algorithms.

2.2.3 Ameliorating Performance Inefficiencies

Irregular applications attempt to mitigate the aforementioned performance bottlenecks by using different techniques. Note that significant changes have to be applied to the algorithm and its data structures to utilize more efficiently the GPU resources. Possible approaches to optimize performance include the following. First, the use of the shared memory present in the SM of the GPU which provides reduced latency and allows banked accesses of uncoalesced accesses. Second, merging kernels to reuse memory requests while increasing use of registers and contention on register files. Third, modifying program data structures in a more compact way in order to improve memory access patterns, i.e. graphs use of Compressed Sparse Row (CSR) [11] format. Fourth, data compaction mechanism such as scan algorithms which reduce

sparse accesses and improve locality by gathering sparse data in a compacted data array. Finally, approaches to improve branch divergence that favour using load balancing techniques employed to leverage the threads in warps and thread blocks to cooperatively process data elements.

Overall, while many techniques enable more efficient GPGPU irregular execution, a significant effort is required to implement these optimizations and reduce the GPGPU architecture shortcomings for irregular applications.

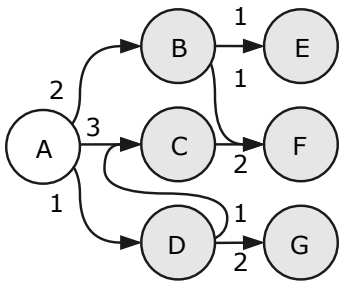
2.3 Graph processing algorithms on GPGPU architectures

Graphs are a fundamental data structure in mathematics that contain a set of objects in which some of them are connected or related in some manner. The objects represented are called *vertices* or *nodes* and the connections *links* or *edges*. Figure 2.5a shows the graphical representation of a Graph in which the nodes are labeled with letters and the edges between nodes are indicated with the arrows. Note that the graph represented is a *directed graph*, since the edges are represented with arrows thus the relationship between nodes only stands in the direction depicted. On the contrary, for an *undirected graph* the edges would connect a pair of nodes in both directions. Since graphs represent relationships between data, the interconnection and properties of graphs can vary wildly, it is specially relevant the connectivity of the edges in a graph. In particular the *out degree* of a node is the number of adjacent nodes (i.e. edges) to it.

Graphs are used to represent many different datasets specially representing topology or knowledge systems, such as: road networks, social networks, language syntax and grammar, DNA sequences, web-pages linking, and many more. Once data is represented in a graph, many graph algorithms can be utilized to analyze the data. Many problems in Machine Learning [93, 134], data analytics [172] and more areas can be described and solved using graph algorithms. These algorithms employ graphs that describe the relationships between elements on a given dataset of interest, and explore them in a specific manner to extract the desired information. GPGPU architectures can be used to accelerate graph processing by exploring in parallel multiple nodes and their connections (i.e. edges) of the graph.

The graph exploration on a typical GPGPU graph processing algorithm starts in a given node and moves to adjacent nodes by processing and traversing that node's edges. At this point, a new *frontier* (i.e. set of nodes or edges) is ready to be explored continuing this process iteratively until the whole connected graph is explored, or until the algorithm dictates it. Figure 2.6 shows how this process unfolds in a given iteration. Each element of the edge frontier array (i.e. indices) points to the position to access in the nodes array to fetch for the next frontier data and continue the graph exploration. The pseudo-code shows the type of irregular access performed on the nodes array, which is an intrinsic part of graph exploration algorithms and a cause of inefficiencies in GPGPU architectures. Additionally, graph-based algorithms exhibit further characteristics that are not amenable for GPGPU such as: low computation to memory access ratio [9], and irregular memory access patterns and poor data locality [94, 166] due to the unpredictable and irregular nature of the relationships expressed in a graph. Consequently, proper GPGPU resource utilization for graph exploration is hard to achieve.

To ameliorate the issues previously mentioned, CUDA/OpenCL implementations of graph



(a) Reference Graph

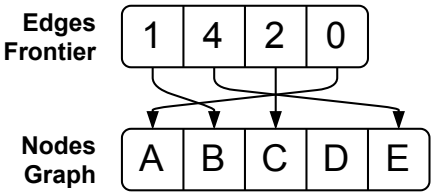
Nodes:	A B C D E F G
Adjacency Offsets:	0 3 5 6 8 8 8
Edges:	B C D E F F C G
Weights:	2 3 1 1 1 2 1 2

(b) CSR Representation

BFS Nodes:	A B C D E F G
BFS Distances: (start node = A)	0 1 1 1 2 2 2
SSSP Nodes:	A B C D E F G
SSSP Distances: (start node = A)	0 2 3 1 3 3 3

(c) Graph Exploration

Figure 2.5: Graph example (a) with its corresponding CSR representation (b) and the exploration results (i.e. computation result per node) when using BFS and SSSP on the starting node A (c). Graph (a) shows each node inside a circle with its corresponding label and each edge (arrow) with their corresponding weight. The CSR representation (b) contains the nodes and edges arrays, while it indicates with the adjacency offsets for each node the corresponding edges (with their corresponding weight value).




```
for i in range(0, N):
  idx = edge_front[i]
  out[i] = nodes[idx]
```

Figure 2.6: Example of a graph application processing an edge frontier and its irregular accesses generated when accessing the nodes in the graph. A pseudo-code example showcases the particular irregular access performed.

algorithms leverage several solutions. First, graph applications employ compact and efficient representation of the graph data structure, Compressed Sparse Row (CSR) [11] being one of the most popular formats. Figure 2.5 shows a reference graph with its corresponding CSR representation. This format consists of an array with all the nodes in the graph, two arrays containing all the edges and their corresponding weights respectively, and an array with the adjacency offsets. CSR enables efficient representation of the graph element relationships without incurring in large memory footprint and sparse representation which negatively impact performance. Second, GPGPU-based graph algorithms employ different approaches to avoid expanding duplicated nodes, loosely approaches or precise ones by using atomic operations. For example, the parallel processing of nodes A and D in the graph of Figure 2.5a would generate two copies of the same node C in a graph traversal kernel, since all the nodes in the frontier are

processed in parallel. GPGPU implementations eliminate/reduce the amount of duplicated nodes by loosely or accurately tracking already visited nodes. Finally, to deal with sparse and irregular memory accesses, GPGPU graph processing leverages stream compaction techniques [12] to gather the sparse data in contiguous memory locations, improving memory coalescing. The compacted array of nodes/edges is typically referred as the *frontier*, which is a contiguous space in memory containing the nodes/edges that are being explored in a given iteration of the algorithm.

In this thesis we focus on common graph algorithms, in particular Breadth-First Search (BFS) [99], Single-Source Shortest Paths (SSSP) [29] and PageRank (PR) [45], which are among the most widely used primitives for graph processing.

2.3.1 Breadth First Search (BFS)

Breadth-First Search (BFS) is one of the most important graph traversal algorithms that computes the minimum distance, in terms of traversed edges, from a given node to all the nodes in a graph (see Figure 2.5c). We use the state-of-the-art CUDA implementation of BFS proposed in [99], that includes several techniques to mitigate issues of GPU-based graph processing mentioned in Section 2.2. The exploration of a graph starts in a given node and an iterative process is done until all nodes are visited, the exploration advances as a wave-front from the starting node. Each iteration consists of an expansion phase and a contraction phase (see Figure 2.7).

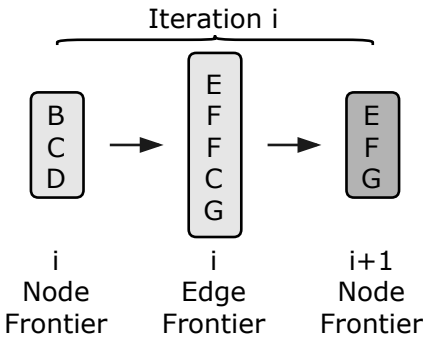


Figure 2.7: Execution of a given iteration of BFS on the graph in Figure 2.5a. The input node frontier generates an edge frontier which is evaluated and creates the next node frontier to process.

Expansion phase

It consumes the node frontier and generates the new edge frontier. Every node in the frontier is processed by an single thread, consequently the total amount of threads instantiated depends on the data to process. The first section of Figure 2.7 shows the data processed and generated when performing expansion on the graph 2.5a. The main challenge of this kernel is workload balancing, a direct approach to generate the edge frontier would be for each thread to create

the edges of their node, but different nodes exhibit largely different number of outgoing edges negatively effecting performance. Several mechanisms are used in [99] to cooperatively expand the edges of a node or several nodes between threads in a warp and threads in a thread block, this mechanism increases warp-level synchronizations but improves workload balance by a large extent.

Contraction phase

It consumes the edge frontier to create the new node frontier, while updating the information of different nodes during this process to keep track of the distance to the source node. Every edge in the frontier is processed by a single thread. The second section of Figure 2.7 shows the data processed and generated when performing the contraction phase on the graph 2.5a. The main challenge is detecting duplicated elements in the frontier, caused by the parallel nature of the algorithm and elements already visited in previous iterations. A straightforward approach would be to use atomic to update a tracking structure and then disable the duplicated threads, but the large amount of threads results in large overheads and make this approach unfeasible. A state-of-the-art solution proposed in [99] is to use a "best-effort" bitmask, i.e. updated without using atomic operations, which may yield false negatives due to race conditions but removes overheads of atomic operations.

2.3.2 Single Source Shortest Path (SSSP)

Single-Source Shortest Path (SSSP) computes the minimum cost from a source node to all the nodes in a graph, where the cost is the addition of the weights of the traversed edges (see Figure 2.5c). We use the CUDA implementation presented in [29], which is similar to BFS. However, the edge frontier is split into the "near" and "far" frontiers for paths with low and high cost respectively, based on a dynamically adjusted threshold. The most promising paths in the "near" frontier, with the lowest cost of traversal, are expanded first in order to reduce the cost of revisiting and updating nodes in following iterations of the algorithm.

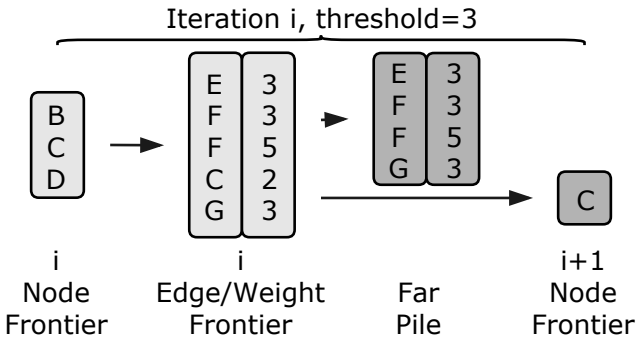


Figure 2.8: Execution of a given iteration of SSSP with a threshold=3 on the graph in Figure 2.5a. The input node frontier generates an edges/weight frontier which is broken down based on the threshold into two structures: the Far Pile and the next node frontier to process. This distinction improves SSSP performance on GPU architectures.

On each iteration, the expansion phase consumes the node frontier to generate the edge and weight frontiers. Next, the contraction phase stores high cost nodes into the "far" pile (see Figure 2.8), whereas low cost nodes are used to create the new node frontier. This process is repeated until the node frontier becomes empty. At this point, the threshold is updated and the contract phase starts consuming nodes from the "far" pile.

Expansion phase

It generates the new edge and weight frontiers, using the same mechanisms for workload balancing as described in Section 2.3.1 for BFS, but at the same time generating and additional frontier, i.e. the weight frontier. Every node in the frontier is processed by an individual thread. The first section of Figure 2.8 shows the data processed and generated when performing expansion on the graph 2.5a.

Contraction phases

It consumes the edge and weight frontiers and decides based on the threshold which nodes are to be processed on the Near frontier and which are to be deferred for later in the Far Pile. Every edge in the frontier is processed by a single thread. The second section of Figure 2.8 shows the data processed and generated when performing the contraction phase on the graph 2.5a. The main challenge is the detection and filtering of duplicated and visited edges. The simpler approach proposed for BFS does not suffice due to the necessity to process the additional weight frontier. The implementation proposed in [29] employs a lookup table with one entry per node in the graph. Each thread writes its ID to the corresponding lookup table entry and, after synchronization, only threads whose ID is stored in the lookup table are allowed to modify the new node frontier. This approach does not require atomic operation to access the lookup table, but it does to update the cost of the nodes (SSSP Distances in Figure 2.5c) with atomicMin to guarantee it is the shortest path.

2.3.3 PageRank (PR)

PageRank (PR) is a well-known graph primitive used in search engines [116] and recommendation systems first proposed by Google. PR initializes the graph with equal score to each node and then proceeds to iteratively compute the updated scores of all the graph nodes based on the following equation until the algorithm converges. Similar to SSSP, every node gets a weight (or rank), and each phase processes this information in a weight frontier. In the formula, α is a constant (i.e. dampening factor) and U_{deg} is the out degree of a node U (i.e. number of outgoing adjacent nodes):

$$V_{score} = \alpha + (1 - \alpha) \sum \frac{U_{score}}{U_{deg}}$$

We borrow the CUDA implementation of PR for a recommendation system proposed in [45].

Each iteration of the algorithm consists of four phases: Expansion, Rank Update, Dampening and Convergence Check.

Expansion phase

It consumes the node frontier and generates the edge frontier and weight (i.e. rank) frontier, using several workload balance mechanisms (see Section 2.3.1). The rank of each node is divided by its out degree.

Rank Update phase

It computes the new ranks using atomic addition operations. An atomic operation is issued for every edge in the graph since, unlike BFS or SSSP, all the nodes are considered active on every iteration of the algorithm.

Dampening phase

The dampening factor is applied to the rank of each node. This phase is well suited for execution on the GPU.

Convergence check phase

The ranks for the current iteration are compared with the ranks from the previous iteration, and the algorithm finishes if the maximum node-wise difference is smaller than a given epsilon value. As the dampening phase, convergence check phase is GPU-friendly.

3

Experimental Methodology

This chapter reviews the methodology, simulators, tools, programs and datasets used throughout the thesis to perform the evaluation of the different contributions presented. This chapter first reviews the complete simulation environment and tools used to evaluate metrics such as performance, energy consumption and area for our works. Afterwards, the chapter explores in detail the simulators developed and hardware modeling done for our works. Finally, attention is brought to the benchmark graph algorithms and graph datasets used throughout the thesis and the evaluation of our contributions.

3.1 Simulation Systems Integration

In the work presented in this thesis we evaluate the effect of our proposals in the performance of a GPGPU architecture. To do so we rely on the well-known and extensively used GPGPU-Sim 3.2.2 [6] simulator. GPGPU-Sim is a cycle-accurate simulator of GPGPU systems which allows to easily evaluate CUDA and OpenCL workloads while allowing to configure architectural parameters and extract relevant performance metrics to use for our research. In addition, GPGPU-Sim is integrated with GPUWattch [84], a validated energy model based upon McPAT which allows to obtain the area and energy requirements of the architectures and workloads evaluated.

We use GPGPU-Sim and GPUWattch in all our works by integrating either with our custom simulators or extending them with our techniques. We configure these simulators to match the characteristics of contemporary GPU architectures: a High-Performance NVIDIA GTX 980 [50] and a Low-Power NVIDIA Jetson TX1 [109]. The relevant configuration parameters and characteristics used to model this GPUs on GPGPU-Sim and GPUWattch are listed in Table 3.1.

Table 3.1: GPGPU-Sim and GPUWattch configuration parameters to model the High-Performance GTX980 and Low-Power Tegra X1 GPU systems.

Characteristic	GTX980	TX1
GPU & Architecture	NVIDIA GTX 980, Maxwell	NVIDIA Tegra X1, Maxwell
Frequency & Technology	1.27GHz, 28nm	1GHz, 28nm
Streaming Multiprocessor	16 SM, (total 2048 threads)	2 SM, (total 256 threads)
SM Shader Registers	64 K per core	64 K per core
SM Functional Units	128 EUs, 1 LD/ST per SM	128 EUs, 1 LD/ST per SM
SM Issue Schedulers	4 Warp Schedulers per SM	4 Warp Schedulers per SM
L1 data cache	32 KB, 4-assoc, 128 B lines	32 KB, 4-assoc, 128 B lines
L2 data cache	2 MB, 8-assoc, 128 B lines	256 KB, 8-assoc, 128 B lines
L1, L2 MSHRs	32/32 assoc, 8/4-merge	32/32 assoc, 8/4-merge
Shared Memory	64 KB standalone	64 KB standalone
Memory Partitions	4 (4 GDDR5 channels)	2 (2 LPDDR4 channels)
Main Memory Technology	4 GB GDDR5	4 GB LPDDR4
Main Memory Bandwidth	224 GB/s	25.6 GB/s

3.1.1 Stream Compaction Unit (SCU)

To evaluate our SCU we use the simulation systems depicted in Figure 3.1. The two main components we require to evaluate are the GPU and the SCU. The GPU performance is evaluated with GPGPU-Sim while power, energy and area measurements are obtained with GPUWattch. Both GPU simulators are configured to model an NVIDIA GTX 980 with 4GB GDDR5 and an NVIDIA Tegra X1 with 4GB LPDDR4 with the parameters listed in Table 3.1. Main memory power measurements for GDDR5 are obtained from GPUWattch's own power model while we use the Micron power model for LPDDR4 [151].

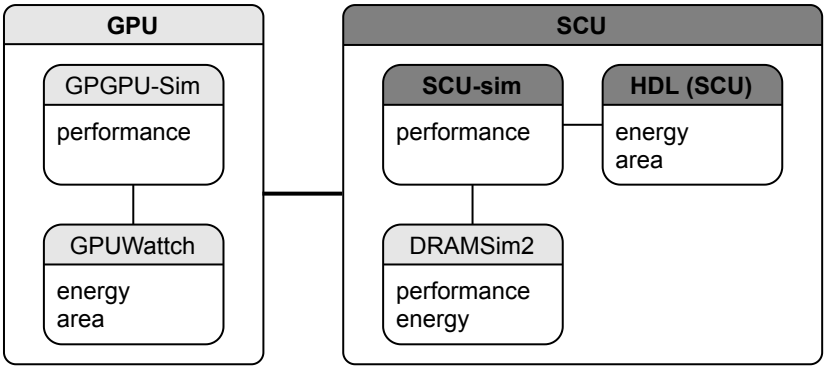


Figure 3.1: SCU complete simulation system comprising GPU simulation and SCU simulation to obtain performance, energy and area of the entire system. The darker color shows our contributions to the simulation system.

To evaluate the performance of the stream compaction operations we implement our own cycle-accurate simulator; the SCU-sim. The SCU-sim performs the workloads described by our proposed SCU API carrying out stream compaction operations consisting of efficient data

movements in the pipelined architecture we propose. The SCU architecture and simulator are scalable, as described in Section 3.2.1, to match the system performance and energy constrains of the different GPU architectures we evaluate. While the SCU-sim simulator enables performance evaluation, in order to obtain power, energy and area measurements of the SCU architecture, we implement it in Verilog Hardware description language (HDL) and synthesize it using the Synopsis Design Compiler [23] and the technology library of 32nm from Synopsys with low power configured at 0.78V. Additionally, we use CACTI [90] to characterize the cache and interconnection.

Finally, we integrate SCU-sim with the main memory simulator DramSim2 [130] to properly model main memory accesses, and we modified it to simulate a 4GB GDDR5 and a 4GB LPDDR4. With DramSim2 we are able to model the performance effect of main memory accesses and their energy requirements.

3.1.2 Irregular accesses Reorder Unit (IRU)

To evaluate our IRU contribution we use the simulation systems depicted in Figure 3.2. Since the IRU is an architectural extension of the GPU we directly extend the cycle-accurate GPGPU-Sim simulator with our proposal. Performance is obtained directly with the IRU extended GPGPU-Sim. Meanwhile, GPU power, energy and area is obtained with GPUWattch while IRU additional power, energy and area overheads are obtained by modeling hardware requirements with CACTI [90]. Both GPU simulators are configured with the parameters shown in Table 3.1 to model the NVIDIA GTX 980 target GPU.

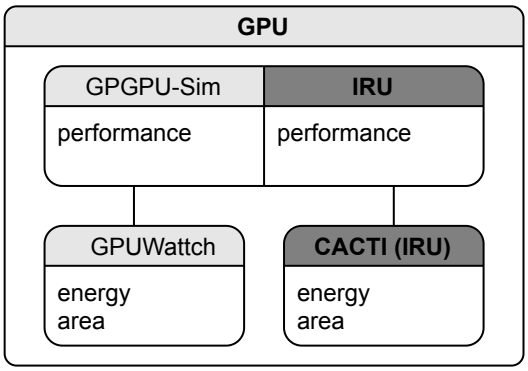


Figure 3.2: IRU simulation system extending the GPGPU-Sim simulator to obtain performance, energy and area of the IRU contribution. The darker color shows our contributions to the simulation system.

In order to implement our proposed IRU GPU operations, further changes are performed to the GPGPU-Sim binary parser component and to the control logic with the objective to support and integrate our IRU proposed operations and achieve seamless execution of IRU-enhanced workloads.

3.1.3 IRU-enhanced SCU (ISCU)

To evaluate our ISCU contribution we use the simulation system depicted in Figure 3.3. We combine both SCU and IRU simulation systems while performing the additions and modifications described in Chapter 6 to both the SCU and IRU hardware to utilize them together in the ISCU system. Similarly, GPU performance is evaluated with the IRU-extended GPGPU-Sim and power, energy and area measurements are obtained with GPUWattch, while IRU related overheads are accounted with CACTI. At the same time, we use SCU-sim to model performance and the Verilog implementation to obtain power, energy and area overheads. We evaluate our proposal on top of an NVIDIA GTX 980 GPU system with the parameters listed in Table 3.1.

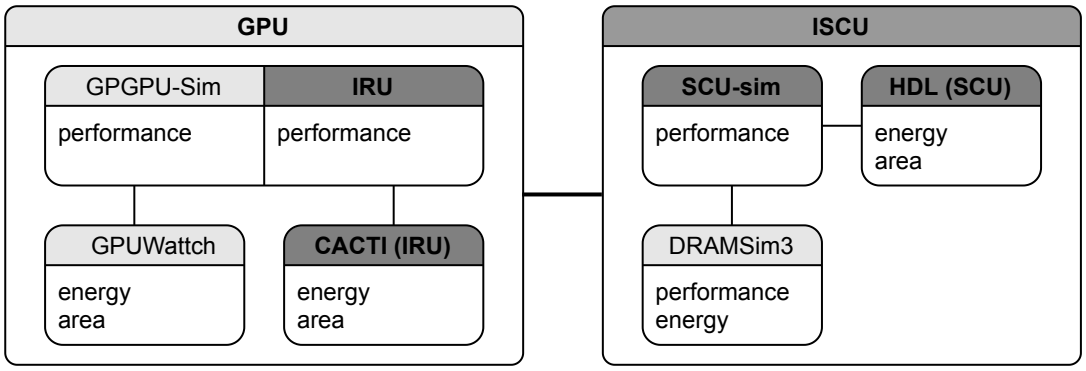


Figure 3.3: ISCU complete simulation system comprising IRU-extended GPU simulation and SCU simulation to obtain performance, energy and area of the entire system. The darker color shows our contributions to the simulation system.

Finally, to complete the SCU simulation system we update the main memory system to use the recent DRAMSim3 [89], which provides improved and more accurate evaluation of main memory and allows us to evaluate a GDDR5 configured with 4 channels and 224 GB/s of bandwidth.

3.2 Hardware Modeling and Evaluation

In this section we review the hardware modeling of our contributions. We have developed the SCU, an architecture for efficiently performing stream compaction operations, the IRU, an architectural extension of the GPU to improve irregular accesses, and we have leveraged them together to improve overall GPU graph processing. While in-depth details about the architecture of each contribution are presented in their respective chapter in this thesis, this section reviews the configuration and characterization of the architectures.

3.2.1 Stream Compaction Unit (SCU)

To model the SCU architecture described in Chapter 4 we have developed the SCU-sim cycle-accurate simulator. The configuration of the main components is shown in Table 3.2,

whereas some additional parameters are included in Table 3.3. We match the SCU frequency to the clock rate of the target GPU, being 1.27GHz and 1 GHz for GTX980 and TX1 respectively. We use a 5 KB FIFO to buffer the vector parameters of the SCU operations, while the *Data Fetch* component includes a 38 KB FIFO requests buffer. The filtering and grouping operations use a flexible and re-configurable in-memory hash table detailed in Table 3.3, in addition to a 18 KB request buffer. Finally, the coalescing units hold up to 32 in-flights requests with a merge window of 4 elements.

