ACCELERATOR DESIGN FOR GRAPH ANALYTICS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Şerif Yeşil

June 2016

Accelerator Design For Graph Analytics
By Şerif Yeşil
June 2016

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Muhammet Mustafa Özdal

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Levent Onural
Director of the Graduate School

ABSTRACT

ACCELERATOR DESIGN FOR GRAPH ANALYTICS

Şerif Yeşil
M.S. in Computer Engineering
Advisor: Özcan Öztürk
June 2016

With the increase in data available online, data analysis became a significant problem in today's datacenters. Moreover, graph analytics is one of the significant application domains in big data era. However, traditional architectures such as CPUs and Graphics Processing Units (GPUs) fail to serve the needs of graph applications. Unconventional properties of graph applications such as irregular memory accesses, load balancing, and irregular computation challenge current computing systems which are either throughput oriented or built on top of traditional locality based memory subsystems.

On the other hand, an emerging technique hardware customization, can help us to overcome these problems since they are expected to be energy efficient. Considering the power wall, hardware customization becomes more desirable.

In this dissertation, we propose a hardware accelerator framework that is capable of handling irregular, vertex centric, and asynchronous graph applications. Developed high level SystemC models gives an abstraction to the programmer allowing to implement the hardware without extensive knowledge about the underlying architecture. With the given template, programmers are not limited to a single application since they can develop any graph application as long as it fits to the given template abstract. Besides the ability to develop different applications, the given template also decreases the time spent on developing and testing different accelerators. Additionally, an extensive experimental study shows that the proposed template can outperform a high-end 24 core CPU system up to 3x with up to 65x power efficiency.

Keywords: graph analytics, accelerators, high level synthesis.

ÖZET

ÇİZGE ANALİTİĞİ İÇİN HIZLANDIRICI TASARIMI

Şerif Yeşil
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Haziran 2016

Çevrimiçi kullanılabilir verideki artışla birlikte veri analizi bugünün veri merkezlerinde çözülmesi gereken önemli bir problem haline gelmiştir. Ayrıca çizge çözümleme uygulamaları, içinde bulunduğumuz büyük veri çağında önemli uygulamalardan biridir. Bununla birlikte, merkezi işlem birimi (CPU) ve grafik işlemcileri (GPU) gibi geleneksel mimariler çizge uygulamalarının ihtiyaçlarını karşılamakta yetersiz kalmaktadırlar. Çizge uygulamalarının düzensiz bellek erişimi, dengesiz yük dağıtımı ve düzensiz hesaplama gibi özellikler taşıması üretilen iş odaklı ya da yerellik bazlı bellek sistemleri üzerine kurulu olan mevcut hesaplama sistemlerini zorlamaktadır.

Öte yandan gelişmekte olan donanım özelleştirme teknikleri yukarıda bahsi geçen problemlerin çözülmesinde yardımcı olmakta ve bu çözümlerin enerji tasarruflu olması beklenmektedir.

Bu tezde; düzensiz, düğüm merkezli (vertex centric) ve eşzamansız (asynchronous) çizge uygulamalarının üstesinden gelebilecek bir donanım hızlandırıcı taslağı önerilmektedir. Gelişmiş yüksek seviyeli SystemC modellerinin programcıya soyut bir arayüz vermesiyle programcının arka plandaki mimari hakkında ayrıntılı bilgiye sahip olmadan donanımı gerçekleştirmesi hedeflenmektedir. Verilen taslak sayesinde programcı tek bir uygulamaya bağımlı olmaktan kurtulur ve bu soyut taslağa uyduğu sürece herhangi bir çizge uygulamasını geliştirebilir. Bunun yanında, verilen şablon kullanılarak farklı uygulamalar geliştirmeye olanak sağlanması, bu uygulamaları geliştirmek ve denemek için harcanan süreyi kısaltır. Buna ek olarak, kapsamlı deneysel çalışmalar sonucunda, önerilen taslağın son teknolojiye sahip 24 çekirdekli CPU sistemlerinden 3 kata kadar daha hızlı ve 65 kata kadar daha güç tasarruflu olduğu gözlenmiştir.

Anahtar sözcükler: çizge uygulamaları, hızlandırıcılar, yüksek seviye sentezleme.

Acknowledgement

First of all, I am grateful for his support and mentoring of my advisor Özcan Öztürk throughout my B.S. and M.S. studies .

I am also thankful to Power, Performance and Test group at Intel Corporation's Strategic CAD Labs and its members Steven M. Burns, Mustafa Özdal, Andrey Ayupov, and Taemin Kim for their great work, the internship opportunity, and mentoring.

I also would like to thank members of my thesis committee, Mustafa Özdal and Süleyman Tosun, for their feedback and interest on this topic.

Moreover, I am thankful to my team mates in Parallel Systems and Architecture Lab at Bilkent University: Funda Atik, Naveed Ul Mustafa, Hakan Taşan, Erdem Derebaşoğlu, Hamzeh Ahangari, Mohammed Reza Soltaniyeh.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for providing financial assistance during my M.S. studies via BIDEB-2210 and ARDEB "Reliability-Aware Network on Chip (NoC) Architecture Design" project (112E360).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shrinking transistor/semiconductor sizes allows us to pack more logic in a chip, but only a part of a chip can be powered up at a given time due to power limitations. This phenomena is called as the dark silicon. Esmaeilzadeh et al. showed that in the next ten years, 50% of the chip will be unpowered for technologies below 10nm [1].

Even though dark silicon seems to be a limitation, an emerging technique, custom hardware accelerators may help us design more energy efficient chips. It is known that application specific accelerators can provide significantly better performance with very small energy.

Additionally, customization in hardware is widely encouraged due to the fact that many cloud computing frameworks execute the same set of workloads repeatedly. Thereby, significant energy and performance gains can be observed thanks to customized hardware in server farms.

Previous studies mainly focus on acceleration of compute intensive workloads using traditional architectures such as central processing units (CPUs), graphics processing unit (GPUs), and SIMD vector extensions (i.e. SSE, AVX) or developing application specific hardware. However, these applications have regular computation patterns and high data-level parallelism. In spite of aforementioned applications, we focus on certain class of graph applications which shows irregular execution and memory access patterns.

Mainly, our focus is on iterative graph-parallel applications with asynchronous execution and asymmetric convergence. It is shown that many graph-parallel applications exhibit those execution patterns [2]. To exploit the efficiency by using these execution patterns, authors in [2] proposed a vertex-centric abstraction model and a software framework called GraphLab. GraphLab allows domain experts to develop parallel and distributed graph applications easily. While programmers implement user-level data structures and serial operations per vertex, the given framework overcomes the complexities of the parallel distributed computing such as scheduling of tasks, synchronization, and communication between compute nodes.

Our objective in this work is similar. However, instead of proposing a software framework for existing platforms, we provide a template that can be used to generate customized hardware. The proposed architecture template in this work specifically targets aforementioned graph applications, which shows irregular execution and memory access patterns. While common operations (i.e. memory access, synchronization, and communication) are implemented in the template, the designers can plug in application specific data structures and operations to generate custom hardware easily. This permits fast exploration and implementation of custom hardware accelerators.

Our main contributions can be summarized as follows:

- We analyse work efficiency that can be gained thanks to the characteristics of graph applications.
- We propose an architecture and a template to generate application specific hardware for vertex-centric, iterative, graph-parallel applications with irregular execution and memory access patterns. Proposed architecture is optimized to support asynchronous execution, asymmetric convergence, and load balancing. Additionally, we provide cycle-accurate and synthesizable SystemC models in which a user can plug in application level data structures and operations to generate custom hardware accelerators for graph applications.

- We provide an experimental study that compares the area, power, and performance of the generated hardware accelerators with CPU implementations. Our area and power values are obtained through physical-aware RTL synthesis of the functional blocks using industrial 22nm libraries.