Table 3.2: SCU hardware parameters.

Component	Requirements
Frequency	1.27GHz / 1GHz
Technology	32 nm
Vector Buffering	5 KB
FIFO Requests Buffer	38 KB
Hash Request Buffer	18 KB
Coalescing Unit	32 assoc, 4-merge

The SCU architecture and their components are implemented in Verilog. In order to obtain area and energy consumption we synthesize it using the Synopsis Design Compiler [23] and the technology library of 32nm from Synopsys with low power configured at 0.78V. Additionally, we use CACTI [90] to characterize the cache and interconnection.

SCU Scalability

The large variability of High-Performance GPUs optimized for performance at the expense of power dissipation, compared to Low-Power GPUs that provide more modest performance while keeping energy consumption extremely low require an adaptable SCU architecture. A fixed design of the SCU will be undersized or oversized in some circumstances, hence the need to resize the SCU architecture in order to adjust its performance, area and energy consumption to the requirements of the target segment.

Table 3.3: SCU scalability parameters selection for the GTX980 and TX1 GPU.

Component	GTX980	TX1
Pipeline Width	4 elements/cycle	1 elements/cycle
Filtering BFS Hash	1 MB, 16-way, 4 bytes/line	132 KB, 16-way, 4 bytes/line
Filtering SSSP Hash	1.5 MB, 16-way, 8 bytes/line	192 KB, 16-way, 8 bytes/line
Grouping SSSP Hash	1.2 MB, 16-way, 32 bytes/line	144 KB, 16-way, 32 bytes/line

To satisfy this re-configurable objective, we facilitate two configuration parameters in our design. The first one is the pipeline width of the SCU, i.e. the number of elements (nodes/edges) that can be processed per cycle. This parameter is included in the RTL code and the user can set an appropriate value before synthesizing the SCU. We found that a pipeline width of 1 provides a good trade-off between area and performance for the low-power TX1 GPU, whereas a pipeline width of 4 is required to outperform a high-performance GPU such as the GTX980. The second

parameter to achieve scalability is the size of the hash tables employed for filtering and grouping operations. Larger sizes potentially provide a more effective filtering and grouping, but may have a negative impact on performance if the L2 cache is too small. These parameters can be set by the user at runtime. Finally, Table 3.3 shows the parameters used for each GPU system.

3.2.2 Irregular accesses Reorder Unit (IRU)

To model the IRU architecture described in Chapter 5 we have extended the architecture of the GPGPU-Sim simulator. Additionally to the introduced IRU hardware, to properly integrate the IRU into the GPGPU simulator we modified the GPU decoding to add our new instructions to the ISA, extended the control logic to support them, as well as incorporating small modifications to the LD/ST unit to handle the new instructions.

Table 3.4: IRU hardware requirements per partition.

Component	Requirements
Requests Buffer	2 KB
Prefetcher Buffer	1.7 KB
Classifier Buffer	1.2 KB
Ring Buffer	2.8 KB
Hash Data	80 KB

The main configurable components of an IRU partition are summarized in Table 3.4 together with the configuration values used in our evaluation set to maximize performance while constraining energy and area overheads and general architectural overheads. Each partition of the IRU uses a 2 KB FIFO to buffer warp requests, 1.7 KB buffering of prefetching data for 8 on-the-fly simultaneous prefetches, prefetching limit set to avoid saturating memory bandwidth and degrading performance. A buffer of 1.2 KB is used internally in the *Classifier* block to determine the data destination. The *Ring interconnection* requires a total of 2.8 KB space for buffering. The main component of the IRU is the *Reordering Hash*, which is a direct mapping hash table with 1024 sets, split in 4 physical partitions. Each IRU partition is 2-way banked and holds 256 sets which amount to a total of 80 KB, significantly smaller than the 512 KB of the L2 partition. Since the IRU is mostly comprised of SRAM elements without complex logic or execution unit we model area and energy consumption using CACTI [90] with a node technology of 32 nm.

3.2.3 IRU-enhanced SCU (ISCU)

To model the ISCU system described in Chapter 6 we incorporate both previously described architectures; the SCU architecture and the IRU GPU architecture extension. Some small changes described in Section 6.2 are performed to the previous architectures to enable the SCU to perform pre-processing optimizations utilizing the IRU. In addition, further modifications are done to the IRU to disable the prefetching and allowing the SCU to operate it directly. The configuration of different architectural components is shown in Table 3.2 and Table 3.4, while the in-memory hashing mechanisms described in Table 3.3 are no longer required.

3.3 Graph Processing Datasets

This section reviews the graph processing algorithms and graph datasets used in the evaluations of the contributions presented in this thesis.

3.3.1 Graph Processing Algorithms

To evaluate our contributions we have used state-of-the-art GPGPU implementations of BFS [99], SSSP [29], and PageRank [45] graph algorithms used in frameworks such as Gunrock [161]. Our implementations are push algorithms (i.e. data transmitted from source to destinations elements) instead of pull algorithms (i.e. data retrieved by destination from sources). Section 2.3 thoroughly describes the details and inner working of the implementations. We select these graph algorithms due to their vast use in applications and in academia as well as being graph primitives that serve as core components of larger graph processing applications.

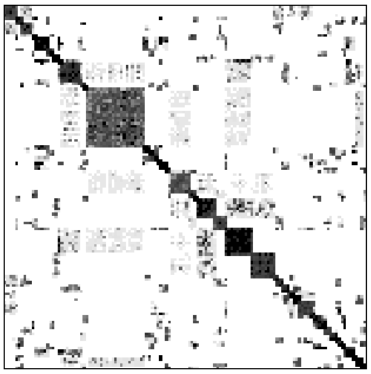
We have implemented and evaluated the implementations using CUDA 7.5 toolkit on a system with an Intel Core i7 6700K and a NVIDIA GTX 980 GPU (i.e. physical, not simulated). For the evaluation of our contributions we have instead used CUDA 4.2 since it is the latest version supported by GPGPU-Sim 3.2.2. Version 4.2 of CUDA does support fewer atomic operations which we have had to overcome with clever approaches to achieve the same behaviour without penalizing performance.

3.3.2 Graph Datasets

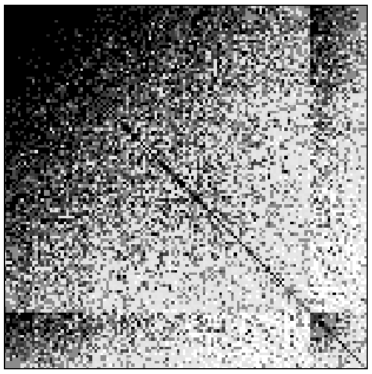
Graph datasets and algorithm are very diverse and their characteristics largely affect the performance of graph exploration. We have evaluated our graph algorithms with the benchmarks datasets shown in Table 3.5, collected from well-known repositories of research graph datasets [30, 33]. Figure 3.4 shows the sparsity plots of the graph indicating their inter-connections. These graphs are representative of different application domains with varied sizes, characteristics and degrees of connectivity which in total represent 18 different combinations of graph algorithms and datasets. Finally, graph exploration algorithms generate and process large amounts of data throughout their execution, this factor together with simulation performance (up to a week) and limited GPU memory size (4GB to allocate data and generation of traces) imposed a limit in the size of graphs we were capable of evaluating.

Table 3.5: Benchmark graph datasets collected from well-known research repositories [30, 33].

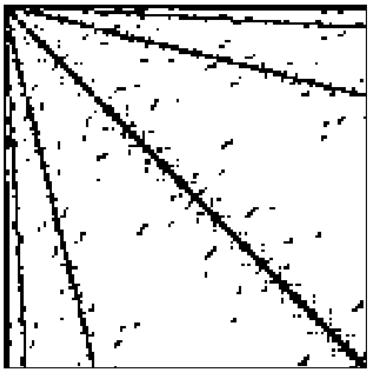
Graph	Description	Nodes(10^3)	Edges(10^6)	Avg.Degree
ca [30]	California road network	710	3.48	9.8
cond [30]	Collaboration network, arxiv.org	40	0.35	17.4
delaunay [33]	Delaunay triangulation	524	3.4	12
human [30]	Human gene regulatory network	22	24.6	2214
kron [33]	Graph500, Synthetic Graph	262	21	156
msdoor [30]	Mesh of 3D object	415	20.2	97.3



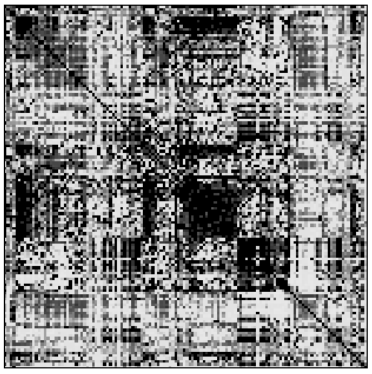
(a) California Road Network (ca).



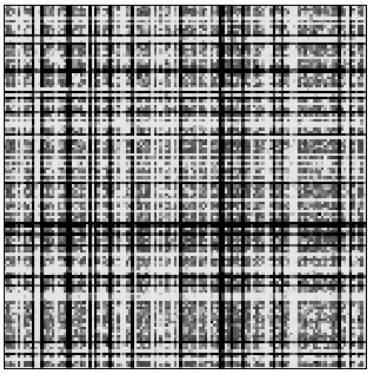
(b) Collaboration Network, arxiv.org (cond).



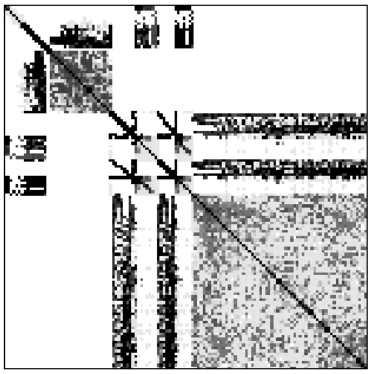
(c) Delaunay Triangulation (delaunay).



(d) Human Gene Regulatory Network (human).



(e) Graph500, Synthetic Graph (kron).



(f) Mesh of 3D Object (msdoor).

Figure 3.4: Sparsity plots showcasing inter-connections between the nodes of the graphs introduced in Table 3.5 and gathered from the graph repository [30]. The gray-scale indicates the degree of connectivity of each of the nodes on the graph. The sparsity plots help to understand the high diversity in locality and structure of the graphs.

4

Energy-Efficient Graph Processing by Boosting Stream Compaction

This chapter comprehensively describes the concepts and details of the Stream Compaction Unit (SCU) contribution introduced in Section 1.4.1. The chapter is organized as follows. First, Section 4.1 reviews and expands the motivation of our SCU proposal. Section 4.2 presents the basic architecture of the SCU and gives an overview of the main functionality and inner working. Afterwards, Section 4.3 describes the important SCU extensions for filtering duplicated elements and improving memory coalescing. Section 4.4 presents the experimental results obtained. Finally, Section 4.5 sums up the main conclusions of this work.

4.1 Introduction

Graph processing algorithms are key in many emerging applications in areas such as machine learning and data analytics. As reviewed in Section 1.2, the processing of large scale graphs exhibits a high degree of parallelism, but the irregular memory access pattern and low computation to memory access ratio lead to poor GPGPU efficiency due to memory divergence, as GPGPU architectures are optimized for compute-intensive workloads with regular memory access patterns. To ameliorate these issues, GPGPU applications perform stream compaction operations to extract the subset of active elements (i.e. nodes/edges) and perform subsequent steps on the compacted dataset. The end result is that subsequent graph processing on the compacted data exhibits more regular memory access patterns.

Stream Compaction is a common primitive used to filter out unwanted elements in sparse data, with the aim of improving the performance of parallel algorithms that work best on the compacted dataset. Figure 4.1 shows the average percentage of time spent on stream compaction

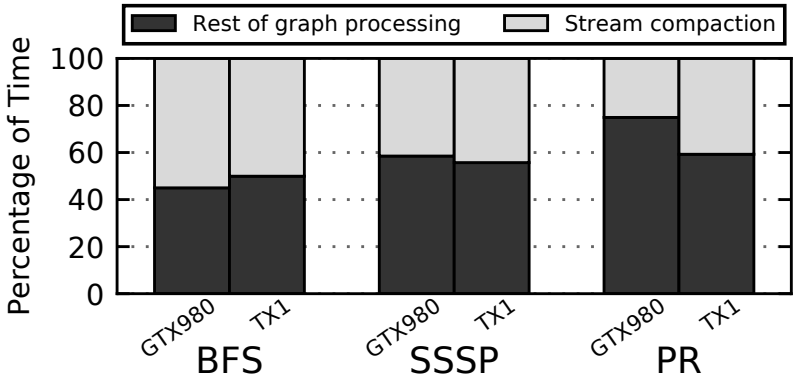


Figure 4.1: Breakdown of the average execution time for several applications (Table 3.5) and three graph primitives (BFS, SSSP and PR). Measured on an NVIDIA GTX 980 and NVIDIA Tegra X1. Darker color indicates execution time spent on graph processing, while lighter color highlights time performing stream compaction operations.

for three commonly used graph kernels: Breadth First Search (BFS), Single Source Shortest Path (SSSP) and PageRank (PR). As it can be seen, stream compaction represents between 25% to 55% of the total execution time. Although state-of-the-art CUDA implementations of BFS [99] and SSSP [29] mix compaction and processing, we split them just for the purpose of making Figure 4.1, as our best effort to quantify the cost of compaction.

In this work we claim that GPGPU architectures are not efficient for stream compaction workloads for several reasons, as summarized in Section 1.4.1. First, stream compaction consists of sparse memory accesses with poor locality that fetch the elements (nodes/edges) to be compacted, resulting in very low memory coalescing and reducing GPU efficiency by a large extent. Second, stream compaction has an extremely low computation to memory access ratio, as it primarily consists of load and store instructions to move data around in main memory. Streaming Multiprocessors (SMs) are optimized for compute-intensive workloads, including hundreds of functional units that are largely underutilized during the stream compaction stage.

Since GPU architectures are not designed to efficiently support compaction operations, we propose to extend the GPU with a novel unit tailored to the requirements of stream compaction. We term this hardware as the Stream Compaction Unit (SCU), which is a small unit tightly integrated in the GPU that efficiently gathers the active elements into a compacted array in memory. By providing such integration and a simple API, we offload compaction operations to the SCU to achieve higher performance and energy efficiency for this compaction phase, while maximizing the effectiveness of the streaming multiprocessors for the phases of the algorithm that work on the compacted dataset. Additionally, the SCU performs filtering of repeated and visited nodes during the compaction process, significantly reducing GPU workload, and writes the compacted elements in an order that improves memory coalescing and reduces memory divergence.

Finally, we evaluate the performance of a state-of-the-art GPGPU architecture extended with our SCU for a wide variety of applications. Results show that for high-performance and for

low-power GPU systems the SCU achieves speedups of 1.37x and 2.32x, 84.7% and 69% energy savings, and an area increase of 3.3% and 4.1% respectively.

To summarize, the main contributions presented in this chapter are the following:

- We characterize the performance and energy consumption of the stream compaction operation on a modern GPU architecture, showing that it takes more than 50% of the execution time and more than 45% of the energy consumption for graph processing applications.
- We propose the SCU, a novel unit that is tailored to the requirements of stream compaction, and describe how this unit can be integrated in existing GPGPU architectures.
- We extend the SCU to perform filtering of duplicated nodes, which removes 75% of GPU workload on average, and to rearrange the compacted data to reduce memory divergence, which improves memory coalescing by 27%.
- Overall, the high-performance and low-power GPU designs including our SCU unit achieve speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively on average for several graph-based applications. The SCU represents a small area overhead of 3.3% and 4.1% respectively.

4.2 Stream Compaction Unit

In this section, we propose a Stream Compaction Unit (SCU) tailored to the requirements of graph applications. Compacting the active nodes/edges in consecutive memory locations is key for achieving high utilization of GPU resources. However, the GPU is inefficient performing stream compaction operations, as it only requires data movements with no computation on the data, but GPU architectures are optimized for compute intensive workloads. Furthermore, the memory accesses for compacting the data are typically sparse and highly irregular, leading to poor memory coalescing. As shown in Figure 4.1, the stream compaction operation takes more than 40% of GPU time in several graph applications.

We propose to offload the stream compaction operations to a specialized unit, the SCU. The SCU is an efficient, compact and small footprint unit that is attached to the streaming multiprocessor interconnection network as shown in Figure 4.2. The SCU performs data compaction in a sequential manner, avoiding synchronization and work distribution overheads, and operates with just the minimum hardware requirements to perform data movement operations for stream compaction workloads.

Our proposed graph processing approach exploits the parallelism of the GPU to explore a graph while making use of the SCU to perform the data compaction operations. Once the compaction phase of the algorithm starts, SCU operations are issued, and the data compaction is performed on the sparse data in memory and compacted into a destination array. Once the operation concludes, the compacted data is available to the GPU which resumes execution continuing the graph exploration.

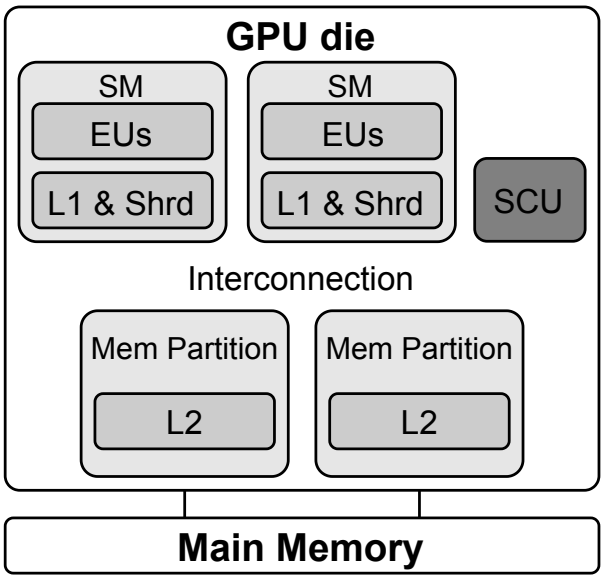


Figure 4.2: Overview of a GPGPU architecture featuring a SCU attached to the interconnection. The depicted GPU shows 2 SM similar to an NVIDIA Tegra X1.

4.2.1 SCU Compaction Operations

The SCU is a programmable unit which includes a number of generic data compaction operations that allow a complete implementation of stream compaction. Figure 4.3 shows the operations supported by the SCU. All SCU operations have some parameters which are omitted from the figure for the sake of simplicity: the size of the data and the number of elements on which to operate. The SCU implements the following operations:

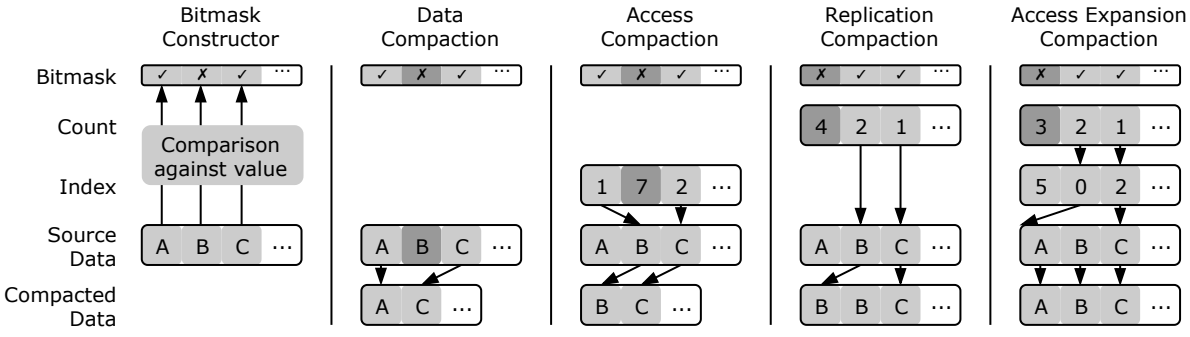


Figure 4.3: SCU operations required to implement stream compaction capabilities, illustrated with the data that each operation uses and generates. Arrow direction indicates flow of data.

- **Bitmask Constructor:** Generates a bitmask vector used by other operations. It requires a reference value and a comparison operation. It creates a bitmask vector for which each

bit set to 1 if the element in the input data evaluates to true using the comparison operator and the reference value, and to 0 otherwise.

- **Data Compaction:** Accesses sparse data sequentially and filters out the unwanted elements using the bitmask vector. The output at the destination contains only valid elements preserving the original order.
- **Access Compaction:** Accesses a sparse index vector sequentially and filters out the unwanted elements using the bitmask vector. The output at the destination contains only valid elements preserving the original order.
- **Replication Compaction:** An extension of the Data Compaction operation, which operates with the count vector. This vector is used to indicate how many times each element in the sparse data will be replicated in the output destination. The output destination contains only the valid elements, but each element is replicated by the amount of times indicated by its corresponding counter.
- **Access Expansion Compaction:** Uses both the indexes and count vectors. It is an extension of the Access Compaction operation that copies a number of consecutive elements instead of only one element from the sparse data indicated by the corresponding indexes vector entry. The number of elements to gather is determined by corresponding entry in the count vector.

4.2.2 Hardware Pipeline

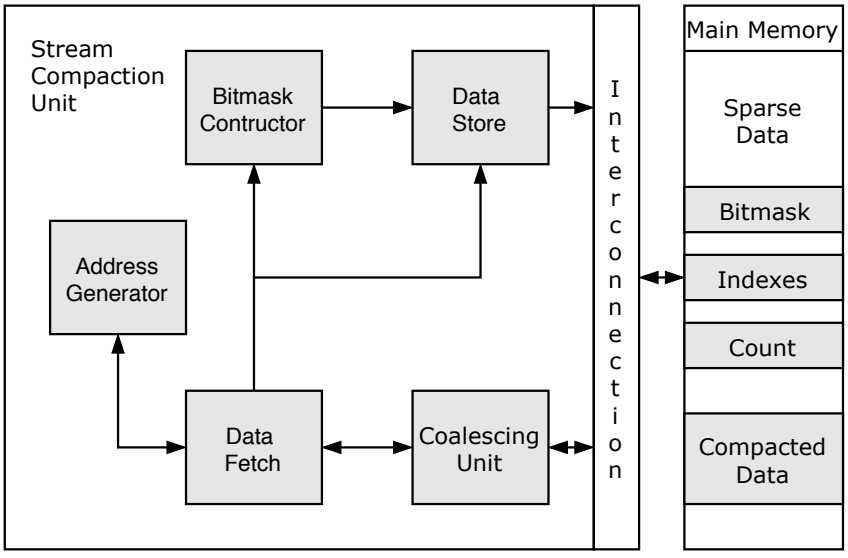


Figure 4.4: Overview of the baseline pipelined architecture of the Stream Compaction Unit with the connection of the different components as well as its interconnection to the main memory. The rightmost column shows the data used for the SCU operations allocated in Main Memory.

The SCU implements the operations previously described with the hardware illustrated in Figure 4.4. The SCU consists of five different functional units. An operation begins by configuring the main component, the Address Generator.

- **Address Generator:** It is configured at the beginning of each operation with the corresponding parameters, and begins execution of the operation by generating the addresses of the data to be compacted, as well as, if needed, the other vector parameters: bitmask, indexes and count. The generated addresses are stored on a small buffer to avoid stalls.
- **Data Fetch:** This is a straightforward component which generates memory requests to the addresses generated by the Address Generation component. The requests are sent to the Coalescing Unit and the order of requests is preserved. When the data is received, it is forwarded to either the Bitmask Constructor or the Data Store component, respecting the FIFO order. For this purpose, it uses a FIFO queue that stores the data until it can be forwarded to the consumer component.
- **Coalescing Unit:** This unit coalesces read memory requests to the same cache memory block in order to reduce congestion of the interconnection and memory requests to upper levels of the memory hierarchy, which reduces energy consumption.
- **Bitmask Constructor:** It is the specialized component used by the bitmask constructor operation to generate the bitmask vector that is used by other operations. Contains logic to perform comparison operations of a reference value against each of the elements that it receives.
- **Data Store:** It is the component that receives the data generated by the other components and generates the consequent write memory requests. Since data is stored in consecutive memory addresses, this unit includes a simple coalescing of write operations to minimize requests and interconnection network traffic.

4.2.3 Breadth-First Search with the SCU

Section 2.3.1 describes the state-of-the-art implementation of BFS for GPU architectures, that consists of two phases: expansion and contraction. Both phases of BFS include compaction operations that can be offloaded to the SCU. Figure 4.5 summarizes the modifications.

```
1  void BFS_Expand (node_frontier) {
2      indexes, count = preparationGPU (node_frontier);
3      edge_frontier = accessExpansionCompactionSCU (edges, indexes,
       count);
4      return edge_frontier;
5  }
6
7  void BFS_Contract (edge_frontier) {
8      bitmask = BFS_contractionGPU (edge_frontier);
9      node_frontier = dataCompactionSCU (edge_frontier, bitmask);
10     return node_frontier;
11 }
```

Figure 4.5: Pseudo-code of GPGPU BFS program modified to use the SCU to offload stream compaction operations.

Expansion phase:

The main workload of this phase is offloaded to the SCU using the Access Expansion Compaction operation. The indexes and count vectors are efficiently prepared by the GPU as it requires contiguous memory accesses. Each entry in the indexes vector represents the offset in the edges vector where the first edge of the corresponding node is stored. The count vector stores for a node its number of edges.

Contraction phase:

This phase takes as an input the edge frontier generated by the expansion phase and generates the new node frontier. It filters out duplicated edges and already visited nodes. The GPU is responsible for generating the mask that will be used for the filtering. Then, the compaction of valid nodes is offloaded to the SCU, by using Data Compaction operation that has the edge frontier and the bitmask vector as inputs.

4.2.4 Single-Source Shortest Paths with the SCU

Section 2.3.2 reviews the optimized implementation of SSSP used in this thesis. Similar to BFS, expansion and the two contraction phases of the algorithm include stream compaction operations that can be offloaded to the SCU. Figure 4.6 summarizes the modifications.

```
1  void SSSP_Expand (node_frontier) {
2      indexes, count = preparationGPU (node_frontier);
3      edge_frontier =
4              accessExpansionCompactionSCU (edges, indexes, count);
5      weight_frontier =
6              accessExpansionCompactionSCU (weights, indexes, count);
7      weight_frontier += replicationCompactionSCU (weights, count);
8      return edge_frontier, weight_frontier;
9  }
10
11 void SSSP_Contract (edge_frontier, weight_frontier, threshold) {
12     bitmask_near, bitmask_far =
13         SSSP_contractionGPU(edge_frontier, weight_frontier, threshold);
14     node_frontier = dataCompactionSCU (edge_frontier, bitmask_near);
15     farPileEdges = dataCompactionSCU (edge_frontier, bitmask_far);
16     farPileWeights = dataCompactionSCU (weight_frontier, bitmask_far);
17     return node_frontier;
18 }
```

Figure 4.6: Pseudo-code of GPGPU SSSP program modified to use the SCU to offload stream compaction operations.

Expansion phase:

The GPU generates the indexes and count vectors, then the SCU generates the edge frontier. Next, the SCU has to generate the weights vector corresponding to the edge frontier. This vector contains the costs associated with the edges which are obtained using two operations: an Access Expansion Compaction and a Replication Compaction operation. The former operation generates the weights associated to each edge, and the latter adds its accumulated cost.

Contraction phases:

This phase takes as an input the edge frontier and filters out duplicated edges and already visited nodes, updates the node's information, and compacts valid nodes of the next frontier. Edges with accumulated cost higher than the current iteration threshold are pushed to the back of the "far" pile. The operations that are offloaded to the SCU are the compaction of valid nodes and the compaction of high cost edges in the "far" pile. In both cases, a Data Compaction operation is used. The GPU is responsible for computing the bitmask, both for low-cost and high-cost edges (near and far respectively) that the SCU will use for filtering.

4.2.5 PageRank with the SCU

The state-of-the-art GPU implementation of Pagerank (PR) is described in Section 2.3.3. In this case, only the expansion phase of PR performs stream compaction operations. PR does not operate with a node frontier, since it explores all the nodes and edges on every iteration of the algorithm. Figure 4.7 summarizes the modifications.

```
1 void PR_Expand (nodes) {
2     indexes, count = preparationGPU (nodes);
3     edge_frontier =
4             accessExpansionCompactionSCU (edges, indexes, count);
5     weight_frontier = replicationCompactionSCU (weights, count);
6     return edge_frontier, weight_frontier;
7 }
```

Figure 4.7: Pseudo-code of GPGPU PR program modified to use the SCU to offload stream compaction operations.

Expansion phase:

The GPU creates the indexes, count and weight vectors. Afterwards the SCU generates the edge frontier, in the same way as it is done for BFS. Finally, the weight frontier is generated using the pre-processed weight vector that contains the original weight of each node, but divided by the output degree of the node, so that the SCU can replicate this value for each edge with a Replication Compaction operation.

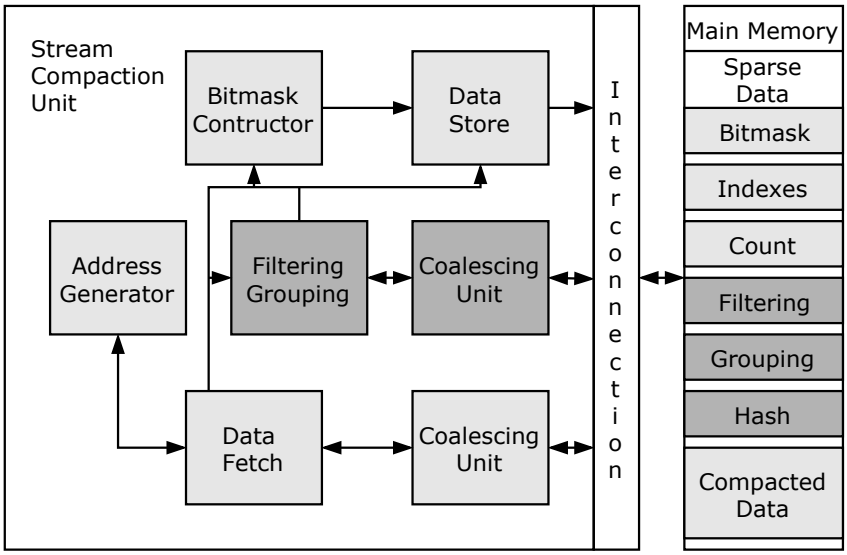


Figure 4.8: Improved pipelined architecture of the SCU. The darker color highlights the extension additions of the filtering and grouping hardware enabling their corresponding operations, as well as an extra coalescing unit. The rightmost column includes the additional data which is allocated in Main Memory and is required for the SCU pre-processing operations featuring the filtering/grouping vectors and the in-memory hash.

4.3 Filtering and Grouping

The Stream Compaction Unit (SCU) presented in Section 4.2 efficiently performs data compaction operations, while the remaining graph processing is performed by the GPU. The GPU graph processing has an important overhead due to nodes/edges duplication. Duplicated elements are a byproduct of parallel graph exploration on GPU architectures. Removing duplicates in the GPU requires costly mechanisms such as large lookup tables or atomic operations. Furthermore, GPU processing is very sensitive to the effectiveness of memory coalescing.

We propose to improve the SCU with the capability of further processing and delivering the compacted data in a more GPU-friendly manner. More specifically, we extend the SCU to remove duplicates and reorder compacted data elements to make them more effective for memory coalescing. We refer to this reorder step as grouping, since it is not a complete order but an order that tries to maximize the effectiveness of memory coalescing by grouping, i.e. storing together, edges whose destination nodes are in the same cache line. To this end, we incorporate a Filtering/Grouping unit and an additional coalescing unit as depicted in Figure 4.8.

4.3.1 Filtering/Grouping Unit

The Filtering/Grouping unit operates by storing each graph element (i.e. edge or node) into a hash table resident in main memory and cached in the L2. By placing the hash in memory we are able to reconfigure the hash organization to target Filtering or Grouping operations, since the requirements for both may be different. In addition, using existing memory does not require any

additional hardware. For the filtering operation, the hash table provides a low-cost mechanism to loosely remove duplicates. Each new edge/node probes the hash table and is discarded if a previous occurrence of the same node/edge is found. To simplify the implementation, in case of collisions the corresponding hash table entry is overwritten. This means that false negatives are possible, but it largely simplifies the cost of the implementation while removing most of the duplicates with relatively small hash table sizes as shown in Section 4.4. For the grouping operation, the hash table is used to create groups of edges whose destination node lies in the same cache line, in order to store them together in the compacted array.

The filtering and grouping of compacted data is done by the SCU in a two step process. In the first step, the SCU performs a compaction operation and identifies the duplicated elements (i.e. generates a bitmask vector indicating the filtered elements) and the reordering required for the grouping (i.e. a reordering vector with indexes indicating the new order of the data). In the second step, the SCU uses the generated data to perform filtering and grouping on the compacted data. All compaction operations shown in Figure 4.3 can generate and operate with filtering and grouping data.

As an example, the compaction operation could be performing the expansion of the node frontier to generate the new edge frontier. The first step creates filtering and reordering information for each of the elements on the edge frontier. Note that the first operation does not generate the new edge frontier but the filtering or grouping information instead. Next, the second step employs the previously generated information and creates the edge frontier compacted, reordered and without the duplicated elements.

4.3.2 Filtering Operation

We use two filtering schemes: filtering by unique element (useful for BFS), filtering by unique-best cost (useful for SSSP). The hash table is configured with 4 bytes per block for BFS and or 8 bytes per block for SSSP. Further details are listed in Table 3.2.

The filtering operation generates a bitmask vector which contains a bit for every output data element. Each bit is set to one if the element is to be kept, or zero otherwise. The filtering operation works as follows. For each element (node or edge) to be compacted, the SCU computes its hash table entry by applying a hash function to its ID. If the corresponding hash table entry is empty, the element ID is stored in the hash table and the corresponding bitmask entry is set to one. If the same element ID is found in the hash table entry a duplicated node/edge is detected and, hence, the element is discarded (bitmask entry for that element is set to zero). In case a different element ID is found, the older element is evicted and the new element ID is stored in the hash table. Note that since we overwrite some elements in case of collisions the removal of duplicates is not complete. However, this implementation provides a good trade-off between complexity and effectiveness in removing duplicates.

The above process describes the filtering scheme of unique elements. In the case of unique-best cost filtering, an additional cost value is stored in the hash table. On a hit, further processing is done: if the element has a better cost, it overwrites the cost in the hash table entry.

4.3.3 Grouping Operation

Grouping is achieved using the same hash table with a different configuration. We use a hash table entry to store a number of elements that access the same memory block, which in our system shown in Table 3.2 can hold up to 32 elements of 4 bytes (line size of L2 cache). In our hashing scheme, however, each hash table entry is limited to grouping 8 elements of 4 bytes (i.e. it creates groups of 8 elements at most). We have experimentally observed that it is better to reduce the number of elements per group to 8, since more elements would imply to reduce the number of sets kept in the hash table for the same total capacity. Furthermore, in sparse datasets, increasing the number of elements per group to 32 provides negligible benefits even if storage was unbounded, as it is quite hard to fill them up.

The output of the grouping operation is a vector that indicates for each input element which order (i.e. position) it will occupy in the compacted array. The grouping process works as follows. For each element, the SCU computes the memory block (i.e. cache line) of the node/edge being processed. Next, a hash function is applied to the memory block number to group together elements that require the same memory block, with the aim of improving memory coalescing on the GPU. If the corresponding memory block is found in the hash table, the new element is added to the hash table entry. If the entry is occupied by a different memory block, the older block is evicted and the elements it contains are written in the output vector to guarantee that they will be stored together in the compacted array. Again, this scheme does not guarantee that all the elements that require the same memory block are stored together, but it is highly effective in practice while being amenable for hardware implementation.

4.3.4 Breadth-First Search with the Enhanced SCU

Filtering out duplicated elements is beneficial for both the expansion and contraction phases of the BFS algorithm. Grouping is also applicable, but interferes with the warp culling filtering efforts done in the GPU processing, which lowers its effectiveness and results in increased workload, largely reducing the performance benefits due to the improved memory coalescing. Filtering reduces the GPU workload, in terms of edges and nodes, to 14% of the original workload on average. Shown on Figure 4.9, the required changes are the following:

Expansion phase:

Performs the filtering directly when processing the data generating the filtered edge frontier. It does not require the use of the filtering vector which would be used to apply the filtering detected to multiple compaction operations.

Contraction phase:

Performs the filtering directly when processing the data and generates the final filtered node frontier. Note that this filtering is applied because the filtering done by BFS is not complete (as

```
void BFS_Expand (node_frontier) {
    indexes,count = BFS_preparationGPU (node_frontier);
    edge_frontier = accessExpansionCompactionSCU
        (edges, indexes, count, do_filter);
    return edge_frontier;
}

void BFS_Contract (edge_frontier) {
    bitmask = BFS_contractionGPU (edge_frontier);
    node_frontier = dataCompactionSCU
        (edge_frontier, bitmask, do_filter);
    return node_frontier;
}
```

Figure 4.9: Pseudo-code of the additional operations for a GPGPU BFS program to use the enhanced SCU.

in SSSP). Otherwise, the filtering of the edge frontier would be done on the GPU when doing the graph exploration and further filtering at the SCU would not provide any benefit aside from atomic synchronization overheads reduction.

4.3.5 Single-Source Shortest Paths with the Enhanced SCU

Filtering out of duplicated elements is beneficial for both the expansion and contraction phases of the SSSP algorithm. Additionally, unlike BFS, the grouping does not interfere with the GPU filtering, and the coalescing improvement results in a net gain in performance. The SCU reduces the GPU workload (i.e. nodes and edges) to 22% of the original workload on average, and improves the coalescing effectiveness by a factor of 27%. Shown on Figure 4.10, the required changes are the following:

Expansion phase:

Two additional Access Expansion Compaction are required. One operation is responsible for constructing the filtering vector and the other for generating the grouping vector. The following operations use the previously generated vectors to filter and group the compacted data of the new edge frontier.

Contraction phases:

The first contraction phase operates on the “near” elements at each iteration of the algorithm. For this phase, only grouping is applicable, since the filtering done on the GPU is complete, and doing SCU filtering would result in no benefit. The grouping information is only used by the subsequent operation that processes “near” elements, which result in the new grouped node frontier.

```
void SSSP_Expand (node_frontier) {
    indexes,count = preparationGPU (node_frontier);

    // Addition
    filtering = accessExpansionCompactionSCU
        (edges, indexes, count, do_filter);
    grouping = accessExpansionCompactionSCU
        (edges, indexes, count, do_grouping);

    edge_frontier = accessExpansionCompactionSCU
        (edges, indexes, count, filtering, grouping);
    weight_frontier = accessExpansionCompactionSCU
        (weights, indexes, count, filtering, grouping);
    weight_frontier += replicationCompactionSCU
        (weights, count, filtering, grouping);
    return edge_frontier, weight_frontier;
}

void SSSP_Contract (edge_frontier, weight_frontier, threshold) {
    bitmask_near,bitmask_far =
    SSSP_contractionGPU (edge_frontier, weight_frontier, threshold);

    // Addition
    grouping = dataCompactionSCU (edge_frontier, bitmaskNear);

    node_frontier = dataCompactionSCU
        (edge_frontier, bitmask_near, grouping);
    farPileEdges = dataCompactionSCU
        (edge_frontier, bitmask_far, grouping);
    farPileWeights = dataCompactionSCU
        (weight_frontier, bitmask_far, grouping);
    return node_frontier;
}
```

Figure 4.10: Pseudo-code of the additional operations for a GPGPU SSSP program to use the enhanced SCU.

The second contraction phase operates on the “far” elements when there are no more “near” elements. For this phase both grouping and filtering are beneficial, since elements on the “far” pile are not filtered beforehand. Two additional Data Compaction operations are used to create the filtering and the grouping information for the “far” elements, which will be used by the subsequent operation that operates on “far” elements and generate the new filtered and grouped node frontier.

4.3.6 PageRank with the Enhanced SCU

Removing duplicated or already visited nodes is not an option for PR, since it considers all the nodes on every iteration of the algorithm. Furthermore, since the application accesses the

entire set of edges and nodes the memory access pattern is less irregular, hence, grouping the nodes provides little memory coalescing improvement. Therefore, the enhanced functionalities of the SCU are not used when running PR.

4.4 Experimental Results

In this section, we evaluate the performance improvement and energy reduction of the SCU. Figure 4.11 shows normalized energy consumption for BFS, SSSP and PR primitives on several graphs on our high-performance (GTX980) and low-power (TX1) GPU systems enhanced with the SCU, whereas Figure 4.12 shows the normalized execution time. The baseline configuration for both is the respective GPU system without our SCU.

4.4.1 Energy Evaluation

Figure 4.11 shows that the SCU provides consistent energy reduction, including both dynamic and static energy, across all graphs, all GPU systems and all graphs primitives. On average, the SCU provides a reduction in energy consumption of 6.55x for GTX980 and 3.24x for TX1 (an 84.7% and 69% of energy reduction respectively). The GTX980 is optimized for high performance at the expense of increased power dissipation, whereas the TX1 keeps energy consumption lower. For this reason, the energy savings are more significant on the GTX980, where we obtain reductions of 12.3x, 11x and 4.65x for BFS, SSSP and PR respectively. Nonetheless, the energy savings are also important for the TX1, achieving reductions of 5.35x, 4.54x and 1.5x for BFS, SSSP and PR respectively.

The energy savings obtained come from several sources. First, the SCU pipeline is specialized and tailored for stream compaction operations, thus being more efficient than the streaming multiprocessors' pipeline from an energy point of view. Second, the filtering operation reduces GPU workload, which further reduces dynamic energy consumption. Third, the grouping operation increases the degree of memory coalescing, which reduces the overall energy consumption of the memory hierarchy. Finally, the speedups reported in Figure 4.12 provide a reduction in static energy. All these factors combined allow the SCU to provide a large reduction in energy consumption.

4.4.2 Performance Evaluation

Figure 4.12 shows that the SCU provides performance improvements across all graphs and GPU systems for both BFS and SSSP graph primitives, and for PR only on the TX1. On average, we achieve speedups of 1.37x and 2.32x for the GTX980 and TX1 respectively. Performance improvement is more significant on the TX1, as it is already designed for maximum energy efficiency. We observe a speedup of 3.83x, 3.24x and 1.05x for BFS, SSSP and PR respectively. We also achieve significant speedups for the GTX980 of 1.41x and 1.65x for BFS and SSSP, although PR incurs a small slowdown. In PR all the nodes are considered active on every

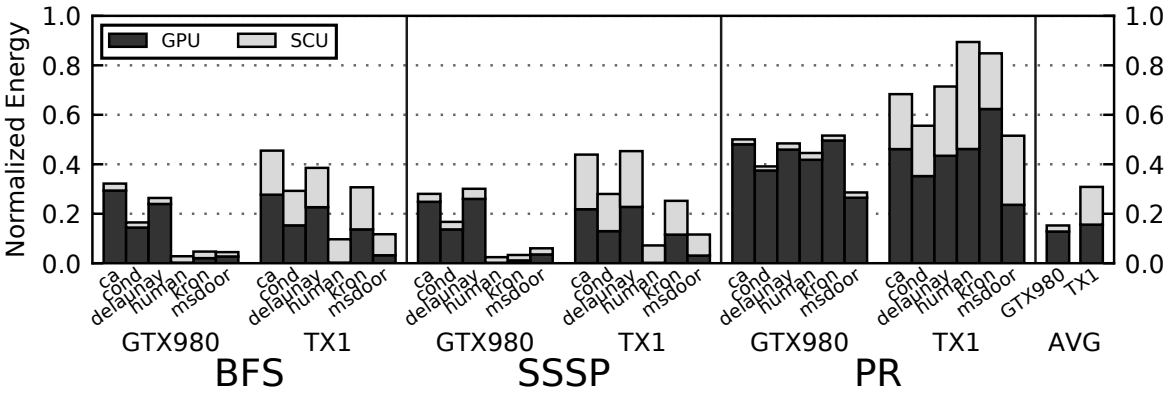


Figure 4.11: Normalized energy for BFS, SSSP and PR primitives on several datasets and in our two GPU systems using the proposed SCU. Baseline configuration is the corresponding GPU system (GTX980 or TX1) without the SCU. The figure also shows the split between GPU and SCU energy consumption.

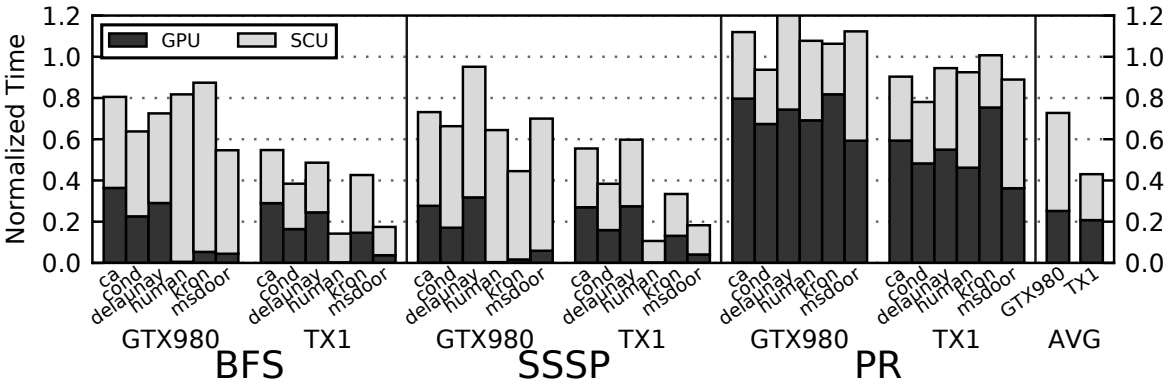


Figure 4.12: Normalized execution time for BFS, SSSP and PR primitives on several datasets and in our two GPU systems using the proposed SCU. Baseline configuration is the corresponding GPU system (GTX980 or TX1) without the SCU. The figure also shows the split between GPU and SCU execution time.

iteration of the algorithm, unlike BFS and SSSP. Therefore, memory accesses are less sparse and irregular, and the potential benefit of the SCU is lower.

Performance benefits come from three sources. First, the SCU performs compaction operations more efficiently than the GPU, as it includes a hardware pipeline specifically designed for this task. Second, the SCU is very effective at filtering duplicated and already visited nodes, thus largely reducing the workload. Third, the grouping operation increases the degree of memory coalescing, which reduces memory accesses and subsequently the pressure on the memory hierarchy.

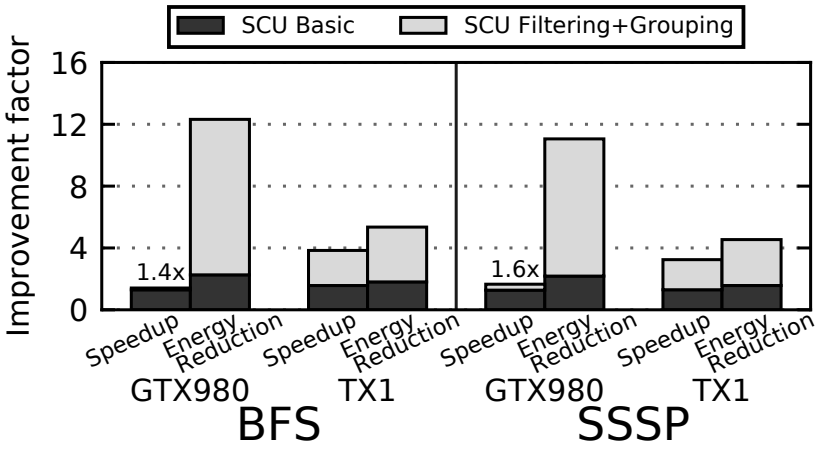


Figure 4.13: Speedup and Energy Reduction breakdown, showing separately the improvements due to the Basic SCU and the Enhanced SCU in both GTX980 and TX1 architectures. The Enhanced SCU achieves important energy reductions on the GTX980, whereas it delivers higher speedups on the TX1.

4.4.3 Enhanced SCU Results

In Figure 4.13 we analyze the performance and energy benefits that result from the basic SCU design presented in Section 4.2 and the additional benefits provided by the filtering and grouping operations described in Section 4.3 which are part of the enhanced SCU design. PR is not shown as it does not use the enhanced SCU capabilities.