- We provide an automated design space exploration methodology which enables programmers to optimize different micro-architectural parameters without detailed knowledge of the underlying hardware.

The rest of this thesis is organized as follows. Chapter 2 presents previous work. Chapter 3 gives the detailed background information and discusses the properties of graph applications. Moreover, we provide an experimental study to quantify the effect of these properties in Section 3.1. In Chapter 4, we describe the abstraction model and the proposed hardware. Experimental results are given in Chapter 5. Finally, Chapter 6 concludes this thesis.

Chapter 2

Related Work

In this chapter, we summarize the existing literature on acceleration of graph analytics applications. Firstly, we explain the solutions proposed to address challenges of graph applications at software stack such as implementations of graph applications on current parallel systems and proposed parallel and distributed software frameworks. Secondly, we discuss the solutions proposed at hardware stack including extensions to the current systems and custom hardware implementations of graph applications.

2.1 Software Solutions

Previous studies at software stack can be divided into three categories: (1) Design of parallel and distributed processing frameworks. (2) Implementation of graph applications for existing platforms such as CPUs, GPUs, and MIC. (3) Analysis of performance bottlenecks of existing platforms for irregular graph applications.

The most famous example of distributed graph processing frameworks is Google's Pregel [3]. Pregel suggests a bulk synchronous environment which avoids the usage of locks and focuses on very large scale computing. On the other hand, GraphLab [2] focuses on asynchronous computations and benefits from asymmetric convergence and asynchronous execution characteristics of graph applications. While Pregel and GraphLab provide high level abstractions for graph applications, Galois [4] gives programming constructs which are close to the native implementations and thus, shows better performance compared to the others [5]. Moreover, Green-Marl [6] is a domain specific language that is designed for parallel graph applications. Other examples of software solutions for graph applications are Giraph [7], CombBLAS [8], and SociLite [9]. While all of these are optimized for graph parallel applications, they are purely software-based systems. Our approach can be extended to support any of these frameworks.

Secondly, there have been efforts on accelerating graph applications on existing accelerator platforms. In [10], the authors propose a warp centric execution model to avoid control divergence and work imbalance in irregular graph applications. Additionally, Medusa [11] is a processing framework which focuses on bulk synchronous processing and targeted for GPUs. They also consider multi GPU acceleration and optimize graph partitioning to reduce the communication between GPUs. On the other hand, [12] adapts vertex centric and message passing execution for CPU and MIC. Furthermore, there are several attempts to implement graph applications, which are used in this thesis, efficiently on existing platforms. For example, [13, 14] provides GPU implementations for PageRank application whereas, [15] discusses edge centric and vertex centric implementations for stochastic gradient descent (SGD) on GPUs. Additionally, [16] presents GPU implementations of single source shortest path (SSSP) application.

While GPU implementations of irregular graph applications are widely studied, there are several problems that GPUs encounter when it comes to irregular applications. Previous studies analysed these bottlenecks such as synchronization problems, effect of irregular memory accesses, and control divergence in detail[17, 18, 19].

In particular, [17] and [18] provide more insight about characteristics of irregular applications. First of all, [17] discusses synchronization problems. GPUs, such as NVIDIA's Tesla series, do not have global synchronization mechanisms. Specifically, kernel launches are used as barriers, and data is transferred back to the CPU in order to decide convergence. This causes more CPU-GPU communication and many kernel invocations. In [17], authors state that irregular graph applications have 20x more interaction with CPU compared to regular applications.

Secondly, GPUs need to run 1000s of threads in parallel in order to utilize available memory bandwidth, this situation makes use of locks nearly impossible in a vertex-centric execution environment.

Thirdly, graph applications are not able to exploit faster memories on GPU systems, such as shared memories, due to the irregular structure of data accesses and lack of data reuse. Additionally, memory access irregularities cause bank conflicts and higher memory latency for the accesses from global memory.

It is reported that CPUs can outperform GPU in spite the fact that GPUs have 1000s of threads running in parallel. For example [15] reports that a GPU implementation performs as good as 14 cores on a 40 core CPU system for SGD application. Similar results can also be found for SSSP in [16]. This work states that even an efficient serial implementation of SSSP can outperform parallel implementations for high diameter graphs and also in cases of scale free graphs where several nodes have very high degrees, highly parallel implementations become inefficient. Another work on PageRank [13] describes a Sparse Matrix Vector Multiplication (SpMV) based method. Moreover, authors in [14] report 5x speed up compared to 4 core system for a vertex centric PageRank GPU implementation. However, both studies in [13] and [14] ignore the asynchronous execution behaviour of PageRank and are unable to exploit opportunities presented in Section 3.1.2.3.

2.2 Hardware Solutions

In addition to the software solutions, graph applications are also studied in hardware domain. Both template based approaches and application specific implementations are available for FPGAs and ASICs. Moreover, there are a few studies which try to tailor existing platforms for the needs of graph applications.

PageRank is one of the most interesting applications in graph analytics domain. Although number of GPU and multi-core CPU implementations is large for PageRank, application specific hardware design suggestions are limited. An FPGA implementation of PageRank is given in [20]. In this work, authors propose an edge streaming architecture. In addition to the edge streaming, proposed system avoids multiplications and non-zero elements in the graph and, thus, makes the computation more efficient.

In addition to the PageRank, breadth-first-search and single source shortest path problems are also widely studied [21, 22, 23, 24, 25]. The work in [21] proposes a compressed sparse row matrix (CSR) representation where vertex data is used interchangeably for converged and unconverged vertices to decrease the number of memory accesses. Secondly, authors in [22] designed a message passing and synchronous environment for write-based BFS using a multi-softcore design, whereas the work in [23] tries to accelerate a read-based BFS implementation. Additionally, [24] uses dense and sparse BFS implementation interchangeably to accelerate BFS on a heterogeneous CPU-FPGA platform.

Furthermore, there are proposals to enhance the performance of graph applications on existing platforms by introducing new hardware constructs. One of these approaches [26] tries to implement a hardware work-list that would make data driven executions for irregular applications feasible on GPGPUs. Recently, another study proposed a PIM (processing in memory) [27] based system which exploits available memory bandwidth in 3D stacked memories.

On the other hand, GraphGen [28] is a framework to create an application specific synthesized graph processor and memory layout for FPGAs. GraphGen also uses a vertex centric execution model to represent graph applications. However, it is targeted towards regular applications and cannot handle irregular applications such as PageRank. Moreover, GraphStep [29] implements a bulk synchronous message passing execution model on FPGAs for graph applications. These two examples are the closest proposals to our work.

To the best of our knowledge, our work is the first accelerator architecture that specifically targets asynchronous, iterative, vertex-centric graph applications with irregular access patterns and asymmetric convergence.

Chapter 3

Analysis of Graph Applications

In this chapter, we discuss the properties of graph applications and vertex centric programming model. Moreover, we discuss the shortcomings of existing systems such as CPUs and GPUs.

3.1 Irregular Graph Applications

The characteristics of computation and memory accesses of the graph applications are widely studied [2, 30, 31]. These previous studies either address the shortcomings of existing architectures or the common problems on traditional platforms. However, in this thesis, we discuss how different properties of graph applications affect the work efficiency and propose templatized hardware mechanisms to exploit the work efficiency.

3.1.1 Vertex Centric Programming Model

”Think like a vertex” is a concept which is introduced by Google’s Pregel [3] framework. In vertex centric execution model, the user program is divided into single vertex programs which takes a vertex, its incident edges, and neighbour vertices as input. In vertex centric execution model, each vertex has an associated vertex data (VD) which represents the data to compute, and vertex information (VI) which has the information for incoming and/or outgoing edges of the vertex. Beside the vertex data and the vertex information structures, each edge can have edge data (ED), to be used for computations, and an associated edge information (EI), for keeping source and/or destination vertices of the edge.

Vertex centric execution model is widely studied but each proposed software framework adapts slightly different execution mechanisms. For example, Pregel supports synchronous execution while GraphLab [2] focusses on asynchronous computation. Additionally, edge access primitives differ such that Pregel provides explicit edge iterators, whereas GraphLab provides *Gather* and *Scatter* to visit incident edges of a vertex. However, in all of these models, execution of a single vertex program can be summarized as follows:

- Each vertex has an iterator for iterating through its incoming and/or outgoing edges.
- First a vertex program executes an accumulation function which will process all incident edges and neighbour vertices of the input vertex.
- Each vertex program can update the input vertex's VD and incident edges ED.
- Finally, a vertex program can activate (send a message to) its neighbour vertices for the next iteration if it is not converged yet.

3.1.2 Properties of Graph Applications

3.1.2.1 Case Study - PageRank

In the following sections, we will use PageRank [32] application as a case study.

High level details of PageRank application are outlined in Figure 3.1. Simply, PageRank takes a web-graph as input and calculates relative importance of web pages

```
PageRank(Input graph: (V, E))
1.    for each unconverged vertex $v \in V$ do:
2.        $sum = 0$
3.        for each vertex $u$ for which $(u \rightarrow v) \in E$
4.            $sum = sum + \frac{r_u}{d_u}$
5.        $r_v^{new} = \frac{(1-\beta)}{|V|} + \beta \cdot sum$
6.        activateNeig = $|r_v^{new} - r_v| > \varepsilon$
7.        $r_v = r_v^{new}$
8.        if activateNeig then
9.            for each vertex $w$ for which $(v \rightarrow w) \in E$
10.               activate $w$
```

Figure 3.1: Pseudo-code of the PageRank algorithm.

according to the importance of its neighbours. Each vertex first calculates the sum of scaled rank (r_u/d_u where d_u is the out degree of the vertex u and r_u is the rank of vertex u) of its neighbours. Then, sum is multiplied with a damping factor β and new rank value of a vertex v is assigned to $\frac{(1-\beta)}{|V|} + \beta \cdot sum$. If change in rank value of a vertex v, $|r_v^{new} - r_v|$, is above a certain threshold ε, v's neighbours are activated. Otherwise, it is considered to be converged.

3.1.2.2 Asymmetric Convergence

In many graph applications, the application is executed until a convergence criteria is satisfied. Delta change observed for a vertex in PageRank application can be considered as an example of a convergence. Until the convergence criteria is met, graph data is updated iteratively. A simple way to implement this behaviour is to execute all vertices simultaneously until the convergence criteria is satisfied. However, previous studies [2, 31] showed that most of the vertices may converge in earlier iterations of the execution and they do not need to be executed in every iteration. This property is called *asymmetric convergence*. Specifically, asymmetric convergence allows us to decrease the number of vertices processed in each iteration. Therefore, it may lead to a faster execution.

To see the effect of asymmetric convergence, we experimented with a well known graph application, PageRank. More specifically, the traditional PageRank [32] application is extended to support asymmetric convergence by adding a bit vector implementation. Our bit vector implementation works as follows: (1) We keep two bit vectors bv_current for checking whether a vertex is active for the current iteration and bv_next for storing activations for the next iteration. (2) Whenever a vertex needs to activate its neighbours, it makes the bits of its incident vertices 1 in bv_next. (3) At each iteration, we swap the bv_current and bv_next, and reset the bits in bv_next to 0.

First, we have examined the change in the number of active (non-converged) vertices for PageRank application with three real world datasets; web-Google (wg), soc-Pokec (pk), and soc-LivJournal (lj); from SNAP graph database [33]. Our results in Figure 3.2 shows that for pk and lj datasets only 36 iterations are needed for 50% of vertices to converge. On the other hand, 50% of vertices are converged in 24 iterations for wg dataset. After the 50^{th} iteration, we observed that less than 1% of the vertices are active for any of the datasets, although the number of iterations needed for convergence can go up to 92 for some of the datasets. It is obvious that it is not efficient to execute all vertices in every iteration.

Furthermore, we have inspected the effect of asymmetric convergence on work efficiency. Figure 3.3 shows that enabling asymmetric convergence in PageRank application yields 47% better work efficiency in terms of the number of multiply-add operations performed.

We observed that implementing a graph application without asymmetric convergence on throughput oriented systems such as GPUs, which require 1000s of threads to utilize available processing cores, is easy but not work efficient. On the other hand, enabling asymmetric convergence changes number of active vertices significantly during the execution. This situation causes warps to have control divergence and leads underutilization of GPU threads. A solution to this would be implementing an active list. However, it is also hard to implement active lists on GPUs since GPUs do not have dynamic memory management.

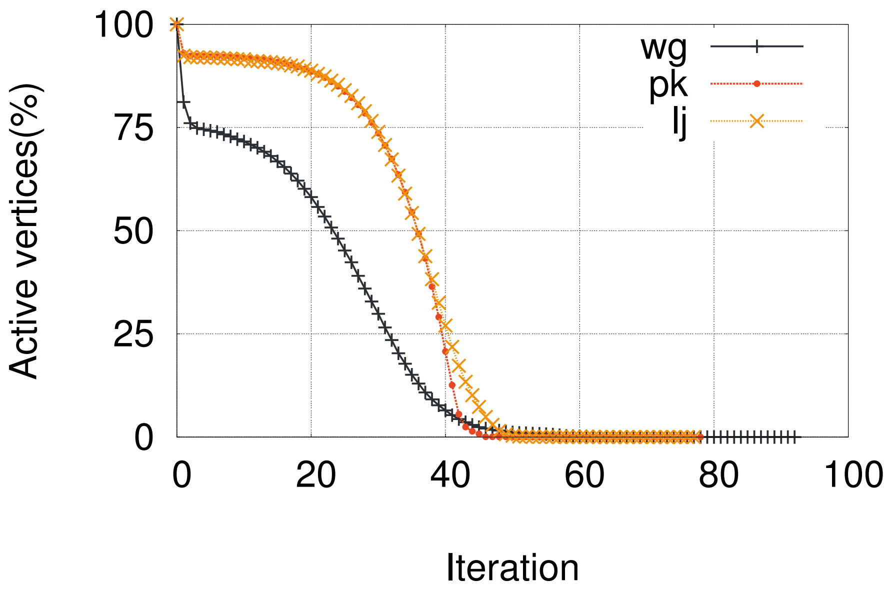


Figure 3.2: Changes in the number of active vertices per iteration for wg, pk, and lj datasets.

3.1.2.3 Asynchronous Execution

Synchronous processing model defines clear iterations as in Pregel's super-steps. Both computations and memory accesses are separated into super-steps. Specifically, when a vertex v needs to access its neighbour u's vertex data in iteration $k+1$, v will access u's vertex data in iteration k. On the other hand, *asynchronous execution* does not have a well defined iteration concept. Updates on vertex data are immediately available to the neighbours.

It is shown that asynchronous execution converges faster than synchronous execution but it also creates more challenges [2, 31]. In synchronous model, there are no data hazards (read-after-write) since memory accesses are also separated by barriers along with two copies of the vertex data. On the contrary, asynchronous execution keeps a single copy of the vertex data and asynchronous parallel execution is susceptible to the data hazards. In order to ensure correctness, we need to enforce sequential consistency[1].

[1]Sequential Consistency: A parallel execution is correct if and only if its execution order corresponds to some sequential order.