The basic SCU design, which is restricted to offloading compaction operations of sparse data accesses from a large graph, provides around 2x energy reduction and 1.5x speedup for both BFS and SSSP on high-performance and low-power GPUs. The enhanced SCU design makes use of filtering and grouping operations that reduce GPU workload and improve memory coalescing, which leads to a reduction of memory hierarchy activity and a significant improvement over the basic SCU design. The enhanced SCU achieves large energy reductions for the GTX980 of 12.3x and 11x for BFS and SSSP respectively. For the TX1, although being already more energy-efficient, we obtain important energy reductions of 5.35x and 4.54x for BFS and SSSP. We also achieve significant performance improvements of 3.83x and 3.24x for BFS and SSSP for the TX1, whereas the improvement is lower in the GTX980 achieving a speedup of 1.4x and 1.6x for BFS and SSSP.

In BFS, the SCU is highly effective at filtering duplicated nodes, which reduces GPU workload by a large extent, improving performance and reducing GPU dynamic and static energy. For SSSP, filtering keeps only the elements with the best cost and grouping prepares the compacted data in a way that improves memory coalescing for the code executed on the streaming multiprocessors, which improves performance and reduces energy consumption on the memory hierarchy. Finally, in PR the stream compaction efforts are offloaded to the SCU without performing further processing, which already improves performance and provides energy reduction.

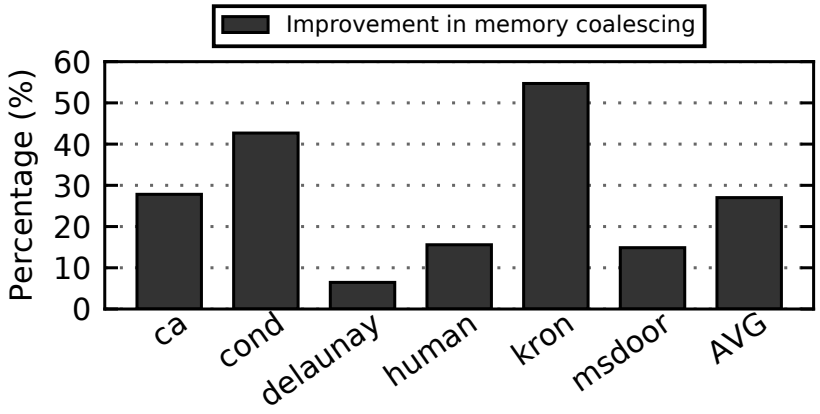


Figure 4.14: Improvement of the memory coalescing when using the grouping operation, for the SSSP algorithm on the TX1 GPU. The baseline configuration is SCU using only the filtering operations.

The filtering operation described in Section 4.3 removes the duplicated and visited nodes processed during the compaction with the goal to reduce the workload of the GPU by further reducing the size of the compacted array. On average, the filtering operation reduces the GPU instructions by 71% in BFS and 76% in SSSP for the TX1, with similar results for GTX980. Although the filtering operation increases the workload of the SCU, it reduces the workload of the GPU, resulting in important net savings in execution time and energy consumption for the overall system.

We have evaluated the efficiency of the grouping operation performed in the SCU to improve memory coalescing (see Section 4.3). The results for the TX1 are shown in Figure 4.14, where grouping improves the memory coalescing effectiveness in all the datasets, achieving an average improvement of 27%. When the required edges are being compacted, the SCU tries to write in consecutive memory locations edges whose destination nodes lie in the same cache line. By doing so, the subsequent processing of the compacted edges achieves higher GPU efficiency by reducing memory divergence. Furthermore, better memory coalescing also means that less memory requests are sent to the memory subsystem, reducing contention and overall memory traffic.

Finally, we analyze memory bandwidth usage as main memory is a constraining factor of graph applications. Graph-based algorithms expose a large amount of data parallelism, but they typically exhibit low data locality and irregular memory access patterns that prevent an effective saturation of memory bandwidth. Figure 4.15 shows how graph applications come short from saturating memory bandwidth. PR achieves higher memory bandwidth usage due to its higher regularity and data locality. Note that each GPU system has a different bandwidth and the figure indicates utilization of peak bandwidth. When comparing the GPU and SCU memory bandwidth utilization of a particular algorithm two factors come into play: performance speedup and memory accesses reduction. The high-end GTX980 system with the SCU exhibits a lower bandwidth utilization than the GPU system due to achieving higher reductions in memory

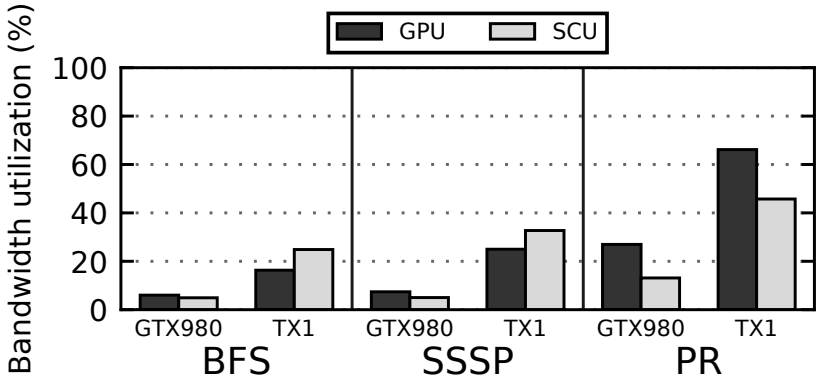


Figure 4.15: Memory bandwidth utilization for the graph applications running on a Baseline GPU system and on a GPU system incorporating the SCU. Note that each GPU system has a different bandwidth and the figure indicates utilization of peak bandwidth.

accesses than in performance, whereas we see the opposite for the low power TX1 system. The TX1 system with the SCU obtains higher speedups than memory accesses reduction and, as a consequence, we see an increase in memory bandwidth utilization in BFS and SSSP.

4.4.4 Area Evaluation

Our results show that the SCU requires a small area of 13.27 mm^2 for the GTX980 system and 3.65 mm^2 for the TX1, including the hardware required for filtering and grouping operations. Considering the overall GPU system, the SCU represents 3.3% and 4.1% of the total area for the GTX980 and the TX1 respectively. The SCU is tightly integrated in the GPU and has access to the L2 cache which is used to store the hash table used for filtering and grouping operations, so it does not require any additional storage.

4.5 Conclusions

In this chapter we have presented our proposal to extend the GPU with a Stream Compaction Unit (SCU) to improve performance and energy-efficiency for graph processing. The SCU is tailored to the requirements of the stream compaction operation, that is fundamental for parallel graph processing as it represents up to 55% of the execution time on average. The rest of the graph processing algorithm is executed on the streaming multiprocessors achieving high GPU efficiency, since it works on the SCU-preprocessed compacted data.

The SCU not only compacts but also filters out duplicated and already visited nodes during the compaction process, reducing the number of GPU instructions by more than 70% on average. In addition, it implements a grouping operation that writes together in the compacted array edges whose destination nodes are in the same cache line, improving memory coalescing by 27% for the remaining GPU workload. The resulting high-performance and low-power GPU designs

including our SCU unit achieve significant average speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively for several graph-based applications, with a small 3.3% and 4.1% increase in overall area respectively.

5

Improving Graph Processing Divergence-Induced Memory Contention

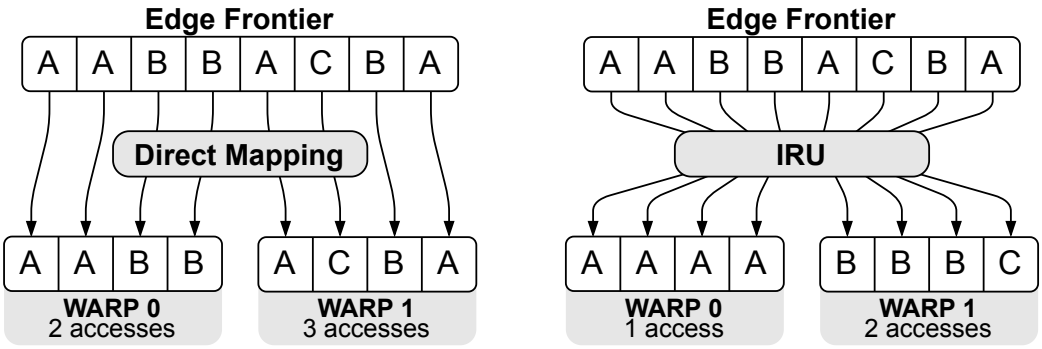
This chapter provides the details of the Irregular accesses Reorder Unit (IRU) introduced in Section 1.4.2. The chapter is organized as follows. First, Section 5.1 reviews and expands the motivation of our IRU proposal. Section 5.2 introduces the IRU architecture, its inner working and integration into the GPU. Afterwards, Section 5.3 describes the API and its main functionalities. Section 5.4 explores the improvements in performance and energy consumption achieved by our proposal. Finally, Section 5.5 sums up the main conclusions of this work.

5.1 Introduction

GPGPU architectures have become established as the dominant parallelization and performance platform achieving exceptional popularization and empowering domains such as linear algebra, Big Data analytics, machine learning, image detection and self-driving cars. As introduced in Section 1.2, GPGPU architectures excel at processing highly-parallelizable throughput oriented applications, which exhibit regular execution and memory access patterns. However, irregular applications struggle to fully exploit GPGPU performance and efficiency as a result of control flow divergence and memory divergence, due to irregular memory access patterns resulting from processing unstructured data.

As explored in Section 2.2, in order to ameliorate these issues, programmers are obligated to carefully consider architecture features and devote significant efforts to modify the algorithms with complex optimization techniques, which shift programmers priorities yet struggle to quell the shortcomings and harm code development and portability.

GPGPU programming models such as CUDA employ threads to exploit parallelism, each



(a) Baseline GPU

(b) GPU with IRU

Figure 5.1: Memory Coalescing improvement achieved by employing the IRU (5.1b) to reorder data elements that generate irregular accesses versus a Baseline GPU (5.1a) execution.

processing its own set of data while synchronizing with the rest to perform complex behaviors determined by the algorithm. The number of threads and the ability to coalesce the memory accesses within a warp are some of the key factors that determine the utilization of the GPU resources as explored in Section 1.2. The simplest way to exploit GPGPU parallelism is to instantiate as many threads as data elements to process and directly assign each element to a given thread, which shows effective for regular application yet causes degradation in utilization and memory coalescing for irregular applications. As explored in Section 2.2, more complex thread to data assignment or additional preprocessing (e.g. reordering) can be performed by the programmer, but at the cost of additional algorithm complexity and computational cost.

We show that in graph-based GPGPU irregular applications these inefficiencies prevail as GPGPU programming models impose restrictions that hinder full resource utilization and since GPGPU architectures are not designed to efficiently support sparse irregular programs. Nonetheless, we find that it is possible to relax the strict relationship between thread and data processed to empower new optimizations as introduced in Section 1.4.2.

Based on this key idea, we propose the Irregular accesses Reorder Unit (IRU), a novel hardware extension tightly integrated in the GPGPU pipeline. The IRU reorders data processed by the threads on irregular accesses which significantly improves memory coalescing, and allows increased performance and energy efficiency. Relaxing the thread-data strict relationship enables the IRU reordering of data serviced to the threads, i.e. deciding at run-time the mapping between threads and data elements leading to greatly improved memory coalescing. Figure 5.1 shows conceptually how the IRU is used to assign data to the threads and achieves an improvement on memory coalescing against the baseline GPU mapping. Programmers can easily utilize the IRU with a simple API, or use compiler optimized code using extended ISA instructions. Additionally, the IRU is capable of filtering and merging duplicated irregular access which further improves graph-based irregular applications by reducing useless resource utilization of the GPU.

Finally, we evaluate our proposal for state-of-the-art graph-based algorithms and a wide selection of applications. Results show that the IRU achieves a memory coalescing improvement of 1.32x and a 46% reduction in the overall traffic in the memory hierarchy, which results in

1.33x and 13% improvement in performance and energy savings respectively, while incurring in a small 5.6% area overhead.

This chapter focuses on improving the performance of irregular applications, such as graph processing, on GPGPU architectures. The main contributions explored are the following:

- We characterize the degree of memory coalescing and GPU utilization of modern graph-based applications. Our analysis shows that memory coalescing can be as high as 4 accesses per warp and GPU utilization as low as 13.5%.
- We propose the IRU, a novel hardware unit integrated in the GPGPU architecture which enables improved performance of sparse and irregular accesses by reordering data serviced to each thread. We further extend the IRU to filter repeated elements in graph-based applications, largely reducing GPU redundant workload.
- We propose an ISA extension and high-level API and show how modern graph-based applications can easily leverage the IRU hardware.
- Overall the GPU architecture with our IRU improves memory coalescing by a factor of 1.32x and reduces the overall memory hierarchy traffic by 46%, resulting in 1.33x and 13% speedup and energy savings respectively for a diverse set of graph-based applications. The IRU represents a small area overhead of 5.6%.

5.2 Irregular accesses Reorder Unit

In this section, we introduce the Irregular accesses Reorder Unit (IRU), which improves performance of irregular workloads such as graph applications on GPGPU architectures. High GPGPU performance is achieved with regular execution and predictable memory access patterns. As outlined previously in Section 1.2, irregular graph applications do not meet these characteristics which significantly lowers their GPU performance and efficiency. Irregular application achieve poor memory coalescing which puts higher pressure on the resources of all components of the memory hierarchy. GPGPU resources utilization remains low even-though many non-trivial techniques are used to reduce overheads, in turn hindering application development and performance.

We propose to extend the GPGPU with the IRU to reduce the overheads caused by irregular accesses. The IRU is a compact and efficient hardware unit integrated into the Memory Partition (MP) of the GPU architecture as seen in Figure 5.3a, which incurs in very small energy and area overheads. The IRU leverages the observation that GPU programs employ threads as a mean to convey parallelism; they are in many occasions independent of the data that they process. The main goal of the IRU is to process the indices used to perform irregular accesses, reorder and redistribute them. The reordering aggregates indices that access the same memory block and services them to a requesting warp, improving the collocation of irregular accesses and thus increasing memory coalescing. In turn, the improved memory coalescing reduces congestion of the resources of the LD/ST unit, L1, interconnection, L2 and main memory. In addition, the reordering is performed across all the indices accessed by all the SMs, and so, collocating

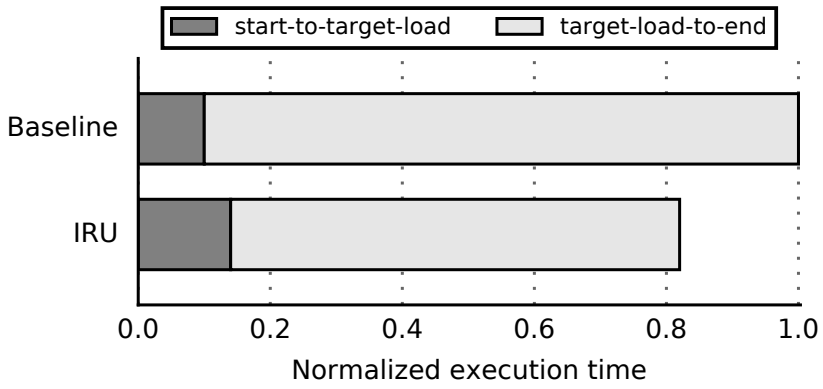


Figure 5.2: Warp average normalized execution with and without IRU. The dark bar indicates execution time until the target load is serviced, and the light bar from service to finalization. The IRU achieves speedups despite the overhead introduced.

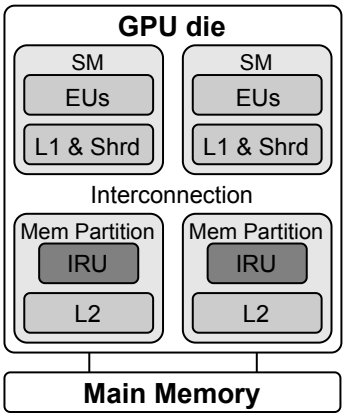
irregular accesses potentially gathers data obtained by irregular accesses in a single or fewer SMs, thus further reducing interconnection traffic and L1 data thrashing. Figure 5.2 shows the average normalized execution of a warp of a baseline GPU against one with the IRU. The dark bar indicates the execution time until the load processed and reordered by the IRU is serviced, while the light bar shows the normalized time until finalization. The overhead incurred by the IRU servicing the load is more than offset by the additional performance gained from the reduction of the overheads due to improved memory coalescing of the targeted irregular access.

The IRU processes the indices of a target irregular instruction, with the objective to optimize its coalescing. Additionally, the elements processed contain more data than just the indices, as mandated by the API described in Section 5.3. While these data are not used for the IRU coalescing logic, since the indices remain the information that the IRU utilizes to improve memory coalescing, it is responsible to fetch, generate and reply to the SM the additional data.

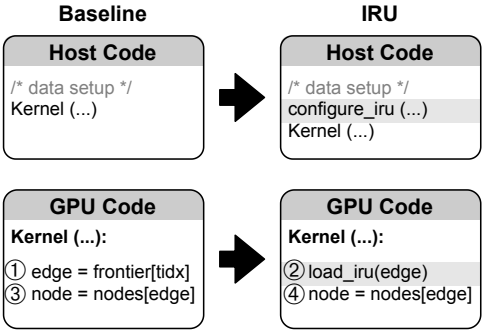
5.2.1 GPU Integration

The IRU integration into the GPU is covered in Figure 5.3, showing architectural 5.3a, programming 5.3b and execution 5.3c-5.3e integration. The execution shows how the Baseline and the IRU modified GPU programs in Figure 5.3b operate with the two warps and data from Figure 5.1.

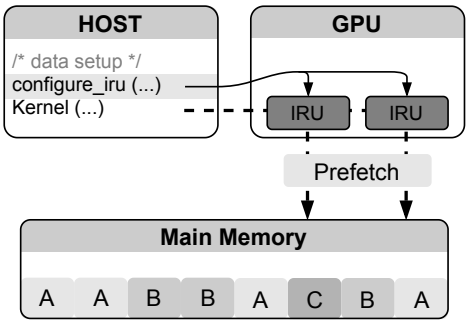
The Baseline program performs a regular access ① to gather indices that are then used for an irregular access. The IRU modified code performs the same operation but using the IRU hardware with the *load_iru* operation ②, which is a simple modification explored in Section 5.3. The baseline code is executed by the GPU as follows. First, the two warps retrieve the indices performing regular accesses to the L1, as seen in Figure 5.3d. Afterwards, Figure 5.3e shows how they perform irregular accesses to the L1 with the retrieved indices which. Due to the high divergence, they result in many L1 accesses ③.



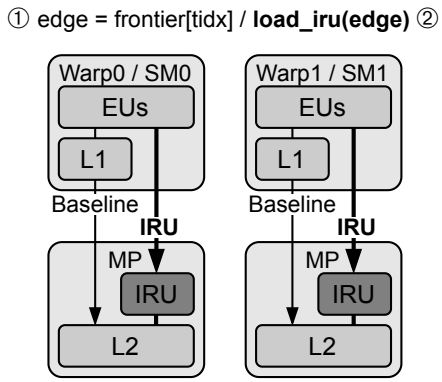
(a) GPU architecture with IRU.



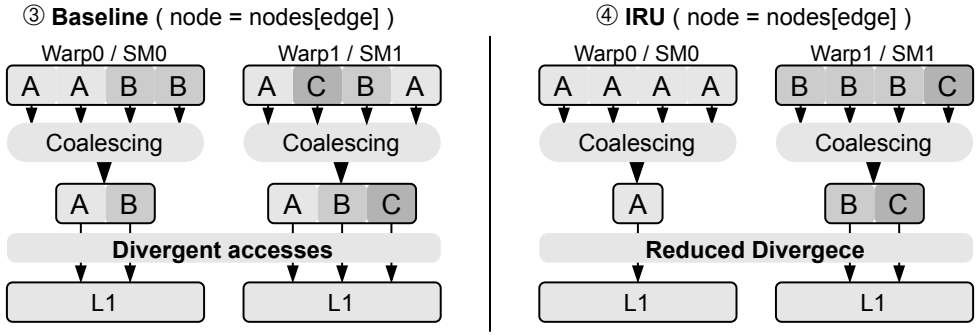
(b) Irregular access code modifications.



(c) IRU Configuration and Initialization.



(d) Irregular accesses indices retrieval.



(e) Divergence improvement of irregular accesses executed with the IRU reordered indices compared to the Baseline.

Figure 5.3: IRU integration with the GPU at different levels: architectural (a), program model (b) and execution (c,d,e). The execution showcases how the program (b) works on the Baseline and the IRU, operating with the two warps and data from Figure 5.1.

In contrast, the IRU program first introduces a configuration step performed on the host, shown in Figure 5.3c, that provides data of the irregular accesses to optimize. The configuration required for this program consists of the base address and data type of the irregular accessed

data, and the indices array and total number of irregular access. Further IRU capabilities are enabled and used with optional parameters on overloaded functions, reviewed in Section 5.3. Afterwards, when the kernel execution starts, the IRU triggers the prefetching of the indices from L2 and memory, which are then autonomously reordered in the IRU hash. The IRU activity is overlapped with the execution of the kernel, and disabled when all the data is processed or if not used for the kernel in execution.

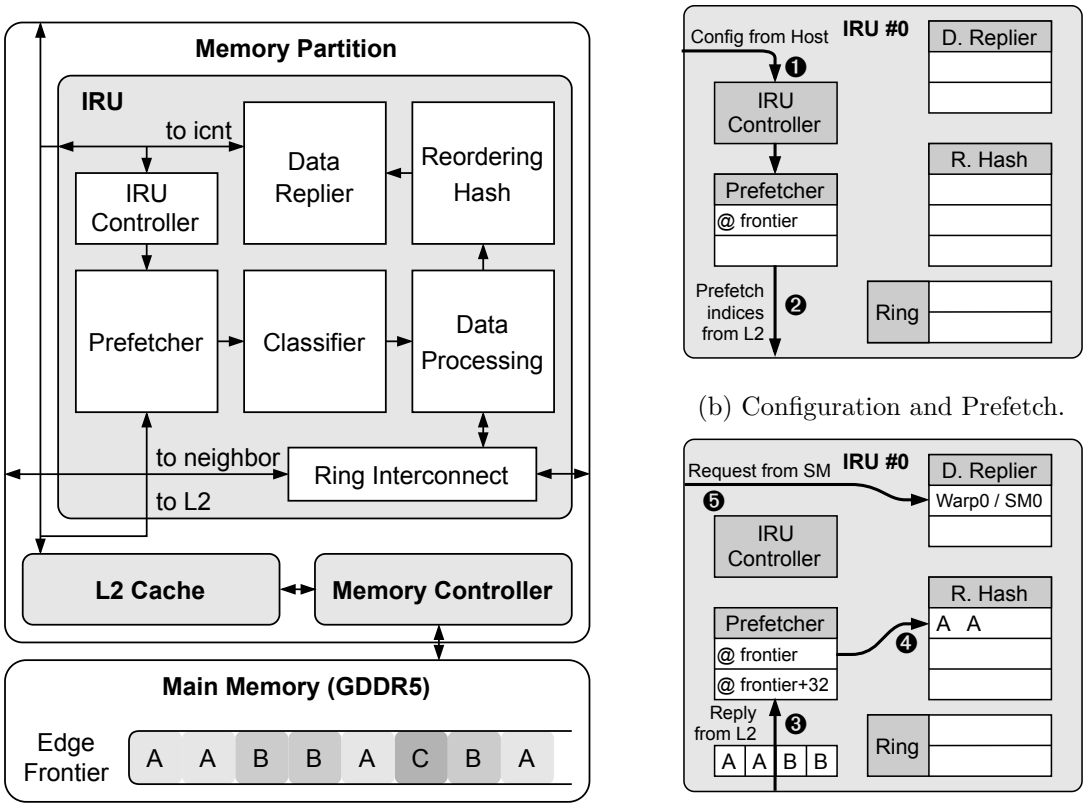
Regular execution proceeds until encountering the *load_iru* operation, at which point the warps retrieve the indices performing requests directly to the IRU bypassing the L1, as seen in Figure 5.3d. The IRU replies with reordered indices either instantly, if they are ready, or otherwise after a timeout to avoid starvation. Finally, the warps perform the irregular access that was the target of the optimization ④. This access is performed with the IRU reordered indices which achieve reduced divergence performing less accesses than the baseline program, as depicted in Figure 5.3e.