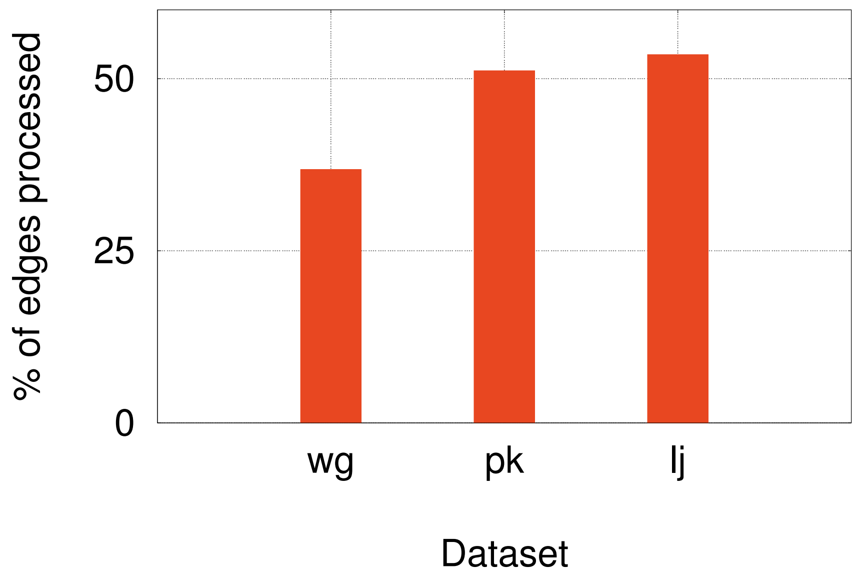


Figure 3.3: Work efficiency when asymmetric convergence is enabled for wg, pk, and lj datasets normalized with respect to the baseline PageRank implementation in which all vertices are executed in every iteration until convergence (lower values are better).

In spite of work efficiency, asynchronous execution may run slower since fine grain synchronization is required for providing sequential consistency.

Let us consider two different implementations of PageRank, namely, Jacobi method and Gauss-Seidel method. Jacobi method follows synchronous execution model while Gauss-Seidel follows asynchronous execution model. We can express Jacobi method as follows:

$$r_v^{k+1} = \frac{(1-\beta)}{|V|} + \beta \sum_{(u \to v) \in E} \frac{r_u^k}{d_u}, \tag{3.1}$$

where r_u^k represents the page rank value computed for vertex u in the k^{th} iteration, and d_u is the out-degree of vertex u. Note that, in calculation of r_v^{k+1} we use r_u^k which is from the previous iteration. Alternatively, Gauss-Seidel iteration has the following formula:

$$r_v^{k+1} = \frac{(1-\beta)}{|V|} + \beta \sum_{\substack{u<v \\ (u \to v) \in E}} \frac{r_u^{k+1}}{d_u} + \sum_{\substack{u>v \\ (u \to v) \in E}} \frac{r_u^k}{d_u} \tag{3.2}$$

Specifically, when vertex u is processed in iteration $k+1$, it uses the rank values of iteration $k+1$ for the vertices before it and the rank values of iteration k for the ones after. It was shown in [34] that the Gauss-Seidel formulation can converge by about 2x faster than the Jacobi formulation.

To test the effect of asynchronous execution on work efficiency, we have implemented two different serial PageRank models using equations 3.1 and 3.2. Figure 3.4 summarizes our results for three datasets. We observe that asynchronous execution provides 63% better work efficiency on average.

Moreover, we further analyse the work efficiency of PageRank application when both asymmetric convergence and asynchronous execution are enabled. This implementation also strictly follows serial execution order and uses a similar bit-vector implementation as discussed in Section 3.1.2.2. Additionally, bit-vector implementation avoids duplicate activations. In this case, the average work efficiency observed increases to 31.1% as can be seen in Figure 3.5.

Multi-core CPU architectures incur synchronization overheads in the asynchronous mode of execution. For example, it has been shown that the GraphLab implementation of PageRank slows down by more than an order of magnitude on a multi-core system when sequential consistency property is enabled [31].

Furthermore, GPUs lack good synchronization and efficient atomic access mechanisms which are required by graph applications. GPUs also lack efficient global synchronization mechanisms and they might require separate kernel invocations for barriers. Thus, asynchronous execution is not suitable for GPUs since it requires expensive lock mechanisms. For this reason many GPU implementations choose to implement synchronous model even if it is not work efficient. Yet, another study shows that even synchronous model incurs overheads due to the amount of interaction between host and GPU because of multiple kernel calls [17].

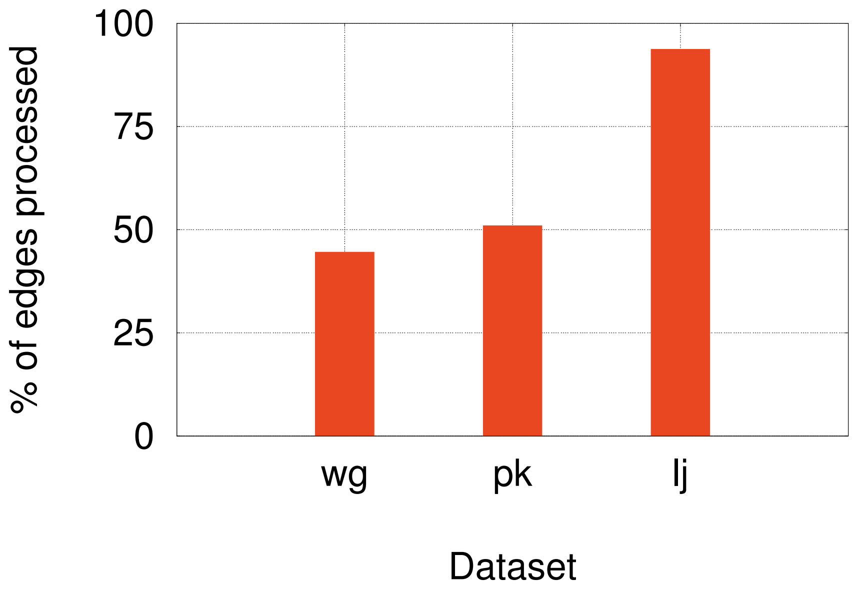


Figure 3.4: Work efficiency when asynchronous execution is enabled for wg, pk, and lj datasets normalized with respect to the baseline PageRank implementation in which all vertices are executed in every iteration until convergence (lower values are better).

3.1.2.4 Load Imbalance

Degree distribution of real graphs changes greatly since these graphs follow a power-law distribution. As an example, we can consider a social network graph where edges represent friendship or follower relations. While most of the vertices have 10s of edges (friends or followers), there will be a few vertices which have millions of edges (friends or followers) such as politicians or celebrities.

Our selected datasets also show similar characteristics. Figures 3.6, 3.7, and 3.8 show that majority of vertices have less than 200 edges when we consider incoming and outgoing edges of vertices. Moreover, it is clear that number of vertices with high degrees are limited.