5.2.2 Hardware Overview and Processing

The internal pipelined hardware of the IRU is shown in Figure 5.4a. It is composed of a number of blocks each with specific purpose, simple logic and buffering of data. The main purpose of the IRU, which is to reorder indices to improve memory coalescing, is accomplished with the use of a hash located inside the *Reordering Hash* block. Instead of multiple private hashes, there is a single logical hash partitioned among the IRUs. This motivates the inclusion of a ring interconnection between the IRUs to forward the data to the corresponding partition of the logical hash. We have observed that the degree of memory coalescing is significantly affected if each IRU hash is private and separated; which would constrain IRUs reordering scope to data from a single memory partition. Finally, requests are issued to the L2 to exploit data locality among kernel executions. Alternatively, requests can be configured to bypass L2, which could become beneficial for streaming kernels that do not reuse the data.

As seen in Figure 5.4a, the IRU's different blocks functionality is as follows:

- **IRU Controller:** Control logic that holds the configuration and general signals for other blocks.
- **Prefetcher:** Configured at kernel launch, it is the block responsible to fetch the data to be reordered.
- **Classifier:** Fetches the data retrieved by the Prefetcher, stores it in different buffers based on whether it has to be locally processed or sent to the ring interconnection.
- **Data Processing:** Retrieves data from the internal buffers or the ring and outputs it to the hash, or to the ring.
- **Reordering Hash:** Contains a partition of the hash. The data is inserted so that data placed on the same hash entry result in indices that point to the same memory block, increasing memory coalescing.



(a) Overview of the IRU internal pipeline.

(b) Configuration and Prefetch.

(c) Data and Request retrieval.

(d) Ring interconnection.

(e) Requests reply.

Figure 5.4: Architecture and the internal processing performed by the IRU. The indices in memory (from Figure 5.1) are processed by two IRU partitions (IRU 0 shown), which is later replied to a request coming from Warp 0 in SM 0.

- **Data Replier:** Prepares reordered data that achieve improved memory coalescing and forwards them to the requesting load from the SM.

- **Ring Interconnect:** Forwards data from a given IRU in a memory partition to another.

The overall internal processing of the IRU is described in Figure 5.4. The figure covers a general overview of the internal IRU architecture and the detailed step by step working of the most relevant components of the IRU covering: configuration and prefetching (5.4b), data and requests retrieval (5.4c), ring interconnection interaction (5.4d) and requests reply (5.4e).

Prefetching and Data Processing

The *IRU Controller* is first initialized from the Host by executing the *configure_iru* function with the corresponding data ❶. The *Prefetcher* later uses this information to determine the addresses to prefetch when the GPU kernel starts executing ❷, then it begins issuing a limited number of on-the-fly prefetches to avoid saturating memory bandwidth and degrading performance. Each IRU only prefetches information from its corresponding memory partition. In Figure 5.4, the first four elements from main memory are fetched by IRU 0, while the next four by IRU 1. When a reply comes back, the retrieved data is stored in a FIFO queue to be later processed.

Afterwards, the *Classifier* block processes the prefetched data ❸ by splitting it into several smaller FIFO queues, an element per cycle per queue. The smaller FIFO queues contain the elements that will be inserted in the hash or forwarded through the interconnection. A hashing function of the element is used to determine which hash entry it is mapped to and, therefore, if it will access a local bank or must be sent through the interconnection. Finally, the *Data Processing* block retrieves elements from both the smaller FIFO queues and the ring, prioritizing the latter, and forwards it to the ring or inserts it to the local hash ❹. On Figure 5.4e, the elements labeled A are inserted into the local hash to the same entry, as they are determined to target the same memory block.

Meanwhile, requests from the SMs can be received at any time which are then processed by the *Data Replier* ❺. This request originates directly from the SM (i.e. bypassing the L1) and are generated by the extended ISA *load_iru* operations, that are responsible to retrieve the IRU processed data. Their information is stored until enough data is available to satisfy the request or until a timeout is reached.

Ring and Data Reply

Due to the partition of the reordering hash, the hash function of the elements fetched from a memory partition can require that element to be inserted in another IRU partition. The *Ring Interconnection* allows to receive and send elements to the neighbor partitions at every cycle. In Figure 5.4d, the elements labeled B are determined to correspond to another IRU partition and so are inserted in the ring ❻. Meanwhile, data from the neighbor partition is received (indices A and C are determined to correspond to IRU 0) ❼.

Lastly, the elements corresponding to this IRU partition are gathered from the ring and inserted into the reordering hash. When the *Data Replier* detects a hash entry that is complete, or enough data is available to reply a request, the oldest request is replied back to the SMs with that entry reordered elements ❽, and the data is evicted from the hash. The data used for the

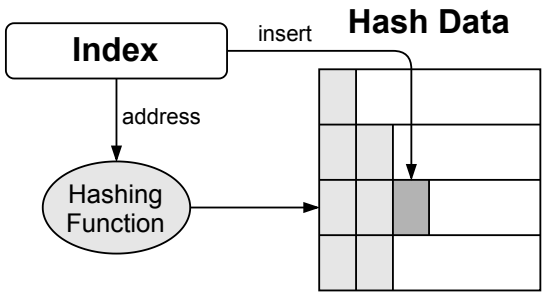


Figure 5.5: Hash insertion diagram showcasing how an element is used for the hashing function and how it is stored in the hash data of the *Reordering Hash*.

reply (four A) are the indices used for the irregular access being optimized, but additionally more data might be processed per element, in which case multiple replies would be issued, at most two additional replies.

Additionally, a timeout is employed to avoid excessively delaying a request. Once the timeout is reached, it then fetches data from the hash with the best coalesced data entry present, and replies once enough data is retrieved, effectively trading-off coalescing for lower latency. Furthermore, simple control logic is added to the SM and IRU partitions to handle balancing issues (i.e between request and entries ready), each SM distributes the requests evenly across the different IRU in the memory partitions, and requests can be replied by IRU partitions other than the original. Finally, when no more data is left to be inserted into the IRU, the *Data Replier* replies to the SM by merging the remaining hash entries which might not be full. These merging tries to avoid splitting a hash entry between two replies, which would consequently impact memory coalescing.

5.2.3 Reordering Hash

The *Reordering Hash* drives the IRU, it contains a physical partition of the global logical hash, which is direct mapped and multi-banked. Each entry holds up to 32 elements that are inserted into the entry in subsequent locations at every hash insertion as depicted in Figure 5.5. Furthermore, the hash function key that points to an entry is generated from the value being inserted into the hash entry. The computation of the hash function collocates in a single hash entry the elements that will generate memory fetches that target the same memory block, which provides the memory coalescing improvement achieved with the IRU.

Unlike a regular hash, an insertion allows to append elements into a hash entry even if the tag does not match. The inherent drawback of this decision is that the elements that a hash entry collocates might actually not access the same memory block, and thus the memory coalescing that it can achieve will not be optimal. Nonetheless, this design decision largely reduces hardware complexity and avoids handling the conflicting elements gathered and replied back to the SM without achieving any improved coalescing. Furthermore, a good dispersion hash function and properly sized hash tables limits the amount of conflicts and ameliorates the negative impact on memory coalescing. Ultimately, when an entry is completely filled with

32 elements, no more data can be inserted to it. At this point, it has 32 collocated elements that potentially will access the same memory block when the program uses them to perform an irregular access, unless there were conflicts. Note that some of these conflicting elements might collocate among themselves, thus not impairing memory coalescing so severely.

Some API operations described in Section 5.3 require additional comparators or adders to be used in a hash insertion. The additional data that the elements might have is processed by this hardware, which effectively merges or filters an element present in the hash with the one being inserted. Since this operations will filter out elements, some threads that requested data will not receive any, which is handled by the *Data Replier* and exposed to the programmer with the API.

5.3 IRU Programmability

Ease of programability is an important aspect when it comes to writing efficient parallel programs, reason for which toolkits such as CUDA are very successful. Efficient irregular programs require complex optimization techniques. The IRU has been designed to be easily integrated and programmable. The IRU extends the GPGPU ISA to support memory load operations that fetch data from the IRU, which require small changes to the pipeline to decode these instructions and changes to the LD/ST unit to route these requests to the IRU. To avoid directly using ISA instructions we provide a simple API easily integrated in CUDA kernels. Furthermore, since the changes to the code are minimal, a compiler that supports the ISA extensions can generate the instrumented code, freeing the programmer from performing the optimization effort and delivering a more efficient GPGPU architecture for irregular applications.

IRU's main optimization is the reordering of indices fetched from memory that are used for irregular accesses. This optimization is based on the premise that the data assigned to the threads is independent of what thread is processing it. Consequently, to be able to correctly utilize the IRU for this optimization, the programmer has to guarantee that the reordering can be applied correctly as other data or accesses might have to be done with the new order achieved. The API provides additional functionality to facilitate this guarantee.

The baseline functionality provided by the API and IRU hardware supports reorder of an array of 24-bit indices. Additionally, a secondary 32-bit array can be processed simultaneously, yet the reordering is based on the indices array as to improve the coalescing achieved when performing an irregular access. The data (i.e. index and entry in the secondary array) provided to the threads is reordered applying the same reordering to both indices and secondary array, maintaining the original pair of index and secondary data. Figure 5.6 shows how the input data, first two rows, is reordered in the output data, last two rows. The reordering is based on the array of indices, the edge frontier, and every edge is kept with its corresponding weight. This secondary array can be used to process attributes or extra data of the elements being processed. It might be the case that more than a single additional array has to be processed. In this case, the reordering operation can return in which position in the original array the reordered element was located. This is indicated in Figure 5.6 by the sub-index of the edge frontier, showing the position in the original array. This position value can be used to fetch any additional attributes required.

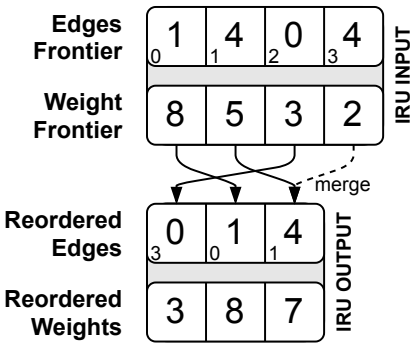


Figure 5.6: IRU processing of two arrays with filtering enabled. The "edges" is the indexing array, while the "weight" is the secondary array. The filtering operation is an addition.

Graph-based algorithms process several nodes and edges simultaneously. For this reason, it is common that several edges lead to the same destination node which causes redundant work. This additional work is usually benign as the program implements filtering techniques, which are effective yet computationally costly due to synchronization requirements. To aid the program with this additional workload, the IRU is extended to provide filtering or merging of elements (i.e. pair of index and attribute). The IRU can easily detect duplicated indices that are processed simultaneously and so it can remove them or might perform some operation to merge both elements. The operations supported by the IRU are integer comparison and floating point addition. Figure 5.6 shows the merging of two indices into one on the output data by adding their attributes in the secondary array. Filtering out elements causes some threads to not receive data, and so we extend the API to indicate if a given thread data has been filtered out. IRU groups the disabled threads in warps rather than preparing replies to warps with reordered data and disabled threads, this approach allows to minimize branch divergence, remove redundant work and improve IPC.

The API seen in Figure 5.7 provides two main functions: $configure\_iru$, used from the host to configure the IRU, and $load\_iru$, used inside the CUDA kernel to retrieve reordered data from the IRU. At the start of kernel execution, the configure function is called to provide all the parameters of the data that will be processed. The required parameters are: target array base address and data type width, both parameters used to configure the offset to be applied to the indices as to compute the coalescing required. The indices array is required too, which is the main data reordered. Finally, the number of elements in the indices array. Optional parameters include the additional secondary array, reordered together with the indices array, and the optional filtering operation performed.

5.3.1 IRU enabled Graph Applications

All the previously described functionalities enable the instrumentation of state-of-the-art implementation of graph-based algorithms such as BFS, SSSP and PR. Although we use push graph implementations, the IRU is not specifically targeting push or pull. The ease of use of our API allows very simple instrumentation an minimal code changes while providing efficient

```
1  void configure_iru (
2      addr_t target_array,
3      size_t target_array_data_type_size,
4      addr_t indices_array,
5      addr_t secondary_array,
6      size_t number_elements,
7      filter_op_t filter_op );
8
9  __device__ bool load_iru (
10     addr_t &indices_array,
11     addr_t &secondary_array,
12     uint32_t &position );
```

Figure 5.7: API additional functions. Multiple definitions used due to optional parameters.

memory coalescing improvements. The following examples show how the *load_iru* can be used from within GPGPU kernels easily replacing existing code.

The basic functionality of the IRU is a good fit for the BFS algorithm as illustrated in Figure 5.8. The indices found in the *edge_frontier* array are used to access the *label* array, resulting in irregular memory accesses and poor memory coalescing. The programmer can easily replace the previous instruction with the *load_iru* operation to obtain the indices in such a way that memory coalescing is improved and thus overall performance improves.

```
1  __global__ void BFS_Contract (...) {
2    int pos = blockDim.x * blockIdx.x +
3              threadIdx.x;
4    if (pos < number_elements) {
5      int edge;
6
7  #ifdef NOT_INSTRUMENTED
8      edge = edge_frontier[pos];
9  #elif USE_IRU
10     load_iru(edge);
11 #endif
12
13     // more computation ...
14     label[edge] = distance;
15   }
16 }
```

Figure 5.8: Simple instrumentation of the BFS algorithm Kernel using the API of the IRU.

The SSSP algorithm processes additional data per element, since each edge has an associated weight value. Figure 5.9 shows how *load_iru* can handle the use of an additional array, while also retrieving the original position of the reordered element in the *pos* variable. Note that the algorithm requires the *pos* variable to be correctly updated with the reordered element in line 17, which is easily accomplished with our API extension.

```
__global__ void SSSP_Compaction (...) {
  int pos = blockDim.x * blockIdx.x +
            threadIdx.x;
  if (pos < number_elements) {
    int edge, weight;

#ifdef NOT_INSTRUMENTED
    edge = edge_frontier[pos];
    weight = weight_frontier[pos];
#elif USE_IRU
    load_iru(edge, weight, pos);
#endif

    int previous =
    atomicMin(&label[edge], weight);
    if (previous > weight)
      lookup[edge] = pos;
  }
}
```

Figure 5.9: Simple instrumentation of the SSSP algorithm Kernel using the API of the IRU. The *load_iru* operation is using all the parameters. The variables *edge* and *weight* are reordered together while *pos* retrieves their original position in the array which is later used.

```
__global__ void PR_Contract (...) {
  int pos = blockDim.x * blockIdx.x +
            threadIdx.x;
  if (pos < number_elements) {
    int edge;
    float weight;
    bool active_thread = true;

#ifdef NOT_INSTRUMENTED
    edge = edge_frontier[pos];
    weight = weight_frontier[pos];
#elif USE_IRU
    active_thread = load_iru(edge, weight);
#endif

    if (active_thread)
      atomicAdd(&label[edge], weight);
  }
}
```

Figure 5.10: Simple instrumentation of the PR algorithm Kernel using the API of the IRU. The *load_iru* operation is used with all parameters, and additionally, filtering deactivates threads with is noted by the *active_thread* variable.

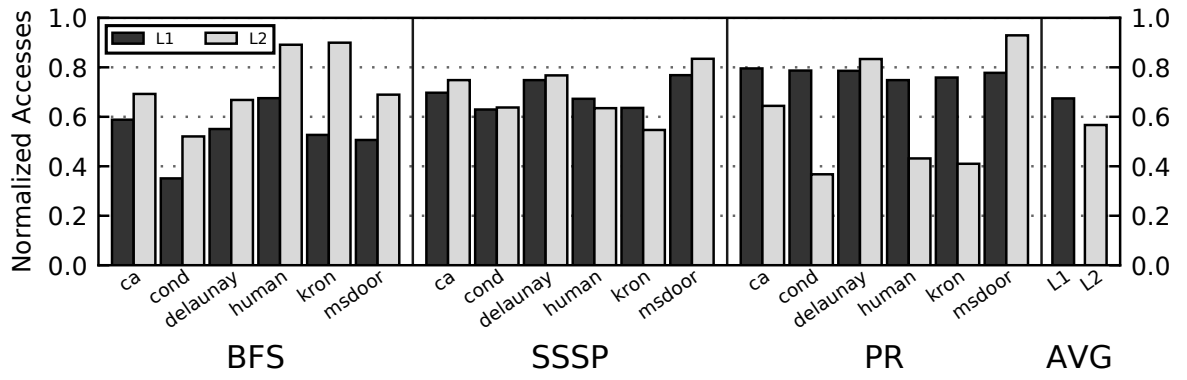


Figure 5.11: Normalized accesses to L1 and L2 caches of the IRU enabled GPU system against the Baseline GPU system. Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset.

Finally, the PageRank kernel shown in Figure 5.10 performs additions of the elements' weights into the *label* array. Utilizing the filtering/merge functionality of the IRU, an initial addition can be performed while the elements are being processed in the IRU, which allows to disable merged out threads. The *load_iru* function returns whether or not the thread has a valid element or it has been merged out; the value in a retrieved element's *weight* has the sum of those *weight* of the same *edge*. Note that the filtering is not complete as it merges only elements found concurrently on the IRU, yet it manages to filter a significant amount of duplicated elements. Overall, this extension allows reducing the workload of the kernel by removing a large number of *atomicAdd* operations.

5.4 Experimental Results

In this section, we analyze how the memory hierarchy contention is reduced, the reduction of interconnection traffic, the improvement on memory coalescing, the IRU filtering capabilities, and the overall performance and energy improvement of our proposed GPU system with the IRU with respect to the baseline GPU. Our workloads are the graph algorithms BFS, SSSP and PR, that are run for a set of diverse graphs shown in Table 3.5.

5.4.1 Memory Pressure Reduction

IRU's main functionality is to reorder irregular accesses improving their memory coalescing and, as a consequence, reducing the overall contention in the memory hierarchy. Figure 5.11 shows how the IRU consistently reduces accesses and contention on both L1 and L2 across all graph algorithms and datasets. Accesses to L1 and L2 are reduced to as low as 35% and 36% for the *cond* benchmark on BFS and PR respectively, and overall accesses are reduced to 67% and 56% of the original L1 and L2 accesses on average.

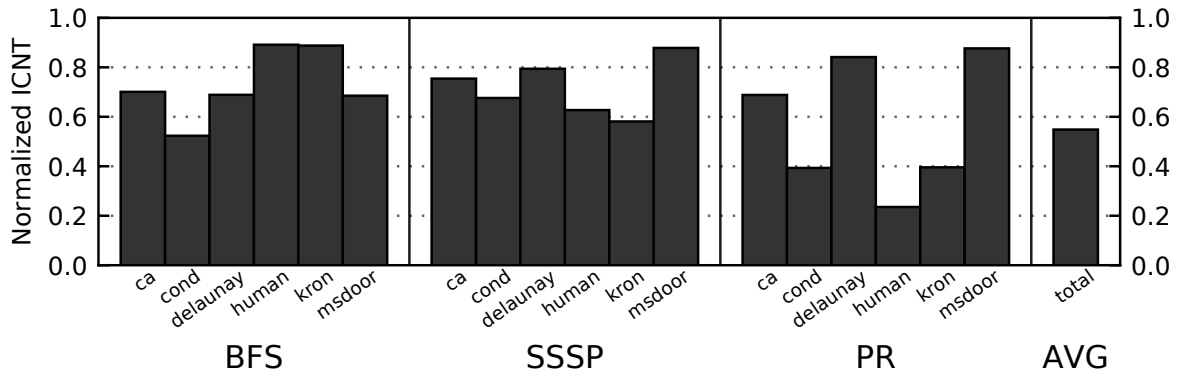


Figure 5.12: Normalized interconnection traffic between SM and MP of the IRU enabled GPU system against the Baseline GPU system. Significant reductions are achieved across BFS, SSSP and PR graph algorithms and every dataset.

This important reduction comes from several factors. First, the IRU reordering of irregular accesses improves coalescing which reduces the accesses to L1. Second, IRU reorders requests across SMs so it helps to collocate accesses of a particular memory block to a single SM, avoiding data replication across L1 data caches, resulting in improved hit ratios. Third, overall reduced accesses to L1 reduce capacity and conflict misses improving data thrashing and consequently reducing L2 accesses. Finally, IRU filtering further reduces accesses by removing or merging duplicated elements already processed, reducing additional accesses performed in the baseline.

L2 accesses reduction is greater than in L1 in some benchmarks for SSSP and PR graph algorithms. A significant amount of the indices reordered by the IRU on SSSP and PR are used for irregular accesses performed by atomic instructions. In GPGPU-Sim atomic operations bypass the L1 and are handled at the L2 on the corresponding memory partition. IRU coalescing and filtering improvement for these operations does not reduce L1 accesses but L2 accesses, explaining the larger reduction in L2 accesses compared to L1 for SSSP and PR. Note that atomic operations within a warp are coalesced as long as different threads in it access different parts of the cache line.

We have also analyzed the impact of the IRU in the Network-on-Chip (NoC) that interconnects the Streaming Multiprocessors (SM) with the Memory Partitions (MP). Figure 5.12 shows the normalized traffic in the NoC. As it can be seen, the IRU consistently reduces interconnection traffic across all graph algorithms and datasets. Traffic between SM and MP is reduced to as low as 23% for the *human* benchmark on PR, overall achieving a reduction to 54% of the original interconnection traffic. This reduction is due to several factors. First, the improved memory coalescing results in a more efficient use of the L1 data cache, significantly reducing the number of misses. Second, filtering also contributes to lower L2 accesses which reduces interconnection contention. Finally, the extended ISA instructions used on a *load_iru* operation allow reduced traffic by issuing a single request to the IRU that receives two replies (up to three replies), whereas the baseline GPU would have issued two requests and two replies in order to gather data in different frontiers.

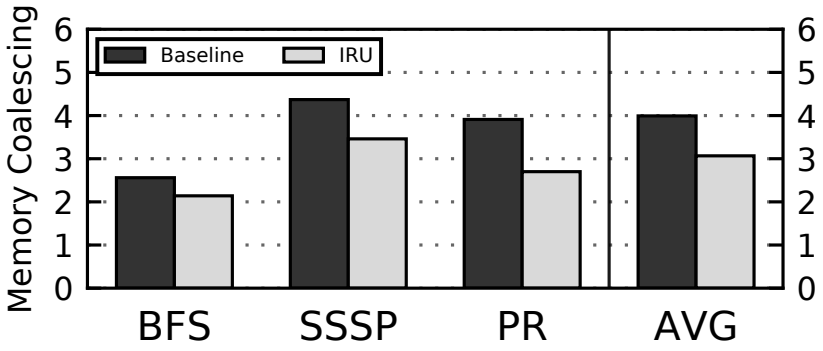


Figure 5.13: Improvement in memory coalescing achieved with the IRU over the Baseline GPU system. Vertical axis shows the number of memory requests sent to the L1 cache on average per each memory instruction, i.e. how many memory requests are required to serve the 32 threads in a warp.

Figure 5.13 shows the improvement in memory coalescing delivered by the IRU. A higher coalescing number indicates that more accesses are needed to serve each warp memory request, with a maximum of 32 accesses per request, and a minimum of 1 access in the best scenario. The IRU improves the overall memory coalescing for every graph algorithm from 4 to 3 accesses per memory requests on average, requiring one fewer access per request. Note that the filtering schemes that some of the algorithms employ, combined with the filtering applied by the IRU, reduce the potential of coalescing, since filtering removes duplicated elements whose access could be coalesced. However, overall memory coalescing is significantly improved, reducing the pressure on the memory hierarchy.

Finally, main memory accesses are reduced by 4% due to reduced L2 misses. Overall, reordering and filtering techniques allow the IRU to deliver significant improvements in memory coalescing and reduce contention in every level of the memory hierarchy.

5.4.2 Filtering Effectiveness

The IRU hardware provides filtering capabilities without complex additional hardware. Figure 5.14 shows the percentage of elements (i.e. indices with their adjacent data) processed by the IRU which are filtered out or merged. We apply the filtering to both SSSP and PR. On average, 48.5% of the elements are filtered by the IRU. This rather high percentage does not directly indicate that a similar amount of accesses to memory are discarded, yet it contributes on top of the IRU reordering improvement. This situation is explained due to its interaction with software graph filtering schemes. Some of the kernels instrumented already employ filtering schemes in their code, typical in graph-based applications, thus the filtering does not always contribute significantly in memory accesses reduction. However, it efficiently filters elements avoiding evaluating the more costly software filtering schemes of graph algorithms, which improves overall performance.