One can consider a parallel implementation where vertices of a graph are assigned to different threads due to power-law distribution of vertex degrees. More specifically, it is possible that some threads will process significantly more edges than the others (multiply-add operations in PageRank example). For this reason assigning vertices to

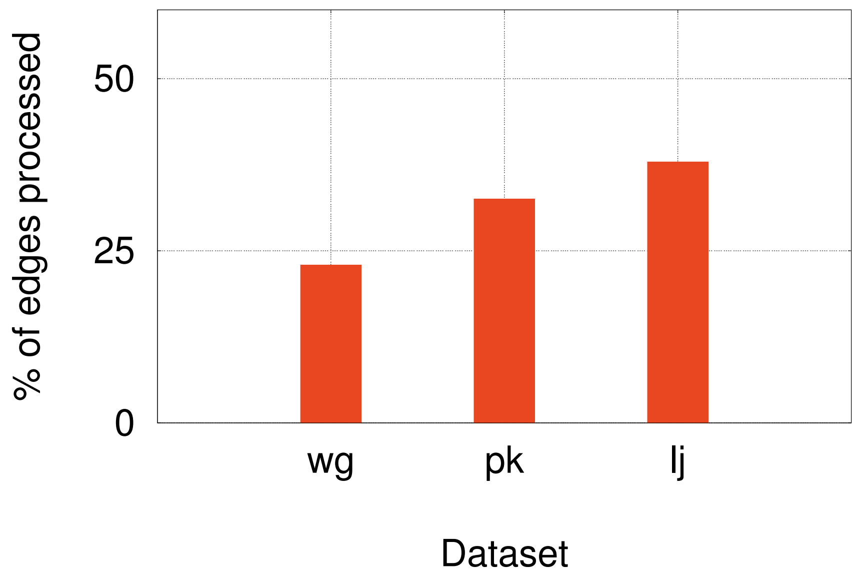


Figure 3.5: Work efficiency when both asymmetric convergence and asynchronous execution are enabled for wg, pk, and lj datasets normalized with respect to baseline PageRank implementation in which all vertices are executed in every iteration until convergence (lower values are better).

processors at compile time may lead to load imbalances. Software based dynamic load balancing techniques are deployed to address this problem.

Software dynamic load balancing, typically resolve this issue effectively for multi-core processors. In contrast, throughput oriented SIMD architectures, such as GPUs, a vertex centric graph application with static vertex assignment to threads will suffer from load imbalances. A study shows that the warp utilization is less than 25% for GPUs when executing graph applications [17].

3.1.2.5 Memory Bottlenecks

In graph applications, it is common to have low compute to memory ratio per vertex program. Moreover, underlying graph structure has large number of vertices and edges to process for large graphs. This situation causes poor temporal locality. As a result, number of memory requests increase and the application suffers from long memory access latencies.

An analytical model proposed in [35] shows that the number of non-continuous memory accesses dominate the runtime ($T(n,p)$) in shared memory machines.

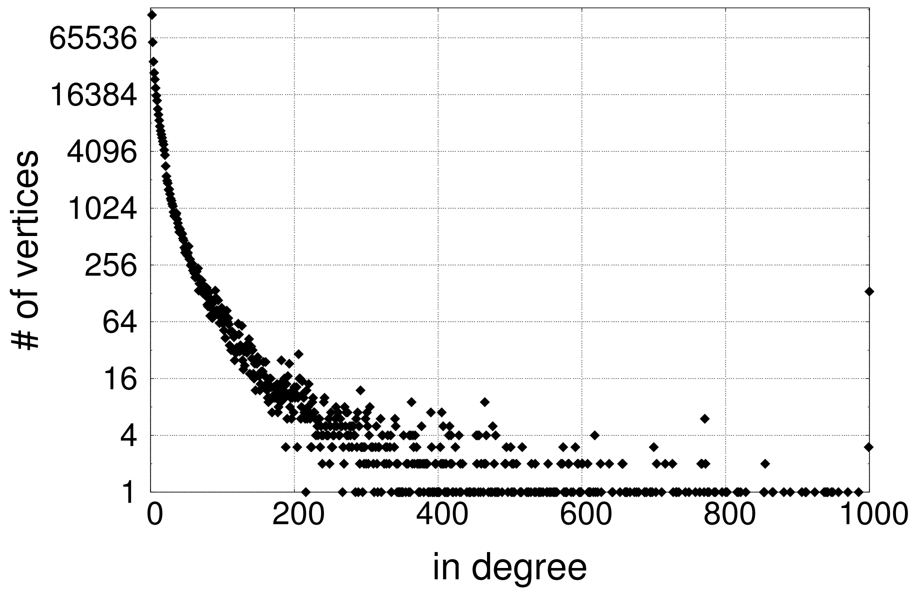
$$T(n,p) = \max\{T_M(n,p), T_C(n,p), B(n,p)\}. \tag{3.3}$$

As shown in Equation 3.3, the runtime of a parallel program depends on maximum number of non-continuous memory accesses per process ($T_M(n,p)$), maximum computation cost per process ($T_C(n,p)$), or cost of barriers ($B(n,p)$). An analysis on the PageRank algorithm given in Figure 3.1 shows that there will be a non-continuous memory access and a multiply-add operation for each neighbour vertex data (lines 3-4). If memory access latency is high, then, for an in-order shared memory machine, we would expect memory access time to dominate the runtime.

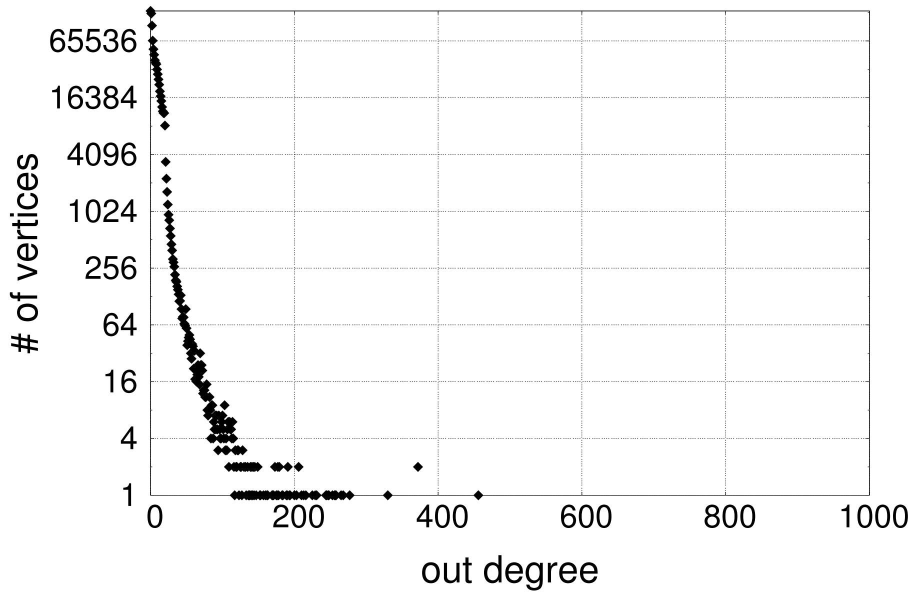
A recent study reports the following observations for the performance of graph applications [36]: (1) Instruction window should be available and a load operation should be able to be scheduled. (2) There should be available registers. (3) Memory bandwidth should be available.

Authors in the same study conclude that (1) memory latency is the main performance bottleneck, (2) low memory level parallelism (MLP) leads to under-utilization of the DRAM bandwidth, and (3) overall performance generally scales linearly with memory bandwidth consumption because of overlapped access latencies. Basically, this study suggests that graph applications are not able to utilize all MSHR entries on core due to window size limitations. They have also found that MLP is not achievable with many number of cores. Moreover, parallel implementations of graph applications reach their peak performance with small number of cores (2-4 cores usually).

Nonetheless, increasing number of simultaneous threads lead to a higher memory bandwidth utilization for graph applications. However, with many cores running in parallel will sacrifice energy efficiency.

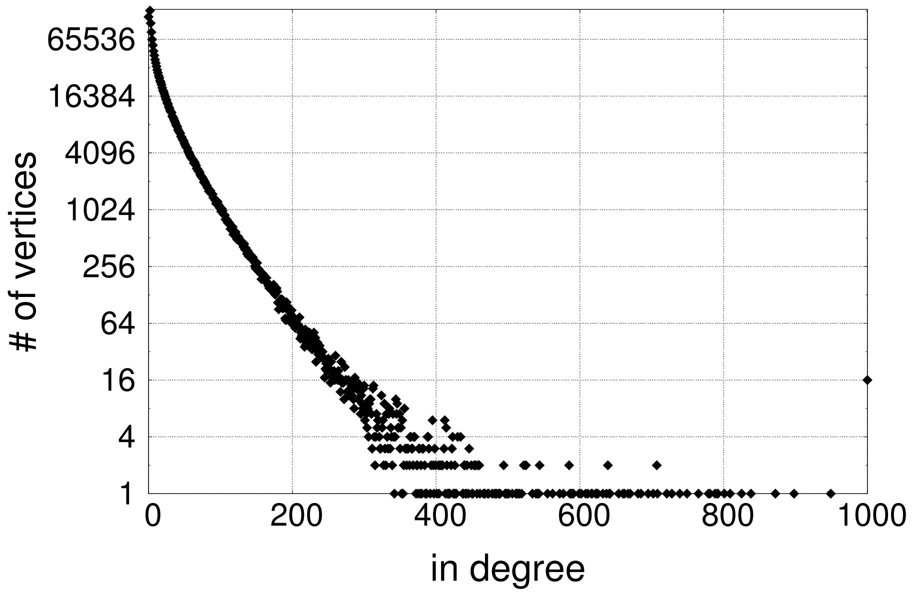


(a) In Edges

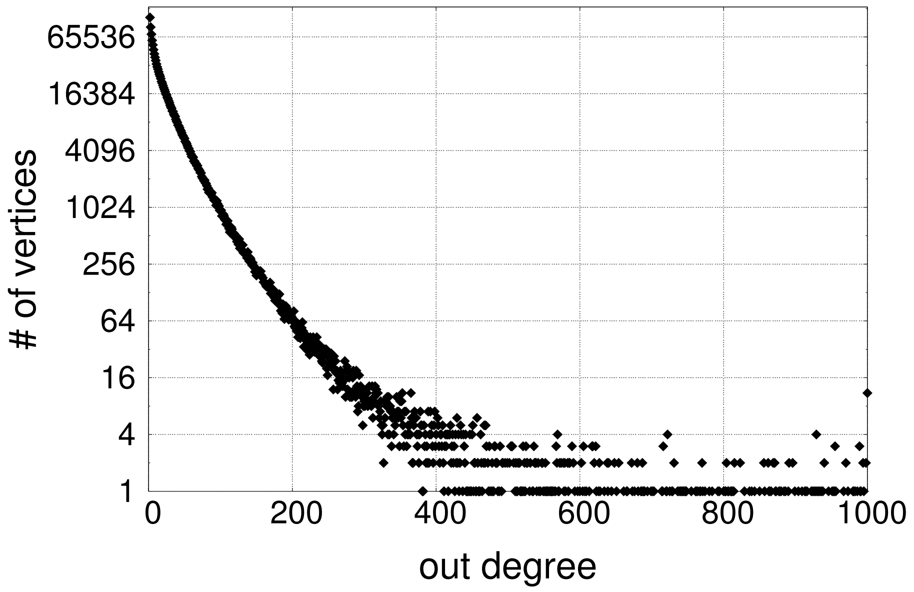


(b) Out Edges

Figure 3.6: Vertex degree distribution of wg dataset.

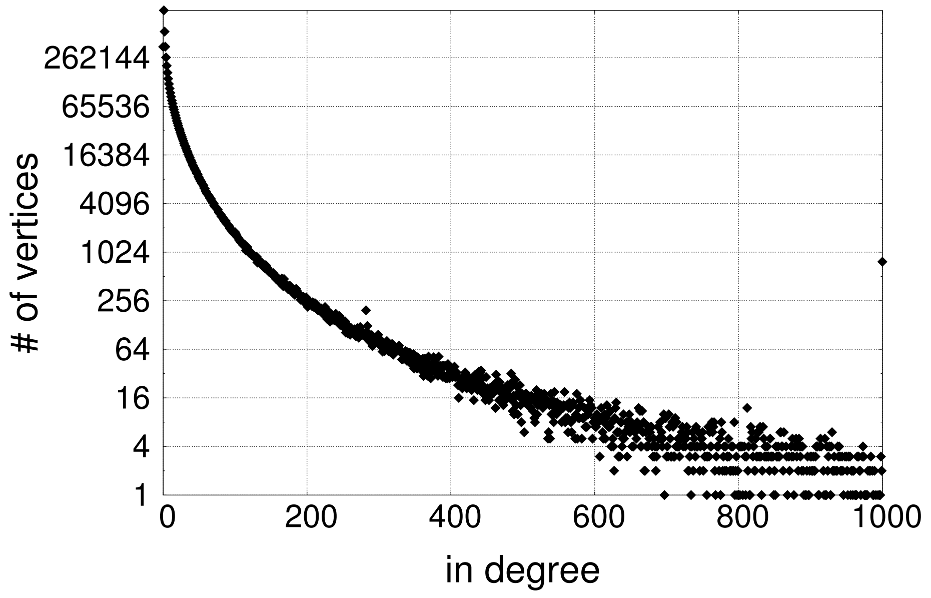


(a) In Edges

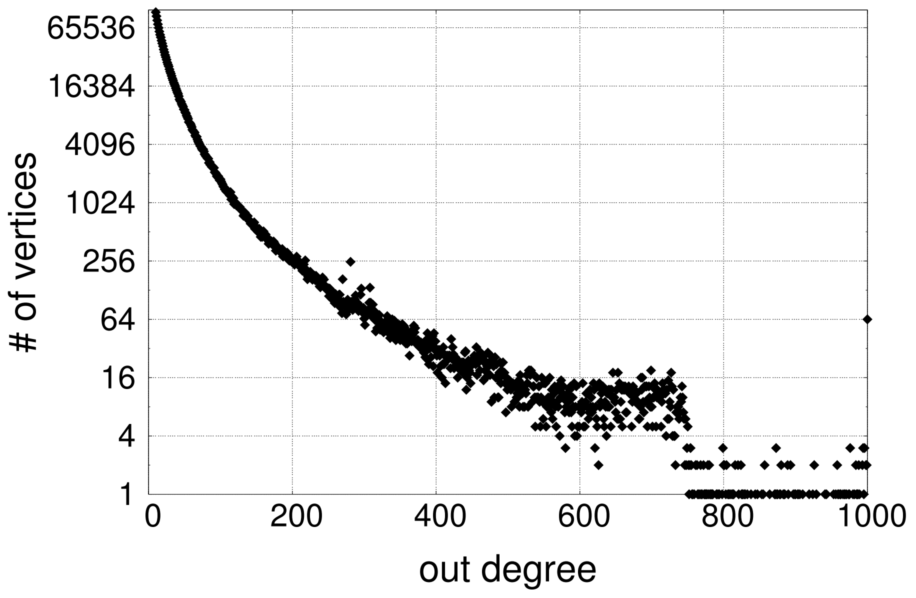


(b) Out Edges

Figure 3.7: Vertex degree distribution of pk dataset.



(a) In Edges



(b) Out Edges

Figure 3.8: Vertex degree distribution of lj dataset.

Chapter 4

Proposed Template and Architecture

4.1 Graph-Parallel Abstraction (Gather-Apply-Scatter) and Data Types

We have already reviewed the vertex centric execution model in Section 3.1.1. As mentioned previously, many parallel and distributed frameworks adopted the vertex centric execution model as programming interface. Most famous examples of this model are Pregel [3], GraphLab [2], and Galois [37]. In this work, we will also focus on Gather-Apply-Scatter (GAS) model as in GraphLab. In GAS model, programmer implements 3 main user functions which composes the vertex program. Moreover, user is able to define several data types to represent graph data in the memory such as vertex data and edge data. The following sections will explain the interface for user defined functions and data types in detail.