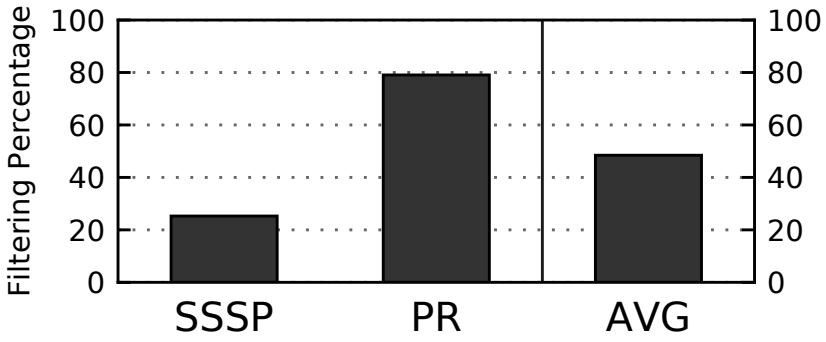


Figure 5.14: Filtered percentage of elements processed by the IRU in our IRU enabled GPU system. The IRU achieves significant filtering effectiveness for different graph algorithms.

5.4.3 Performance Evaluation

IRU provides performance improvement across all algorithms and benchmarks, as seen in Figure 5.15. On average the IRU achieves a speedup of 1.33x, with average speedups of 1.16x, 1.14x and 1.40x for BFS, SSSP and PR respectively. PR experiences higher speedups due to significantly larger reduction of L2 accesses due to the IRU filtering, which merges data and avoids costly atomic L2 accesses. SSSP achieves the lowest speedup due to lower filtering effectiveness.

Overall the performance improvements come from two sources. First, the improved memory coalescing due to the IRU reordering of indices used for irregular accesses, this reordering reduces contention on the memory hierarchy. Second, the IRU filtering and merging that enables further reduction of accesses, interconnection NoC traffic and redundant use of the Execution Units of the GPU.

5.4.4 Energy Evaluation

Figure 5.15 also shows the energy savings achieved with the IRU, which are significant across all graphs and datasets. On average, the IRU achieves an energy reduction of 13%, with reductions of 17%, 5% and 15% for BFS, SSSP and PR respectively. Energy savings are more limited than performance improvements since the IRU greatly reduces L1 and L2 accesses but achieves a more modest reduction of main memory accesses, main memory representing a very significant portion of the total energy consumed. The IRU energy overhead represents a small 0.5% of the final energy.

Overall, energy savings are obtained from several sources. First, the reduced accesses to L1 and L2 and contention to the memory hierarchy. Second, the reduced execution time cuts down on the static power and thus, the overall energy consumption of the GPU system. Third, the energy efficient IRU which enables the reduction in accesses and contention, and allows more efficient filtering than the costly filtering employed by the graph applications. Finally, the

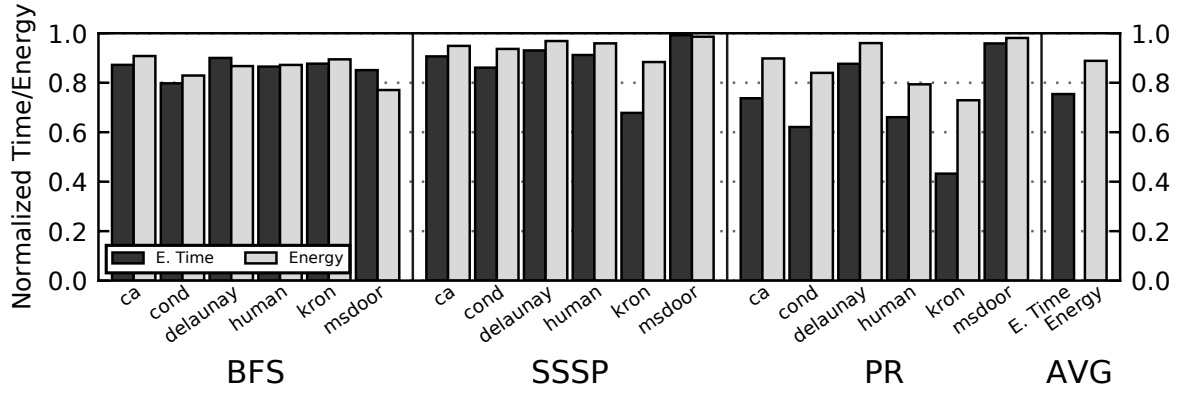


Figure 5.15: Normalized execution time and normalized energy consumption achieved by the IRU enabled GPU with respect to the baseline GPU system. The IRU shows consistent speedups and energy savings achieved across BFS, SSSP and PR graph algorithms and all the different datasets.

IRU reordering leads to a reduction in main memory accesses which contributes to the achieved energy reduction.

5.4.5 Area Evaluation

Our evaluation of the IRU energy and area estimations indicate that the IRU requires a total of 23.9 mm^2 when adding up all the 4 partitions of our GPU system with a GTX980, each partition being 5.98 mm^2. The entire IRU represents 5.6% of the total GPU area. Overall, the IRU is a very compact and efficient hardware which manages to deliver significant performance and energy savings with minimal area requirements.

5.5 Conclusions

In this chapter we propose the Irregular accesses Reorder Unit (IRU), a GPU extension that improves performance and energy efficiency of irregular applications. Efficient execution of irregular applications on GPU architectures is challenging due to low utilization and poor memory coalescing, which force programmers to carry out complex code optimization techniques to achieve high performance. The IRU is a novel hardware unit that delivers improved performance and overall memory traffic of irregular applications by reordering data serviced to the threads. This reordering is made possible by relaxing the strict relationship between threads and data processed.

We extend the IRU to filter out repeated elements while performing the reordering, this results in increased performance by greatly reducing redundant GPU activity. The IRU reordering and filtering optimization delivers 1.32x improved memory coalescing, significantly reducing the

traffic in the memory hierarchy by 46%. Our IRU augmented GPU system achieves on average 1.33x speedup and 13% energy savings for a diverse set of graph-based applications and datasets, while incurring in a 5.6% area overhead.

6

Combining Strengths of the SCU and IRU

This chapter provides details of the IRU-enhanced SCU (ISCU) first introduced in Section 1.4.3. The chapter is organized as follows. First, Section 6.1 reviews the strengths of SCU and IRU for graph processing, motivating our ISCU proposal. Section 6.2 presents the architecture of the ISCU and reviews its processing. Afterwards, Section 6.3 describes its programmability, showcasing the instrumentation of graph applications. Section 6.4 presents the experimental results obtained. Finally, Section 6.5 sums up the main conclusions of this work.

6.1 Introduction

Graph-based applications are ubiquitous in important domains such as data analytics, machine learning and many other examples. Road navigation and self-driving cars, recommendation systems and speech recognition are paradigmatic examples of graph processing workloads. As explored in Section 1.1, current trends towards increased data gathering and knowledge-based society and applications result in increased importance of graph-based applications and, at the same time, a demand for higher data processing capabilities, which motivates high-throughput graph processing on GPGPU architectures.

As reviewed in Section 2.2, GPGPUs achieve high performance for regular programs showing low branch and memory divergence. Unfortunately, graph algorithms exhibit sparse memory accesses, as they traverse unstructured and irregular data with unpredictable patterns. Hence, GPGPU implementations of graph algorithms show significant memory divergence, which leads to high contention in the memory hierarchy and poor utilization of the functional units, limitations discussed in Section 1.2. Not surprisingly, recent work has focused on improving graph processing on GPGPUs through software-level optimizations as seen in Section 1.3.4, while graph frameworks such as Gunrock [161], nvGRAPH [107], HPGA [170] or MapGraph [37] have been introduced

in recent years. Despite all these efforts, we found that state-of-the-art CUDA implementations still suffer from high contention in the memory hierarchy and low utilization of the functional units, as low as 13.5% on average in our graph datasets as determined in Section 1.4.2.

Chapter 4 explores one of the most effective optimizations for GPGPU graph processing, which is stream compaction. Stream compaction gathers data compacting graph datasets in contiguous memory, resulting in much more regular memory access patterns. However, the GPU inefficiency at performing stream compaction leads to it representing a large fraction of execution time as shown previously in Figure 4.1. Consequently, recognizing the importance of stream compaction in graph processing we proposed the Stream Compaction Unit (SCU) hardware extension in Chapter 4. The SCU is tailored to perform stream compaction operations efficiently while the remaining graph-based algorithm steps are executed on the SMs. As such, graph exploration benefits from both parallelism and operating with SCU-prepared data, improving efficiency.

The SCU pre-processing improves memory coalescing and, in addition, it performs filtering of repeated and already visited nodes during the compaction process, significantly reducing GPGPU workload. As explored in Section 4.3, filtering duplicated and already visited elements is key for high-performance graph processing, as software based solutions require either expensive atomic synchronization solutions or imprecise filtering approaches. Our SCU employs a hash table, stored in the L2 cache, to track already processed nodes/edges. According to our measurements, a large amount of data is moved between the SCU and the L2 partitions just for the filtering. More specifically, we have measured that 57% of the traffic in the NoC is due to the filtering operations of the SCU. We claim this is an important limitation and we improve the SCU design to avoid this bottleneck.

As previously introduced in Chapter 5, our Irregular accesses Reordering Unit (IRU) shows a more efficient design for filtering. Instead of offloading the entire stream compaction, the IRU improves coalescing of irregular memory accesses by reordering the node/edge frontier on-the-fly, so threads within the same warp receive nodes/edges stored in the same cache line. During this reordering, the IRU filters duplicated elements but, unlike the SCU, it is located inside the GPU memory partitions and it performs the filtering directly in the L2, reducing traffic in the NoC by a large extent.

Table 6.1 summarizes both approaches, SCU and IRU. The SCU achieves significant speedups and large energy savings, but it produces high contention in the NoC since a lot of data is moved between the L2 and the SCU for the filtering operation. On the other hand, the IRU achieves more modest energy savings, since it focuses on irregular load operations while most of the stream compaction still runs on the GPU, but its filtering operation is highly efficient as it is done inside the memory partitions, largely reducing contention in the NoC and resulting in 46% lower traffic compared to the SCU.

In this chapter we show that the IRU and the SCU have interesting synergies and we propose a novel GPU design that effectively combines both techniques and leverages the strengths of both approaches. We do so by exploiting the SCU to offload the stream compaction operation. However, we modify the behavior of the filtering operation. Instead of fetching data from L2 and performing the filtering in the SCU, our SCU issues requests to the IRU to filter repeated elements in the memory partitions. In this manner, the IRU ameliorates the main bottleneck of

Table 6.1: Comparison between SCU, IRU and ISCU hardware extensions for Graph Processing on GPGPU architectures, showcasing their main functionality and performance metrics.

	GPU	GPU+SCU	GPU+IRU	GPU+ISCU
Offloaded task	Nothing	Stream Compaction	Irregular Loads	Stream Compaction
Speedup	-	1.37x	1.33x	2.2x
Energy savings	-	84.7%	13%	90%
Area overheads	-	3.3%	5.6%	8.5%
NoC contention	High	High	Low	Low
Node/Edge filtering	In software (expensive)	In SCU (data in L2)	In L2	In L2

the SCU, achieving the benefits of both solutions: large energy savings due to offloading stream compaction to a specialized unit and low contention in the NoC since filtering of duplicated elements is done inside the memory partitions. We call this system the IRU-enhanced SCU (ISCU).

Finally, we evaluate the ISCU for a wide variety of state-of-the-art graph-based algorithms and applications. Results show that the ISCU obtains a speedup of 2.2x and 90% energy savings derived from a high reduction of 78% in memory accesses, while incurring in 8.5% area overhead.

This chapter focuses on improving the performance of graph processing on GPGPU architectures. The main contributions are the following:

- We characterize the bottlenecks of the SCU. We observe that the main limiting factor is the large amount of data movement between the L2 cache and the SCU for the filtering operation, which represents 57% of NoC traffic.
- We identify the synergies between the SCU and the IRU, and show that they perfectly complement each other. Based on this observation, we propose the ISCU, a novel GPU extension that combines both the efficient SCU and the filtering mechanism of the IRU to improve overall graph processing efficiency.
- We evaluate our proposal on top of a modern GPU architecture. Our experimental results show that the ISCU improves performance by 2.2x and delivers 90% energy savings for a diverse set of graph-based applications over a GTX 980 GPU. Compared to the GPU+SCU, our ISCU improves performance by 63%, while achieving 66% energy savings.

6.2 IRU-enhanced SCU (ISCU)

In this section we present the IRU-enhanced SCU (ISCU), a GPGPU hardware extension targeting graph processing applications. The ISCU improves the SCU by utilizing the IRU hardware extension to perform pre-processing operations, in particular the filtering of duplicated nodes/edges. The ISCU combines the powerful SCU optimizations obtained by offloading stream compaction operations with the efficient hashing mechanism used in the IRU. Figure 6.1 showcases a GPGPU architecture featuring the ISCU and IRU partitions located in the GPGPU

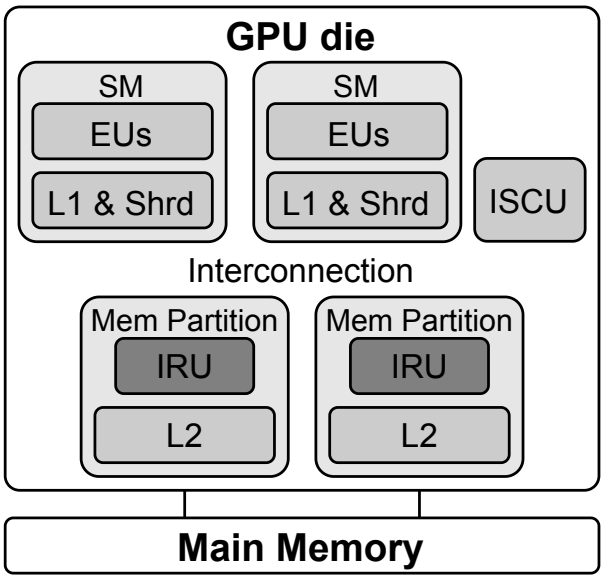


Figure 6.1: Overview of a GPGPU architecture featuring the ISCU attached to the interconnection as well as the IRU located in the Memory Partitions.

Memory Partitions (MP). The ISCU hardware extension is motivated by SCU's bottleneck experienced when performing pre-processing filtering and grouping optimizations for graph processing applications.

SCU's main bottleneck arises from the limited interconnection throughput to the L2 and due to the memory accesses required to perform filtering/grouping operations through the in-memory hash table. Figure 6.2 shows the utilization of the filtering/grouping unit, measured as the percentage of cycles this data pre-processing unit is being utilized over the total execution, and the percentage of Network-on-Chip (NoC) traffic generated by it. As it can be seen, utilization of this component is high during the execution of the compaction operations, reaching 92% of the execution for BFS and an average of 51% for the different graph algorithms. Furthermore, it is responsible for a significant amount of traffic and accesses to the interconnection, as much as 80% for BFS and an average of 58% for the different graph algorithms. Consequently, this component's high utilization of the pipeline and NoC limits performance and provides an opportunity for optimization.

The memory accesses that saturate NoC throughput come from several sources. First, from fetching the sparse data and then writing the elements in the compacted array. Second, from the parameters used in the operations. Finally, when doing pre-processing, several accesses are required to retrieve and operate with the in-memory hash table. Nonetheless, the high filtering efficiency achieved, reaching up to 76% of the workload, reduces significantly the accesses required for the data compaction operations themselves, consequently accesses to the in-memory hash table represent a larger split and become a significant bottleneck.

Insertion of elements to the in-memory hash table requires several accesses. For filtering, first it requires fetching the tag entry, and performing the corresponding comparison. In case of a miss, tag and data entries have to be updated. Consequently, processing an element (edge or

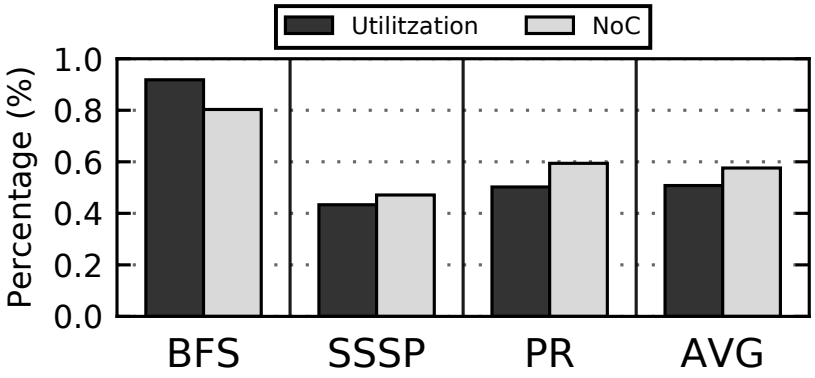


Figure 6.2: Utilization of the SCU pre-processing component (i.e. filtering/grouping unit) and the percentage of NoC traffic devoted to filtering/grouping operations. The utilization is the percentage of cycles that the filtering/grouping unit is active over total execution. The SCU invests a large number of cycles and NoC transactions in the data pre-processing operations.

node) incurs in multiple accesses to the L2. Although the SCU in-memory hash table design is multi-banked, the throughput to L2 is limited to a single access from the *Filtering/Grouping* component per cycle, severely affecting the performance of hash insertions.

We propose to use the IRU efficient distributed hash table as a replacement of the in-memory hash used for the SCU pre-processing, additionally increasing the throughput of requests to the IRU. The resulting system that we term ISCU contains both SCU and IRU hardware extensions with our modifications to fit the requirements of the end system.

6.2.1 Hardware Modifications

The main changes in the SCU are modifications to the *Data Fetch* and *Filtering/Grouping* component shown in Figure 4.8. Due to the data path changes reviewed in Section 6.2.2, *Data Fetch* is only required to issue the fetch operation, yet the IRU is the hardware receiving that data, avoiding unnecessary data movements. Similarly, the *Filtering/Grouping* component logic is largely removed since it was responsible to manage request to the in-memory hash table. Additionally, the coalescing unit attached to this component is no longer required as it was used to merge requests to tag and data entries.

Similarly, adapting the IRU requires minor hardware changes. Additional control logic is required to support configuring the IRU to perform SCU pre-processing. This control logic modifies the data-path, so compaction operations are performed at different locations in the system: the SCU will initiate the requests to the data, but the replies from the memory controller will be directly passed to the IRU, that is located inside the L2 partition. In this manner, data can be filtered and reordered in the IRU without being transferred to the SCU through the NoC, saving NoC bandwidth by a large extent. In our system, the IRU does not use the prefetcher to gather data, since the SCU is in charge of orchestrating the stream compaction operation and it takes care of generating the read requests to the memory controller. Additionally, the *Data*

Replier does no longer require to gather requests, and can send replies back directly when data from the *Reordering Hash* is ready. Although the *Prefetcher* is not utilized for the ISCU, this structure is maintained to provide the IRU main functionality for other CUDA kernels.

The hash table mechanism of the original SCU is not bound by on-chip memory size as it is stored in main memory and cached in the L2. In comparison, the IRU includes limited on-chip storage for the hash table. This reduces memory bandwidth usage at the cost of less accurate filtering of duplicated nodes, since in case of conflicts in the hash table the old data is evicted. We have observed that with a modest size of 80KB per memory partition the filtering mechanism is highly effective, as it is able to avoid the vast majority of duplicated elements.

6.2.2 Detailed Processing

The ISCU has two main internal processing data-flows which are represented in Figure 6.3a for regular operations, and in Figure 6.3b for data pre-processing, both corresponding to the SCU operations listed in Section 4.2.1. Note that the *Data Processing* ISCU component listed in Figure 6.3 is in fact a simplification of Figure 4.8 containing both the *Bitmask Constructor* and the *Filtering/Grouping* components.

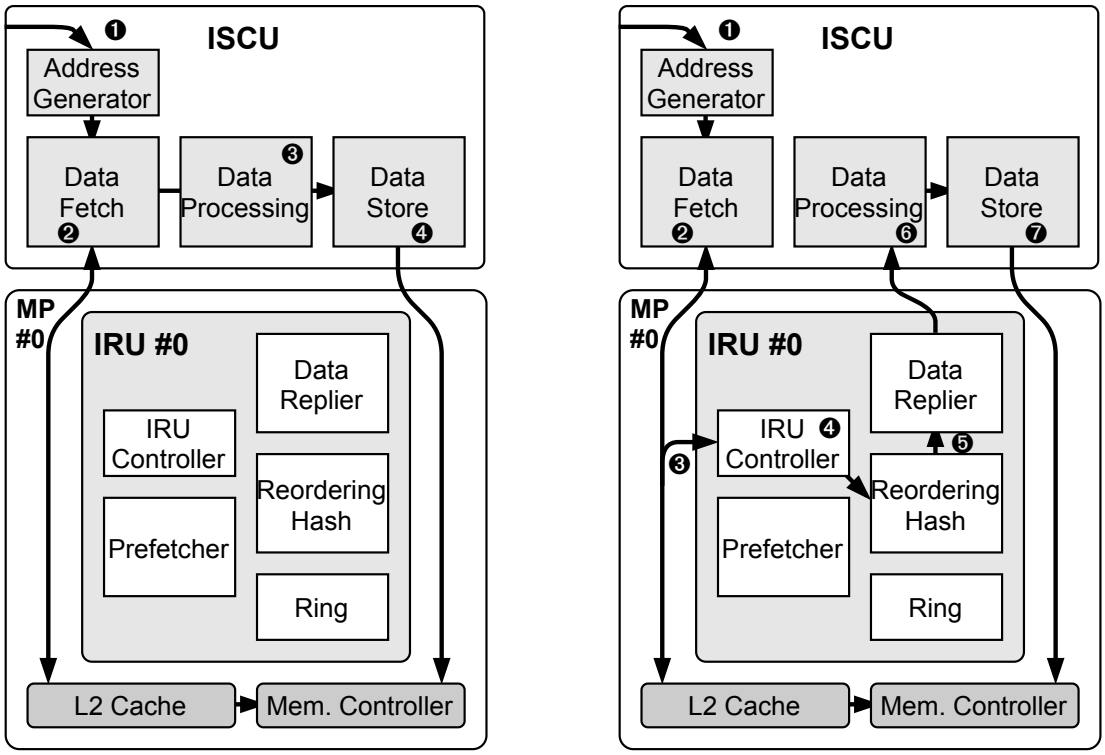
Regular ISCU operations

The internal processing and data-flow of regular ISCU operations is shown in Figure 6.3a. Initially, an operation is issued from the host which configures the ISCU with the required parameters and starts the execution ❶. This initializes the Address Generator to fetch parameters and start fetching from L2 the sparse data to compact ❷. Afterwards, according to the corresponding operation, some processing is applied to the data, such as replication or indirection ❸. Finally, the sparsely gathered data is compacted and written directly to main memory ❹.

Pre-processing ISCU operations

The behavior and data-flow of pre-processing ISCU operations is shown in Figure 6.3b. The initial configuration ❶ and fetching of the sparse data ❷ is done the same way as regular ISCU operations. Additionally, the ISCU pre-processing operation configures the *IRU Controller* to receive and process data from the ISCU.

The processing and data-flow changes with respect to the SCU start in the *Data Fetch* component. The data fetched by the ISCU is directly sent to the IRU ❸. In this manner, duplicated and already visited nodes/edges are not transferred to the ISCU since they are removed in the memory partition, saving NoC bandwidth by a large extent. The elements used for pre-processing are then forwarded to the corresponding IRU through the *Ring*, if the hashing function dictates it. Afterwards, these elements are inserted into the hash table performing the corresponding filtering or reordering operation ❹. When a hash entry is ready or no more data is to be inserted, the pre-processed data is forwarded to the *Data Replier* ❺, which sends a reply to the ISCU. Since the resulting pre-processed data has to be written to main memory, it



(a) Behavior and data-flow for a regular ISCU operation, i.e. compaction operations.

(b) Behavior and data-flow for a pre-processing ISCU operation, i.e. filtering/grouping.

Figure 6.3: Overview of the behavior and data-flow for the different ISCU operations performed on the ISCU hardware extension together with its interactions with the IRU extension.

might be destined to a different memory partition and so the ISCU handles the final writing. Finally, the ISCU creates the corresponding filtering/grouping vectors ❻ which are then written in memory directly by the *Data Store* ❼.

IRU operations

Finally, the ISCU also allows to perform IRU operations onto the IRU hardware as detailed in Section 5.2.2 and Figure 5.4. The ISCU performs stream compaction operations without the use of the GPU hardware, while the IRU extension is available to optimize irregular accesses issued by the GPU.

6.3 ISCU Programmability

This section provides an overview of the complete ISCU programming model as well as how an ISCU instrumented graph application operates with its different components. Note that, since the modifications introduced in the ISCU extension are architectural, the programming

model remains largely unaffected. Finally, as an illustrative example, an instrumentation of the Pagerank is with pre-processing is reviewed.

Our system performs stream compaction with the SCU operations which are described in Section 4.2.1. These operations are performed by issuing them from the Host (CPU) and execute on the ISCU, without involving the GPU in the process. Additionally, the pre-processing operations described in Section 4.3 which perform filtering or grouping execute utilizing both ISCU and IRU hardware extensions and similarly do not utilize the SMs of the GPU.

Furthermore, our system allows to perform the irregular access optimizations described in Chapter 5 by utilizing the IRU independently from the ISCU hardware extension. The IRU optimizations are achieved by using the programming model described in Section 5.3 which involves both the SMs of the GPU and the IRU hardware to deliver irregular access coalescing improvements.

6.3.1 Graph Processing Instrumentation

We instrument state-of-the-art implementations of BFS, SSSP and PR to utilize both stream compaction and irregular accesses optimizations offered by the ISCU. We use the stream compaction instrumentation described in Section 4.3 for both BFS and SSSP. We provide a new version of PR that performs filtering, which does not improve performance for the regular SCU due to the previously mentioned bottlenecks.

After the SCU offloading, the GPU is still responsible for the processing of the graph exploration, as only the data compaction efforts are offloaded. The *contractionGPU* kernels present in all graph algorithms, which are show in Figure 4.9 and Figure 4.10, are still executed in the GPU. For these kernels, the IRU is used to improve memory coalescing as described in Section 5.3 and following the changes performed in Figure 5.8 for BFS, Figure 5.9 for SSSP and Figure 5.10 for PR.

PageRank ISCU instrumentation

With the ISCU we manage to remove the bottlenecks of the SCU extension, which allows us to utilize filtering pre-processing operations for PR.

Removing duplicated or already visited nodes is not an option for PR since it requires to consider all the nodes' ranks on every iteration of the algorithm. The Update phase of the PR requires the use of atomic operations to correctly add the weights, a mechanism that is very costly, especially in large graphs. The filtering operation of the ISCU can be employed to compute the new ranks instead of using a large number of expensive atomic operations in the GPU. In other words, the filtering hardware in the IRU can be used to perform a reduction operation, adding the weights of duplicated nodes. Shown on Figure 6.4, the required changes are the following. For the expansion phase we include an additional ISCU operation with the filtering mechanism that generates the filtering vector. Furthermore, as described earlier, the IRU is used to provide irregular accesses improvement to other PR kernels. This IRU optimization is

described in Figure 5.10 in which the workload of the *Contract* kernel of the PR is optimized achieving significant performance and energy improvements covered in Section 5.4.

```
1  void PR_Expand (nodes) {
2      indexes, count = preparationGPU (nodes);
3      filtering = accessExpansionCompactionSCU (edges, indexes, count);
4
5      edge_frontier = accessExpansionCompactionSCU
6          (edges, indexes, count, filtering);
7      weight_frontier = replicationCompactionSCU
8          (weights, count, filtering);
9      return edge_frontier, weight_frontier;
10 }
```

Figure 6.4: Pseudo-code of the additional operations for a GPGPU PR program to use the ISCU.

6.4 Experimental Results

In this section, we evaluate the improvements in performance and energy consumption achieved by our ISCU scheme. We evaluate four different configurations. Configuration labeled as *GPU* represents a pure software-based CUDA implementation of the graph algorithms running on an NVIDIA GTX 980. Configuration labeled as *SCU* is the system presented in Chapter 4 that combines the GPU and the SCU. The system labeled as *IRU* is the GPU extended with the IRU hardware as described in Chapter 5. Finally, configuration labeled as *ISCU* represents our scheme as described in Section 6.2.

We first evaluate the energy consumption and performance of the ISCU in sections 6.4.1 and 6.4.2 respectively, using as the baseline the NVIDIA GTX 980. Afterwards, we compare the performance and energy of the ISCU with the SCU and the IRU in Section 6.4.3. Finally, we analyze the memory improvements of the ISCU in Section 6.4.4 and discuss its area requirements in Section 6.4.5.

6.4.1 Energy Evaluation

Offloading compaction operations to our ISCU provides consistent and large energy savings. Figure 6.5 shows the normalized energy consumption achieved by the ISCU over the baseline GPU system for all graphs and datasets. Additionally, the figure indicates the source of the remaining energy consumption distinguishing between the GPU, the majority, and the ISCU. On average, the ISCU delivers a reduction of 90% in energy consumption, achieving an energy reduction of 92%, 91% and 85% for BFS, SSSP and PR respectively. Overall, energy savings are obtained from several sources. First, stream compaction offloading to a more efficient ISCU hardware reduces static and dynamic energy consumption required to perform compaction operations. Second, the workload filtering and memory coalescing provided by the ISCU improves GPGPU

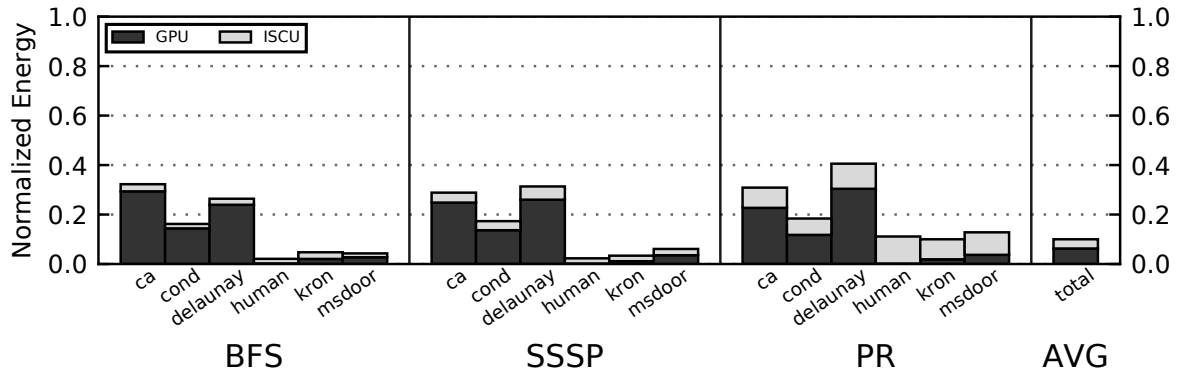


Figure 6.5: Normalized energy consumption of the ISCU enabled GPU with respect to the baseline GPU system (GTX 980), showing the split between GPU and ISCU energy consumption. Significant energy savings are achieved across BFS, SSSP and PR graph algorithms and every dataset.

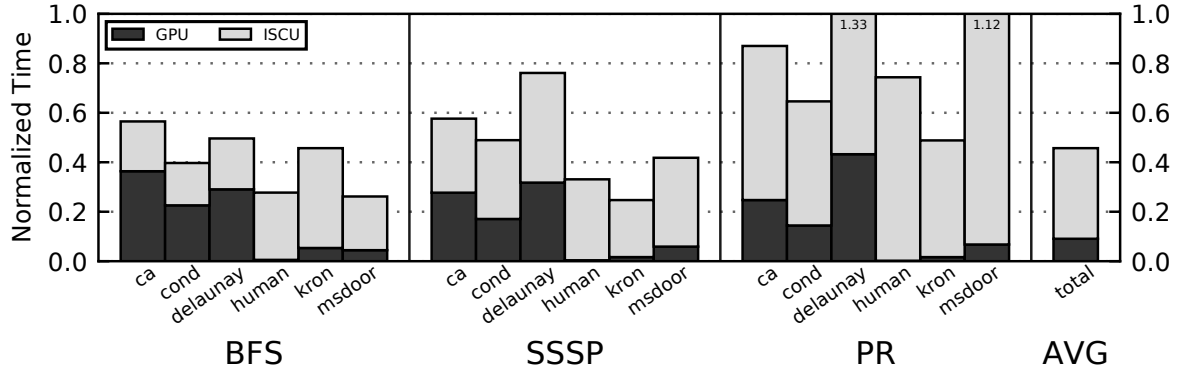


Figure 6.6: Normalized execution time of the ISCU enabled GPU with respect to the baseline GPU system, showing the split between GPU and ISCU execution time. Significant speedups are achieved across BFS, SSSP and PR graph algorithms and the majority of the datasets.

resources utilization, which lowers overall GPGPU energy consumption. Third, irregular access optimization enabled by the IRU further reduces memory contention. Finally, performance speedup further reduces static energy consumption of the system. Note that graph datasets with higher inter-connectivity see more energy savings due to higher computation offloading and increased workload filtering.

6.4.2 Performance Evaluation

The ISCU delivers significant speedups across different graph algorithms and datasets as seen in Figure 6.6, which shows normalized execution time using the ISCU over the baseline

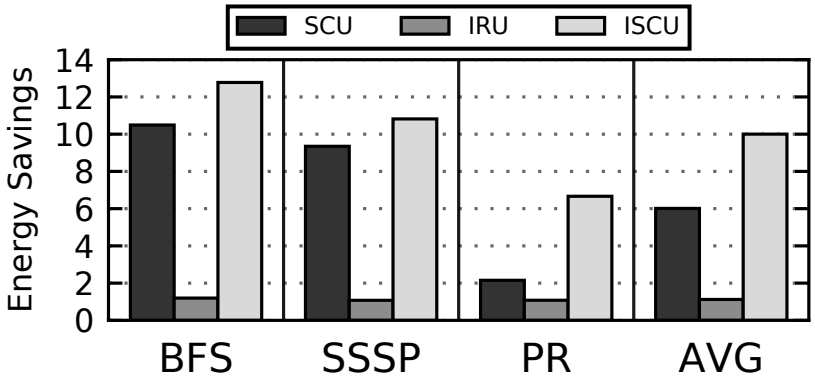


Figure 6.7: Energy savings of the SCU, IRU and ISCU with respect to the baseline GPU system. The ISCU synergetically improves energy savings achieved with SCU and IRU.

GPU system. Additionally, the figure indicates the split of the execution time between the GPU and the ISCU, highlighting the high performance improvements over GPU execution of the higher inter-connectivity graphs. On average, the ISCU achieves a speedup of 2.2x with average speedups of 2.8x, 2.56x and 1.44x for BFS, SSSP and PR respectively. The efficiency of the ISCU is not as high for PR which in some cases incurs in overheads, a consequence of the large frontiers due to PR exploring the entire graph at every iteration. For PR, since every element is accessed on each iteration, all the data in the graph dataset is accessed incurring in less sparse accesses and higher locality. Nonetheless, the overheads observed are compensated by the high reduction in energy achieved due to the offloading of the compaction operations.

Overall, performance improvements are obtained from several sources. First, the efficient execution on hardware tailor-made for stream compaction operations delivers better performance than GPU architectures. Second, the ISCU pre-processing reduces GPGPU workload, additionally reducing GPGPU atomic synchronization overheads and improving memory coalescing. Third, the irregular accesses optimization enabled by the IRU further improves GPGPU performance by increasing memory coalescing.

6.4.3 Comparison with SCU and IRU

We compare the energy savings and speedups achieved with the ISCU against previous GPGPU architectural extensions for graph processing. Figure 6.7 shows how by combining in the ISCU the strengths of the SCU and IRU we are able to achieve a synergistic energy improvement, reaching on average a huge 10x improvement in energy consumption compared to the GPU baseline, even though the SCU and IRU achieved on average 6x and 1.13x respectively. The big factor contributing to energy savings is delegating stream compaction operations to our specialized compaction hardware, as such the IRU optimizations do not deliver such huge energy savings. Note that the less sparse exploration performed by PR reduces its the energy savings. Furthermore, the ISCU avoids a large percentage of NoC transactions compared to the SCU, as the filtering is performed directly in the memory partitions.

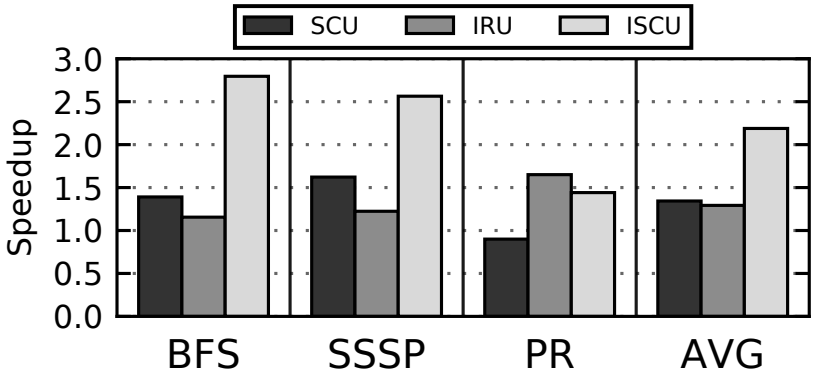


Figure 6.8: Speedup of the SCU, IRU and ISCU with respect to the baseline GPU system. The ISCU synergetically improves speedups achieved with SCU and IRU. The ISCU is able to overcome the SCU overheads which slowed down PR, delivering significant speedups.

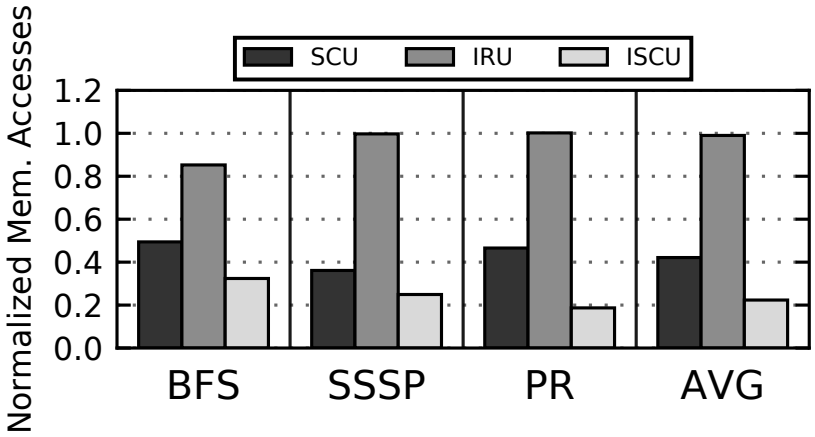


Figure 6.9: Normalized memory accesses of the SCU, IRU and ISCU with respect to the baseline GPU system. The ISCU synergetically improves the memory reduction achieved with SCU and IRU due to the optimizations which significantly reduce memory traffic.

Performance improvements are also synergistic as seen in Figure 6.8, where the ISCU achieves on average a significant 2.2x speedup compared to the baseline GPU, while the SCU and IRU deliver on average 1.37x and 1.33x speedups respectively. The ISCU manages to overcome the overheads that impact the PR filtering enhanced SCU instrumentation, which enable higher performance and energy savings. Although the IRU achieves a better speedup for PR than the ISCU, the minimal performance difference is more than made up by the significant difference in energy efficiency reaching 6.66x for ISCU against a 1.08x for the IRU.

6.4.4 Memory Improvements Evaluation

The ISCU significantly contributes to reduce memory accesses performed by graph processing applications as seen in Figure 6.9. The ISCU achieves on average a reduction of 78% in the total memory accesses performed by the baseline GPU, while the SCU and IRU achieve a reduction of 58% and 1% respectively. Although IRU memory accesses reduction is low, it significantly contributes to reduce intra-GPU memory resource utilization and interconnection traffic.

6.4.5 Area Overhead Evaluation

We evaluate the overhead of the complete ISCU extension which contains the improved SCU and the IRU hardware extensions. We do so by synthesizing and characterizing the different components, which require a total of 37.17 mm^2 additional area for the proposed system. Considering the overall GPU system, the ISCU represents a 8.5% of the total GPU area. The ISCU area overhead is very low given the high energy savings and speedups achieved. In terms of both performance/area and energy, the ISCU results in very high benefits compared with the baseline and the SCU and IRU solutions.

6.5 Conclusions

In this chapter we propose the IRU-enhanced SCU (ISCU), a GPGPU hardware extension that efficiently performs stream compaction operations commonly used by graph-based applications. The ISCU combines the strengths of the SCU and IRU hardware extensions to synergistically achieve high performance and energy-efficiency for GPGPU graph-based applications.

The ISCU solves the bottlenecks caused by the in-memory hash table used in the SCU to filter duplicated elements, that requires a large amount of traffic in the NoC. We propose to leverage the efficient IRU hash mechanism to perform filtering operations in the memory partitions, saving NoC traffic by a large extent and achieving significant speedups and energy savings.

The ISCU optimizations for graph processing operations deliver on average a 2.2x speedup and a reduction of 90% in energy consumption for a diverse set of graph-based applications and datasets, while achieving a high reduction of 78% in memory accesses, at the expense of a 8.5% GPU area overhead.

7

Conclusions and Future Work

This last chapter presents the main conclusions of the thesis, summarizes the contributions and finally closes with possible open-research areas for future work.

7.1 Conclusions

This dissertation has analyzed the performance of state-of-the-art irregular graph processing in GPGPU architectures. Our evaluations have led us to several contributions in the form of hardware extensions and deeper understanding of challenges of graph processing in GPGPU architectures, resulting in the following main conclusions.

In first place, we characterize state-of-the-art graph applications on GPGPU architectures. We observe that up to 55% of the execution time is dominated by stream compaction operations, while the remaining time is dedicated to graph exploration execution. We observe that even though GPU stream compaction takes a significant amount of time, it is executed inefficiently due to their low-computation, data-movement type of operation. With these observations we come up with a proposal to extend the GPU architecture with a Stream Compaction Unit (SCU) to improve performance and energy-efficiency for graph processing. The SCU is tailored to the requirements of the stream compaction operation, while the rest of the graph processing algorithm is executed on the streaming multiprocessors achieving high GPU efficiency.

We extend the SCU hardware to pre-process the data serviced to the GPU with the objective to improve the GPU efficiency as it will process SCU-prepared compacted data. We achieve this through two approaches. First, filtering out duplicated and already visited nodes during the compaction process, which reduces the number of GPU instructions by more than 70% on average. Additionally, we implement a grouping operation that writes together in the compacted

array edges whose destination nodes are in the same cache line, improving memory coalescing by 27% for the remaining GPU workload. Overall, the resulting High-Performance and Low-Power GPU system including our SCU unit achieves significant speedups of 1.37x and 2.32x, and 84.7% and 69% energy savings respectively on average for several graph-based applications, with a small 3.3% and 4.1% increase in overall area respectively.

In second place, we characterize the degree of memory coalescing and GPU utilization of modern graph-based applications. Our analysis shows poor memory coalescing poor (four accesses per warp) and low GPU utilization (13.5% on average). Current techniques to improve this negative aspects of graph-based applications are complex and require extensive hardware knowledge from programmers, while at the same time they impose strict constraints. Based on this analysis, we propose the Irregular accesses Reorder Unit (IRU), a GPU extension that improves performance and energy efficiency of irregular applications. The IRU is a novel hardware unit that delivers improved performance and overall memory traffic of irregular applications by reordering data serviced to the threads. The reordering is facilitated by relaxing the strict relationship between threads and data processed.

We extend the IRU to filter out and merge repeated elements while performing the reordering, achieving performance improvements by greatly reducing redundant GPU workload with minimal cost. Overall, the IRU reordering and filtering optimization delivers 1.32x improvement in memory coalescing, significantly reducing by 46% the traffic in the memory hierarchy. Our IRU augmented GPU system achieves on average 1.33x speedup and 13% energy savings for a diverse set of graph-based applications and datasets, while incurring in a 5.6% area overhead.

Finally, we characterize the limitations of the SCU concluding that its major bottleneck is the large amount of data movement between itself and the L2 cache. These data transfers between the SCU and the L2 are caused by the pre-processing operation with an in-memory hash, which contributes to 57% of Network-on-Chip (NoC) traffic. With this insight we realize we can leverage the IRU hardware strengths as its optimizations are highly effective at reducing contention in the NoC. Based on this observation, we propose the ISCU, a novel GPU extension that combines both systems, SCU and IRU. The ISCU leverages the powerful stream compaction offloading achieved by the SCU and the efficient filtering mechanism of the IRU. In other words, the SCU employs the IRU to filter duplicated nodes/edges directly in the GPU memory partitions, reducing NoC traffic by a large extent.

The proposed ISCU system improves overall graph processing efficiency by combining the strengths of SCU and IRU. When implemented on top of an NVIDIA GTX 980, the ISCU achieves a reduction of 78% in memory accesses, reduces energy consumption by 90% and provides 2.2x speedup on average for a diverse set of graph-based workloads, while incurring in a 8.5% area overhead.

In summary, the experimental results presented in this dissertation show that GPU graph processing incurs in many inefficiencies such as requiring stream compaction operations, experiencing high memory divergence, increasing memory hierarchy traffic and achieving low GPU resource utilization. We propose several solutions to improve GPU graph processing. First, offloading stream compaction operations to specialized SCU hardware. Second, delivering improved coalescing, memory traffic reduction and performance improvement by reducing irregular accesses divergence with the IRU hardware. Finally, we attain an overall improved

graph performance and energy-efficient GPU architecture with the ISCU which synergistically combines the strengths of both our previous SCU and IRU contributions.

7.2 Contributions

In this dissertation different techniques have been proposed to improve irregular graph processing performance on GPGPU architectures. We have explored the different challenges and limitations of the memory hierarchy of GPGPU architectures, and as a result, we have come up with multiple GPGPU hardware extensions improving overall performance, energy efficiency and memory contention of graph processing. The contributions of this dissertation are summarized as follows.

In first place, we propose and design the Stream Compaction Unit (SCU), a GPGPU architecture extension capable of efficiently processing stream compaction operations required for graph processing. Our design leverages hardware offloading and data pre-processing which provide powerful improvements for stream compaction operations. The hardware offloading enables significant energy savings while the data preprocessing improves performance by reducing workload and increasing overall memory coalescing. Finally, by providing hardware scalability characteristics we evaluate our proposal for both High-Performance and Low-Power GPUs, achieving huge energy savings and important speedups for both GPU designs as evaluated in Chapter 4. This work has been published in the 46th International Symposium on Computer Architecture (ISCA) [139]:

- "SCU: A GPU Stream Compaction Unit for Graph Processing".
  Albert Segura, Jose-Maria Arnau, and Antonio González.
  International Symposium on Computer Architecture, June 2019 (**ISCA '19**).
  DOI: https://doi.org/10.1145/3307650.3322254

We explore further improvement of irregular applications with our second contribution where we propose the Irregular accesses Reorder Unit (IRU), a GPGPU architecture extension targeting the reduction of memory divergence in irregular accesses performed by graph processing. Our design relaxes programming model restrictions which enables hardware-driven remapping optimizations of data processed by threads, leading to improved memory coalescing and overall memory hierarchy improvements. Our design improves the memory coalescing of graph applications while at the same time halving memory hierarchy traffic. The memory hierarchy improvements lead to significant performance and energy improvements for irregular graph processing GPGPU applications as detailed in Chapter 5. This work has been submitted for publication [137].

- "Irregular Accesses Reorder Unit: Improving GPGPU Memory Coalescing for Graph-Based Workloads".
  Albert Segura, Jose-Maria Arnau, and Antonio González.

Finally, in our last contribution we combine the strengths of our previous techniques in a new GPGPU architecture extension which efficiently performs graph processing applications. This design combines the offloading and processing capabilities of the SCU, with the efficient filtering and reordering of irregular accesses of the IRU in a synergistic manner, which enables higher efficiency of graph processing in GPGPU systems and overcomes limitations of our previous contributions. Our design achieves high energy savings and important speedups for graph processing in modern GPGPU architectures as explored in Chapter 6. This work has been submitted for publication [138].

- "Energy-Efficient Stream Compaction Through Filtering and Coalescing Accesses in GPGPU Memory Partitions".
  Albert Segura, Jose-Maria Arnau, and Antonio González.

7.3 Open-Research Areas

Graph processing is a research field which has seen increased interest over the last years motivated by the growing importance of data analytics in today's knowledge-based society. Previous introduction of technologies such as social networks and smartphones and future introduction of new technologies such as Internet of Things (IoT) and 5G mobile networks underpins the importance of graph processing research. The literature on GPU graph processing is not extensive, yet efficiency of graph-based workloads on GPGPU architectures has ample room for improvement.