4.1.1 User Defined Functions

As in GraphLab, our template provides three main functions: (1) *Gather*, (2) *Apply*, and (3) *Scatter*. These functions operate on given input vertex as follows:

- **Gather:** Gather operation iterates on incident edges and neighbour vertices. It executes an accumulation function on neighbours and incident edges. Result of accumulation operation is passed to the apply function.
- **Apply:** Apply determines the updated/new value of the input vertex's data.
- **Scatter:** Scatter operation activates the neighbour vertices according to the current value of the input vertex. It may also distribute the data calculated in apply to the neighbours.

GatherInit:	Initialize GatherState before processing incoming edges
Gather:	Update the GatherState using the neighbouring EdgeDataG and SharedVertexData
GatherFinish:	Finalize the GatherState after processing incoming edges
Apply:	Perform the main computation for *v* using the collected data
ScatterInit:	Initializes ScatterState before processing outgoing edges
Scatter:	Distribute the computed data to neighbours. Determine whether to schedule the neighbouring vertices in the future
ScatterFinish:	Finalize updates to the VertexData associated with the current vertex

Figure 4.1: Vertex-centric execution with Gather-Apply-Scatter abstraction.

In addition to the three main functions, the template includes helper functions. For both *Gather* and *Scatter* functions, we have *init* and *finish* functions. For both cases init functions are used for initializing the associated state data for corresponding operations while finish is used to finalize the associated state and update the associated graph data. A summary of the template functions can be found in Figure 4.1.

4.1.2 User Defined Data Types

We can categorize the data types in our template into two categories such as *graph data types*, which are used to store the graph in memory, and *local data types*, which are used as intermediate communication mediums.

The template provides 8 different data types to ease the programming. First of all, main data type is *VertexData* which is divided into 2 different types: (1) PrivateVertexData and (2) SharedVertexData. *PrivateVertexData* keeps the data which is only accessible by the corresponding vertex. On the other hand, *SharedVertexData* also keeps the data which is accessible by the neighbours. In addition to the VertexData, there are 2 more data types associated with edges and these are: (1) EdgeDataG which is accessed by the *Gather* and (2) EdgeDataS which is accessed by the *Scatter*.

Moreover, in order to create the communication between 3 main functions, the template gives 3 state data types which are *GatherState*, *ApplyState*, and *ScatterState*. Figure 4.2 summarizes the user defined data types in our template.

PrivateVertexData:	Data associated with one vertex that can be accessed by only the corresponding vertex
SharedVertexData:	Data associated with one vertex that can be accessed by neighbouring vertices
VertexData:	Combination of PrivateVertexData and SharedVertexData
EdgeDataG:	Data associated with one edge that is used in the Gather stage of neighbouring vertices
EdgeDataS:	Data associated with one edge that is used in the Scatter stage of neighbouring vertices
GatherState:	The state computed in the Gather stage and passed to the Apply stage
ApplyState:	The state computed in the Apply stage and passed to the Scatter stage
ScatterState:	The state computed in the Scatter stage

Figure 4.2: The application-level data structures.

4.1.3 PageRank Implementation in the Template

As an example implementation in our template, we will consider PageRank application. PageRank is used to calculate the relative importance of a web page in a web graph by analysing the importance of neighbouring vertices of a web page.

While Figure 3.1 shows the details of the PageRank algorithm, user needs to map this computation to the proposed template functions and data types which are given in 4.1 and 4.2.

GlobalParams:	ε *// error threshold*
	rankOffset //$= (1-\beta)/\lvert V\rvert$
	β
PrivateVertexData:	oneOverDegree //$= 1/d_v$
SharedVertexData:	scaledPR //$= r_v/d_v$
VertexData:	**PrivateVertexData** & **SharedVertexData**
ApplyState:	doScatter
GatherState:	accumPR *// accumulated rank value*

Figure 4.3: Data types of PageRank application for our template.

The global parameters such as error threshold (ε), rankOffset ($(1-\beta)/|V|$), and the rank scaling coefficient (β) are stored as global parameters (globalParams). Moreover, the *VertexData* is stored in two parts which are *SharedVertexData* and *PrivateVertex-Data*. As can be seen from Figure 4.3, SharedVertexData keeps *scaledPR* which will be accessible by the neighbour vertices while *oneOverDegree* is kept by the PrivateV-ertexData and will be accessible by only the vertex itself.

Firstly, we accumulate the pagerank values of the neighbours in gather. Note that, *GatherState* stores an *accumPR* value which is initialized to be zero in *gather_init*. Then, *gather* accumulates on accumPR by visiting neighbours' vertex data. Secondly, the new pagerank value for the vertex is calculated by using GatherState in *apply*. Apply also checks whether the change on the vertex data is above the given error threshold or not and according to this information assigns *doScatter* to true or false. Finally, if doScatter is true then scatter is executed and neighbour vertices are activated. Figure 4.4 shows the details of each function.

```
gather_init()
  Output gatherSt:GatherState
1.     $gatherSt.accumPR = 0$ // initialize accumulated rank

gather()
  Input otherVD:SharedVertexData // other vertex data
  Output gatherSt:GatherState
2.     $gatherSt.accumPR+ = otherVD.scaledPR$

apply()
  Input gatherSt:GatherState
  Input/Output localVD:VertexData // local vertex data
  Output applySt:ApplyState
3.     $newRank = rankOffset + \beta * gatherSt.accumPR$
4.     $newRankScaled = newRank * localVD.oneOverDegree$
5.     if $|newRankScaled - localVD.scaledPR| > \varepsilon$
6.     then
7.       $applySt.doScatter = true$
8.     end
9.     $localVD.scaledPR = newRankScaled$

scatter()
  Input applySt:ApplyState
10.    if $applySt.doScatter == true$
11.    then
12.      $activateNeighVtx = true$
13.      // send activation for neighbour
14.    end
```

Figure 4.4: Pseudocode of the PageRank application for our template.

4.2 Proposed Architecture

We have already defined the template functions which can be specified by the user as discussed in Section 4.1. In this section, we will explain how we have addressed the challenges stated in Section 3.1 and we will explain the details of the corresponding hardware components in the system.

The main features of the proposed architecture can be summarized as follows:

1. Gather and Scatter are able to handle tens of vertices and hundreds of edges in parallel. This increases memory level parallelism and hides the latency of memory accesses.
2. Keeping partial states in Gather and Scatter allows us to distribute the workload for high degree vertices to multiple partial states. This allows us to handle load balancing for scale-free graphs.
3. To handle synchronization, a sophisticated synchronization unit is proposed. Synchronization Unit (SYU) ensures that execution follows sequential consistency. Moreover, SYU works in a distributed fashion and minimizes the overhead of synchronization.
4. The system keeps an active list for non-converged vertices to exploit work efficiency thanks to asymmetric convergence.
5. The system also implements a memory subsystem which is tailored for graph applications.

Figure 4.5 shows the internal architecture of a single accelerator unit. This is a loosely-coupled accelerator connected to the system DRAM directly. As shown in the figure, the accelerator architecture consists of several components, which will be explained here briefly. Active List Manager (ALM) is responsible for keeping the set of active vertices that need to be processed before convergence. Runtime Unit (RT) receives vertices from ALM and schedules them for execution based on the resource availability in the system. RT sends the next vertex to Sync Unit (SYU) to start its execution.