A compelling extension of the contributions proposed in this dissertation would be to explore synergies with processing in memory and newer 3D stacking memories. Data movement across the memory hierarchy comes at a cost in both energy and performance due to latency and resources utilization. This reason motivates processing data close to memory, specially operations such as stream compaction operations, remapping and other kinds of pre-processing of data. The data processed by such operations is brought into the GPU to be either processed by it or by our proposals (i.e. SCU or IRU), just to be sent back to memory and finally be brought back later in order to be processed by the program. Instead, avoiding this data movement operations and performing the computations close to memory could deliver huge improvements in energy consumption as well a performance and resource utilization, motivating the exploration of synergies between our SCU and IRU contributions and processing in memory approaches. Nonetheless, many processing in memory computation offloading approaches have been proposed over the last years with varying degree of success, often times with increased complexity in programmability of the proposed solutions.

Synergies with other improvements in the memory hierarchy are also worth exploring. Several proposals in the literature, such as Elastic-cache [85] or DyCache [51], have explored to deliver more flexible GPU cache solutions with finer grain caches allocation. Additionally, recent GPGPU architectures introduce new coalescing hardware and *Streaming Throughput-oriented* caches which facilitate more in-flight misses. One of the main motivations of these solutions is the ratio between the data brought to caches and the actual amount of data used: for sparse

and irregular accesses it is low, and thus potential for energy efficiency and performance arises. Synergies could be exploited between these approaches and our SCU and IRU contributions to further reduce data movement and deliver more energy efficient solutions. On the other hand, further granularity of the cache allocation of data could be achieved with approaches exploiting the graph topology with prefetching or prediction approaches, although at a cost of reduced general use.

Similarly, many works have explored standalone accelerator approaches, specially exploring synergies with varied memory technologies. These accelerators have the benefit that they can more easily exploit graph data structures and topology to significantly improve performance. Similar approaches leveraging these technologies in GPU architectures might also deliver large improvements in performance and energy efficiency, yet might incur in high area overheads and low generic use of the added hardware.

A more far-fetched idea could be to leverage DNNs to improve the existing GPU hardware, i.e. use machine learning techniques to improve memory coalescing and reduce data movements. This DNN would be trained to learn the topology of several graphs and predict the memory access patterns. At runtime, the DNN information generated by the GPU could be used by software or hardware to inform prefetching mechanisms, perform remapping to collocate data or inform cache data allocation. This approach could deliver improvements in performance and efficiency leveraging the computational power already present in the GPU, which is inefficiently used by irregular graph-based workloads.

Bibliography

[1] *3dfx Interactive:Voodoo Graphics PCI*. URL: https://en.wikipedia.org/wiki/3dfx_Interactive#Voodoo_Graphics_PCI (visited on 04/2020).

[2] Junwhan Ahn et al. "A scalable processing-in-memory accelerator for parallel graph processing". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 105–117.

[3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[4] Shaahin Angizi et al. "GraphS: A graph processing accelerator leveraging SOT-MRAM". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 378–383.

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[6] Ali Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 163–174.

[7] Peter Bakkum and Kevin Skadron. "Accelerating SQL database operations on a GPU with CUDA". In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 2010, pp. 94–103.

[8] Darius Bakunas-Milanowski et al. "Efficient algorithms for stream compaction on GPUs". In: *International Journal of Networking and Computing* 7.2 (2017), pp. 208–226.

[9] Scott Beamer. "Understanding and improving graph algorithm performance". PhD thesis. UC Berkeley, 2016.

[10] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[11] Nathan Bell and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". In: *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM. 2009, p. 18.

[12] Nathan Bell and Jared Hoberock. "Thrust: A productivity-oriented library for CUDA". In: *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[13] Markus Billeter, Ola Olsson, and Ulf Assarsson. "Efficient stream compaction on wide SIMD many-core architectures". In: *Proceedings of the conference on high performance graphics 2009*. 2009, pp. 159–166.

[14] Guy E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[15] Mariusz Bojarski et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[16] Mariusz Bojarski et al. "Explaining how a deep neural network trained with end-to-end learning steers a car". In: *arXiv preprint arXiv:1704.07911* (2017).

[17] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. "Simultaneous branch and warp interweaving for sustained GPU performance". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2012, pp. 49–60.

[18] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. "A quantitative study of irregular programs on GPUs". In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2012, pp. 141–151.

[19] Shuai Che. "GasCL: A vertex-centric graph model for GPUs". In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2014, pp. 1–6.

[20] Shuai Che et al. "Pannotia: Understanding irregular GPGPU graph applications". In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2013, pp. 185–195.

[21] Xuhao Chen et al. "Adaptive cache management for energy-efficient gpu computing". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 343–355.

[22] Jike Chong, Ekaterina Gonina, and Kurt Keutzer. "Efficient automatic speech recognition on the gpu". In: *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 601–618.

[23] Design Compiler. *Synopsys inc.* 2000.

[24] *cuBLAS*. URL: https://docs.nvidia.com/cuda/cublas/index.html (visited on 04/2020).

[25] *CUDA (Compute Unified Device Architecture)*. URL: https://en.wikipedia.org/wiki/CUDA (visited on 04/2016).

[26] *cuFFT*. URL: https://docs.nvidia.com/cuda/cufft/index.html (visited on 04/2020).

[27] *cuSPARSE*. URL: https://docs.nvidia.com/cuda/cusparse/index.html (visited on 04/2020).

[28] Guohao Dai et al. "Graphh: A processing-in-memory architecture for large-scale graph processing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.4 (2018), pp. 640–653.

[29] Andrew Davidson et al. "Work-efficient parallel GPU methods for single-source shortest paths". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 349–359.

[30] Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), p. 1.

[31] Jeff Dean, David Patterson, and Cliff Young. "A new golden age in computer architecture: Empowering the machine-learning revolution". In: *IEEE Micro* 38.2 (2018), pp. 21–29.

[32] Robert H Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[33] DIMACS. *10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering*. 2010. URL: https://www.cc.gatech.edu/dimacs10/ (visited on 2017).

[34] *DirectX*. URL: https://en.wikipedia.org/wiki/DirectX (visited on 04/2020).

[35] Jeffrey Donahue et al. “Long-term recurrent convolutional networks for visual recognition and description”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 2625–2634.

[36] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[37] Zhisong Fu, Michael Personick, and Bryan Thompson. “Mapgraph: A high level api for fast development of high performance graph analytics on gpus”. In: *Proceedings of Workshop on GRAph Data management Experiences and Systems*. 2014, pp. 1–6.

[38] Wilson WL Fung and Tor M Aamodt. “Thread block compaction for efficient SIMT control flow”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE. 2011, pp. 25–36.

[39] Wilson WL Fung et al. “Dynamic warp formation and scheduling for efficient GPU control flow”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 407–420.

[40] *GDDR5 SDRAM*. URL: https://en.wikipedia.org/wiki/GDDR5_SDRAM (visited on 04/2020).

[41] *GDDR6 SDRAM*. URL: https://en.wikipedia.org/wiki/GDDR6_SDRAM (visited on 04/2020).

[42] *GeForce 20 series*. URL: https://en.wikipedia.org/wiki/GeForce_20_series (visited on 04/2020).

[43] *GeForce 3 series*. URL: https://en.wikipedia.org/wiki/GeForce_3_series (visited on 04/2020).

[44] *GeForce 8 series: GeForce 8800 Series*. URL: https://en.wikipedia.org/wiki/GeForce_8_series#GeForce_8800_Series (visited on 04/2020).

[45] Afton Geil, Yangzihao Wang, and John D Owens. “WTF, GPU! computing twitter’s who-to-follow on the GPU”. In: *Proceedings of the second ACM conference on Online social networks*. ACM. 2014, pp. 63–68.

[46] *General-purpose computing on graphics processing units*. URL: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units (visited on 04/2020).

[47] Prasun Gera et al. “Traversing large graphs on GPUs with unified memory”. In: *Proceedings of the VLDB Endowment* 13.7 (2020), pp. 1119–1133.

[48] Abdullah Gharaibeh et al. “Efficient large-scale graph processing on hybrid CPU and GPU systems”. In: *arXiv preprint arXiv:1312.3018* (2013).

[49] Antonio González. "Trends in Processor Architecture". In: *Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms*. Springer, 2019, pp. 23–42.

[50] NVIDIA GeForce GTX. "980 Whitepaper". In: *NVIDIA Corporation* (2014).

[51] Hui Guo et al. "DyCache: Dynamic multi-grain cache management for irregular memory accesses on GPU". In: *IEEE Access* 6 (2018), pp. 38881–38891.

[52] Sang-Won Ha and Tack-Don Han. "A scalable work-efficient and depth-optimal parallel scan for the GPGPU environment". In: *IEEE Transactions on Parallel and Distributed Systems* 24.12 (2012), pp. 2324–2333.

[53] Tae Jun Ham et al. "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[54] Tianyi David Han and Tarek S Abdelrahman. "Reducing branch divergence in GPU programs". In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM. 2011, p. 3.

[55] Mark J Harris. "Fast fluid dynamics simulation on the GPU." In: *SIGGRAPH Courses* 220.10.1145 (2005), pp. 1198555–1198790.

[56] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU gems* 3.39 (2007), pp. 851–876.

[57] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[58] *High Bandwidth Memory*. URL: https://en.wikipedia.org/wiki/High_Bandwidth_Memory (visited on 04/2020).

[59] Mark D Hill and Michael R Marty. "Amdahl's law in the multicore era". In: *Computer* 41.7 (2008), pp. 33–38.

[60] Ron Ho, Kenneth W Mai, and Mark A Horowitz. "The future of wires". In: *Proceedings of the IEEE* 89.4 (2001), pp. 490–504.

[61] Jared Hoberock et al. "Stream compaction for deferred shading". In: *Proceedings of the Conference on High Performance Graphics 2009*. 2009, pp. 173–180.

[62] Sungpack Hong et al. "Accelerating CUDA graph algorithms at maximum warp". In: *ACM SIGPLAN Notices*. Vol. 46. ACM. 2011, pp. 267–276.

[63] Daniel Horn. "Stream reduction operations for GPGPU applications". In: *Gpu gems* 2.36 (2005), pp. 573–589.

[64] Joel Hruska. *Intel Acknowledges It Was 'Too Aggressive' With Its 10nm Plans*. 2019. URL: https://www.extremetech.com/computing/295159-intel-acknowledges-its-long-10nm-delay-caused-by-being-too-aggressive (visited on 04/2019).

[65] Zhigang Hu et al. "Microarchitectural techniques for power gating of execution units". In: *Proceedings of the 2004 international symposium on Low power electronics and design*. 2004, pp. 32–37.

[66] *International Data Group (IDG)*. URL: https://www.idg.com/ (visited on 04/2020).

[67] D Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. "D2MA: Accelerating coarse-grained data transfer for GPUs". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 431–442.

[68] Zhihao Jia et al. "A distributed multi-gpu system for fast graph processing". In: *Proceedings of the VLDB Endowment* 11.3 (2017), pp. 297–310.

[69] Peng Jiang, Changwan Hong, and Gagan Agrawal. "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2020, pp. 376–388.

[70] S. Jun et al. "GraFBoost: Using Accelerated Flash Storage for External Graph Analytics". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 411–424. DOI: 10.1109/ISCA.2018.00042.

[71] Shinpei Kato et al. "Autoware on board: Enabling autonomous vehicles with embedded systems". In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 287–296.

[72] Paresh Kharya. *Record 136 NVIDIA GPU-Accelerated Supercomputers Feature in TOP500 Ranking*. 2019. URL: https://blogs.nvidia.com/blog/2019/11/19/record-gpu-accelerated-supercomputers-top500/ (visited on 04/2020).

[73] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. "Scalable simd-efficient graph processing on gpus". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 39–50.

[74] Farzad Khorasani et al. "CuSha: vertex-centric graph processing on GPUs". In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM. 2014, pp. 239–252.

[75] Ji Yun Kim and Christopher Batten. "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 75–87.

[76] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[77] Rakesh Komuravelli et al. "Stash: Have your scratchpad and cache it too". In: *ACM SIGARCH Computer Architecture News* 43.3S (2015), pp. 707–719.

[78] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[79] Jens Krüger and Rüdiger Westermann. "Linear algebra operators for GPU implementation of numerical algorithms". In: *ACM SIGGRAPH 2005 Courses*. 2005, 234–es.

[80] Snehasish Kumar et al. "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 376–388.

[81] Nagesh B Lakshminarayana and Hyesoon Kim. "Spare register aware prefetching for graph algorithms on GPUs". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 614–625.

[82] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[83] Jinho Lee et al. "ExtraV: Boosting graph processing near storage with a coherent accelerator". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1706–1717.

[84] Jingwen Leng et al. "GPUWattch: enabling energy optimizations in GPGPUs". In: *ACM SIGARCH Computer Architecture News*. Vol. 41. ACM. 2013, pp. 487–498.

[85] Bingchao Li et al. "Elastic-cache: GPU cache architecture for efficient fine-and coarse-grained cache-line management". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 82–91.

[86] Chao Li et al. "Locality-driven dynamic GPU cache bypassing". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, pp. 67–77.

[87] Junjie Li, Sanjay Ranka, and Sartaj Sahni. "Strassen's matrix multiplication on GPUs". In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE. 2011, pp. 157–164.

[88] Pingfan Li et al. "High performance parallel graph coloring on GPGPUs". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pp. 845–854.

[89] Shang Li et al. "DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator". In: *IEEE Computer Architecture Letters* (2020).

[90] Sheng Li et al. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 469–480.

[91] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. "Why GPUs are Slow at Executing NFAs and How to Make them Faster". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 251–265.

[92] Samuel Liu et al. *Operand collector architecture*. US Patent 7,834,881. Nov. 2010.

[93] Yucheng Low et al. "Graphlab: A new framework for parallel machine learning". In: *arXiv preprint arXiv:1408.2041* (2014).

[94] Andrew Lumsdaine et al. "Challenges in parallel graph processing". In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[95] Lingxiao Ma et al. "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication". In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 195–207.

[96] Kiran Kumar Matam et al. "GraphSSD: graph semantics aware SSD". In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 116–128.

[97] Simon McIntosh-Smith. "The gpu computing revolution". In: (2011).

[98] Duane Merrill and Michael Garland. "Single-pass parallel prefix scan with decoupled look-back". In: *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[99] Duane Merrill, Michael Garland, and Andrew Grimshaw. "High-performance and scalable GPU graph traversal". In: *ACM Transactions on Parallel Computing (TOPC)* 1.2 (2015), pp. 1–30.

[100] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[101] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[102] Gordon E Moore et al. “Progress in digital integrated electronics”. In: *Electron devices meeting*. Vol. 21. 1975, pp. 11–13.

[103] Onur Mutlu et al. “Processing data where it makes sense: Enabling in-memory computation”. In: *Microprocessors and Microsystems* 67 (2019), pp. 28–41.

[104] *Nintendo 64 technical specifications: Reality Coprocessor*. URL: https://en.wikipedia.org/wiki/Nintendo_64_technical_specifications#Reality_Coprocessor (visited on 04/2020).

[105] *Nintendo Switch: Hardware*. URL: https://en.wikipedia.org/wiki/Nintendo_Switch#Hardware (visited on 04/2020).

[106] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. “Tigr: Transforming irregular graphs for gpu-friendly graph processing”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 622–636.

[107] NVIDIA. *nvGRAPH*. URL: https://developer.nvidia.com/nvgraph (visited on 04/2020).

[108] *Nvidia Drive*. URL: https://en.wikipedia.org/wiki/Nvidia_Drive (visited on 04/2020).

[109] *NVIDIA Jetson TX1*. URL: https://en.wikipedia.org/wiki/Tegra#Tegra_X1 (visited on 04/2017).

[110] *NVIDIA RTX Voice*. 2020. URL: https://www.nvidia.com/en-us/geforce/guides/nvidia-rtx-voice-setup-guide/ (visited on 04/2020).

[111] *NVIDIA Tegra*. URL: https://en.wikipedia.org/wiki/Tegra (visited on 04/2020).

[112] Molly A O’Neil and Martin Burtscher. “Microarchitectural performance characterization of irregular GPU kernels”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 130–139.

[113] Tayo Oguntebi and Kunle Olukotun. “Graphops: A dataflow library for graph analytics acceleration”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 111–117.

[114] *OpenGL*. URL: https://en.wikipedia.org/wiki/OpenGL (visited on 04/2020).

[115] Muhammet Mustafa Ozdal et al. “Energy Efficient Architecture for Graph Analytics Accelerators”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 166–177. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.24. URL: https://doi.org/10.1109/ISCA.2016.24.

[116] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab, 1999.

[117] Yuechao Pan et al. “Multi-GPU graph analytics”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 479–490.

[118] *Parallel Thread Execution ISA*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html (visited on 04/2020).

[119] Juan C Pichel et al. “Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs”. In: *Microprocessors and Microsystems* 36.2 (2012), pp. 65–77.

[120] *PlayStation technical specifications: Graphics processing unit (GPU)*. URL: https://en.wikipedia.org/wiki/PlayStation_technical_specifications#Graphics_processing_unit_(GPU) (visited on 04/2020).

[121] *PyTorch*. URL: https://pytorch.org/ (visited on 04/2020).

[122] *Radeon HD 2000 series*. URL: https://en.wikipedia.org/wiki/Radeon_HD_2000_series (visited on 04/2020).

[123] Milan Radulovic et al. “Another trip to the wall: How much will stacked dram benefit hpc?” In: *Proceedings of the 2015 International Symposium on Memory Systems*. 2015, pp. 31–36.

[124] David Reinsel, John Gantz, and John Rydning. “The digitization of the world from edge to core”. In: *IDC White Paper* (2018).

[125] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[126] Minsoo Rhu et al. “A locality-aware memory hierarchy for energy-efficient GPU architectures”. In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2013, pp. 86–98.

[127] Cody Rivera et al. “ISM2: Optimizing Irregular-Shaped Matrix-Matrix Multiplication on GPUs”. In: *arXiv preprint arXiv:2002.03258* (2020).

[128] David Roger, Ulf Assarsson, and Nicolas Holzschuch. “Efficient stream reduction on the GPU”. In: 2007.

[129] Christopher Root and Todd Mostak. “MapD: a GPU-powered big data analytics and visualization platform”. In: *ACM SIGGRAPH 2016 Talks*. 2016, pp. 1–2.

[130] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. “DRAMSim2: A cycle accurate memory system simulator”. In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[131] Karl Rupp. *Years of Microprocessor Trend Data, 2018*. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ (visited on 04/2020).

[132] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. “Subway: minimizing data transfer during out-of-GPU-memory graph processing”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[133] Fabien Sanglard. *A history of NVidia Stream Multiprocessor*. 2020. URL: https://fabiensanglard.net/cuda/index.html (visited on 06/2020).

[134] Franco Scarselli et al. “The graph neural network model”. In: *IEEE Transactions on Neural Networks* 20.1 (2008), pp. 61–80.

[135] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[136] Albert Segura Salvador. “Characterization of Speech Recognition Systems on GPU Architectures”. MA thesis. Universitat Politècnica de Catalunya, 2016.

[137] Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. *Irregular Accesses Reorder Unit: Improving GPGPU Memory Coalescing for Graph-Based Workloads*. 2020. arXiv: 2007.07131 [cs.AR].

[138] Albert Segura, Jose-Maria Arnau, and Antonio González. "Energy-Efficient Stream Compaction Through Filtering and Coalescing Accesses in GPGPU Memory Partitions". In: *Submitted and under reviewing process* (2020).

[139] Albert Segura, Jose-Maria Arnau, and Antonio González. "SCU: A GPU Stream Compaction Unit for Graph Processing". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 424–435. ISBN: 9781450366694. DOI: 10.1145/3307650.3322254. URL: https://doi.org/10.1145/3307650.3322254.

[140] Dipanjan Sengupta et al. "GraphReduce: processing large-scale graphs on accelerator-based systems". In: *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2015, pp. 1–12.

[141] Shubhabrata Sengupta, Mark Harris, and Michael Garland. "Efficient parallel scan algorithms for GPUs". In: *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003* 1.1 (2008), pp. 1–17.

[142] Shubhabrata Sengupta et al. "Scan primitives for GPU computing". In: (2007).

[143] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[144] Xuanhua Shi et al. "Graph processing on GPUs: A survey". In: *ACM Computing Surveys (CSUR)* 50.6 (2018), pp. 1–35.

[145] Xuanhua Shi et al. "Optimization of asynchronous graph processing on GPU with hybrid coloring model". In: *ACM SIGPLAN Notices* 50.8 (2015), pp. 271–272.

[146] Seunghee Shin et al. "Neighborhood-aware address translation for irregular GPU applications". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 352–363.

[147] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. "Out-of-core GPU memory management for MapReduce-based large-scale graph processing". In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2014, pp. 221–229.

[148] Yogesh Simmhan et al. "Goffish: A sub-graph centric framework for large-scale graph analytics". In: *European Conference on Parallel Processing*. Springer. 2014, pp. 451–462.

[149] L. Song et al. "GraphR: Accelerating Graph Processing Using ReRAM". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 531–543. DOI: 10.1109/HPCA.2018.00052.

[150] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[151] Micron Technology. *TN-53-01. LPDDR4 Power Calculator*. Technical Report, 2016.

[152] *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 04/2020).

[153] *Tesla (microarchitecture)*. URL: https://en.wikipedia.org/wiki/Tesla_(microarchitecture) (visited on 04/2020).

[154] *Tesla Model S*. URL: https://en.wikipedia.org/wiki/Tesla_Model_S (visited on 04/2020).

[155] Thomas N Theis and H-S Philip Wong. "The end of moore's law: A new beginning for information technology". In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50.

[156] *Ultra-high-definition television*. URL: https://en.wikipedia.org/wiki/Ultra-high-definition_television (visited on 04/2020).

[157] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. "Automatic restructuring of GPU kernels for exploiting inter-thread data locality". In: *International Conference on Compiler Construction*. Springer. 2012, pp. 21–40.

[158] *Virtual reality*. URL: https://en.wikipedia.org/wiki/Virtual_reality (visited on 04/2020).

[159] Dani Voitsechov and Yoav Etsion. "Single-Graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 205–216. ISBN: 9781479943944.

[160] Bin Wang et al. "Dacache: Memory divergence-aware gpu cache management". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, pp. 89–98.

[161] Yangzihao Wang et al. "Gunrock: A high-performance graph processing library on the GPU". In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016, pp. 1–12.

[162] Bo Wu et al. "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu". In: *ACM SIGPLAN Notices* 48.8 (2013), pp. 57–68.

[163] Qing Wu, Massound Pedram, and Xunwei Wu. "Clock-gating and its application to low power design of sequential circuits". In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47.3 (2000), pp. 415–420.

[164] Yuduo Wu et al. "Performance characterization of high-level programming models for GPU graph analytics". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 66–75.

[165] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[166] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. "Graph processing on GPUs: Where are the bottlenecks?" In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 140–149.

[167] Mingyu Yan et al. "Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 615–628.

[168] Mingyu Yan et al. "Characterizing and Understanding GCNs on GPU". In: *IEEE Computer Architecture Letters* 19.1 (2020), pp. 22–25.

[169] Shengen Yan, Guoping Long, and Yunquan Zhang. "StreamScan: fast scan algorithms for GPUs without global barrier synchronization". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 229–238.

[170] Haoduo Yang et al. "HPGA: A High-Performance Graph Analytics Framework on the GPU". In: *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*. IEEE. 2018, pp. 488–492.

[171] Pengcheng Yao et al. "An efficient graph accelerator with parallel data conflict management". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–12.

[172] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[173] Eddy Z Zhang et al. "On-the-fly elimination of dynamic irregularities for GPU computing". In: *ACM SIGPLAN Notices* 46.3 (2011), pp. 369–380.

[174] Eddy Z Zhang et al. "Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. 2010, pp. 115–126.

[175] Jianlong Zhong and Bingsheng He. "Medusa: Simplified graph processing on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2013), pp. 1543–1552.

[176] Jianlong Zhong and Bingsheng He. "Towards GPU-accelerated large-scale graph processing in the cloud". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE. 2013, pp. 9–16.

[177] Jinhong Zhou et al. "Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2017, pp. 731–734.

[178] Youwei Zhuo et al. "GraphQ: Scalable PIM-Based Graph Processing". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 712–725.