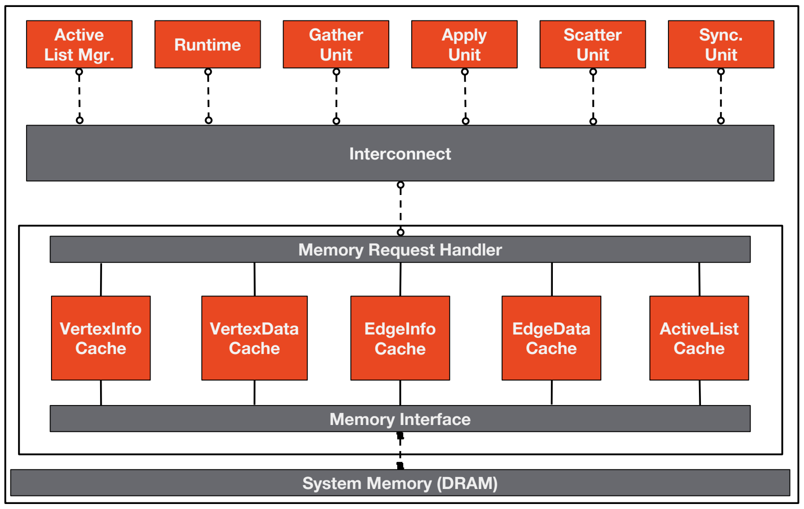


Figure 4.5: Single accelerator unit.

SYU is responsible for making sure that all edges and vertices are processed and sequential consistency is guaranteed. SYU checks and avoids the potential read-after-write (RAW) and write-after-read (WAR) hazards. Then, SYU sends the vertices to the Gather Unit (GU), which is the starting point of the vertex program execution. It executes the *gather* operation as discussed in Section 4.1. An important feature of GU is that it can process tens of vertices and hundreds of edges to hide long access latencies to the system memory. It can also switch between many small-degree vertices and few large-degree vertices for dynamic load balancing. After GU is done with the gather operation of a vertex, its state data is sent to the Apply Unit(APU), which performs the main computation for the vertex. After APU is done, vertex data is passed to the Scatter Unit where the *scatter* operation (Section 4.1) will be done. In this stage, the neighbouring vertices can be activated (i.e. inserted into the active list) based on application-specific conditions. Similar to GU, SCU also processes tens/hundreds of vertices/edges concurrently. In addition to the computational modules, there is a special memory subsystem, consisting of caches for different data types, specifically optimized for graph applications. Readers can refer to [38] for further details of this architecture.

4.2.1 Details of Hardware Components

In this section, we will describe computational units in detail.

4.2.1.1 Gather Unit

Gather Unit (GU) executes the *gather* function of the user program. There are two main features of the GU which enables us to hide latency of memory accesses. First GU stores partial states for vertices (vertex rows) and can keep multiple vertices in its limited local storage. Second, GU stores many edges (edge slots) and distribute available storage for edges by using a credit based system. The credit based system considers two main properties of vertices which are being processed: (1) priority of the vertex for ordering purposes, (2) degree of the vertex. While property (1) helps sequential consistency, latter provides load balancing. GU may assign all available edge slots to a high degree vertex or many low degree vertices may share the available edge slots. Also note that, assignments for edge slots are done dynamically.

4.2.1.2 Apply Unit

Apply Unit (APU) executes the *apply* function of the user program. APU takes current vertex data and *GatherState* as input. However, APU does not interact with system memory. Moreover, the execution of APU is pipelined in order to process multiple vertices in parallel.

4.2.1.3 Scatter Unit

Scatter Unit (SCU) executes the *scatter* function of the user program. Implementation details of the SCU are similar to the GU. As GU, SCU also keeps vertex rows and edge slots to enable, mainly, load balancing. Moreover, processing vertices and edges in parallel hides memory latency.

In addition to the aforementioned properties of SCU, SCU also sends activation messages to the Sync. Unit (SYU). If the scatter function written by the user wants to activate the neighbour an activation message with a true flag will be sent. Otherwise, a message with false flag will be sent. The purpose of latter message is to prevent WAR hazards. The final vertex data is not written back to the memory until it receives an acknowledgement message from the SYU.

4.2.1.4 Sync. Unit

Main duty of the *Sync. Unit (SYU)* is to ensure sequential consistency. SYU avoids read-after-write (RAW) and write-after-read (WAR) dependencies. Moreover, it avoids multiple activations for the same vertex.

SYU uses edge consistency model to ensure sequential consistency. In edge consistency model, vertices are only allowed to update its own vertex data. Additionally, edge consistency enforces an ordering between adjacent vertices. Basic idea behind SYU's operation is to assign rank to the vertices. Lower ranks are assigned to the vertices which starts their execution earlier than the others and ranks are increased monotonically.

Operations of the SYU are the following:

- Vertex states are kept in SYU. When SYU receives a vertex from the runtime unit, the vertex receives a unique rank and stores it in a table. The table keeps the ID and the execution state of the vertex. When SYU receives gather-done or scatter-done signals it updates the vertex state.
- SYU maintains the RAW ordering. When there are two different vertices u and v with an edge $e : u \rightarrow v$ between them being executed and $rank(u) < rank(v)$, SYU enforces the logical execution order of u, v. Basically, SYU makes sure that vertex v does not read the data for either u or e.
- SYU maintains WAR ordering.

- SYU avoids unnecessary activations. When there is an edge $e : u \rightarrow v$ and u wants to activate v, if vertex v is already queued for processing, it is not necessary to activate vertex v since it will be processed by the Accelerator Unit after vertex u will be able to use the most current value of vertex u since SYU also enforces sequential consistency.

4.2.1.5 Active List Manager

Active List (AL) keeps the list of non-converged vertices. Active list manager (ALM) has two main duties. First one is to extract active vertices from active list and dispatch them to the runtime unit. Secondly, ALM receives activation requests from SYU and adds them to the active list. While unnecessary activations are handled in SYU, ALM still needs to prevent duplications in the active list.

To access the data efficiently, ALM divides storage of active vertices into two parts: (1) ALM keeps bit vectors which can track 256 vertices and represent each vertex with a single bit. (2) ALM keeps a queue of bit vector indices.

When ALM is extracting the active vertices, it first accesses the queue which keeps bit vector indices. Then, ALM accesses the bit vector which corresponds to the dispatched ID. After that, ALM sends the vertices which have ”1”s in the bit-vector to the runtime for execution. When a vertex is sent to the runtime unit, its bit is set to ”0” in the bit-vector.

When there is an activation request, ALM first checks the local buffers. If the bit-vector exists in the buffer then ALM basically sets the corresponding bit locally. Otherwise, ALM sends a request to the active list load store unit.

Moreover, ALM needs to take care of bit-vectors in flight. Specifically, when a vector is sent to runtime unit for execution, it also needs to be registered to Sync. Unit and an acknowledgement should be received from SYU before removing it from the bit-vector. Secondly, ALM and AL load store unit should co-operate to prevent multiple activations for the same vertex.

4.2.1.6 Runtime Unit

Runtime Unit (RT) monitors the available resources in the system. It receives vertices to process from ALM and sends them to the SYU. Runtime is also responsible for checking the termination condition and sending the completion signal when there are vertices in execution and AL is empty.

4.2.1.7 Memory Subsystem

The accelerator template uses the Compressed Sparse Row format for storing the graph in the memory. Edge Info (EI) that keeps pointers to the destination vertices of edges are stored sequentially. The offsets and edge counts of vertices are stored as Vertex Info (VI). Additionally, user specific data types mentioned in Section 4.1 are also stored in main memory. Similarly, active list needs to be stored in main memory.

In the template, we have specialized caches and load store units for each data type in the graph. Locality for each data structure and access patterns can differ for each data type. For instance, while spatial locality is high for EI, spatial locality is low for vertex data, especially when accessing neighbour vertices. Our template also allows user to change the parameters for caches.

4.2.2 Multiple Accelerator Units

While a single accelerator unit is optimized for maximizing the throughput, we can achieve higher throughput and thus better performance by increasing the number of accelerator units. When replicating the accelerator units, we also need to take care of vertex partitioning. In this template, we adopt a simple static partitioning method that each chunk of 256 vertices are assigned to a single accelerator unit.

Memory accesses are also categorized according to their access types. First one is local accesses that VI and EI are accessed by a single accelerator unit and thus accessed by a local memory request handler. On the other hand, data types which can

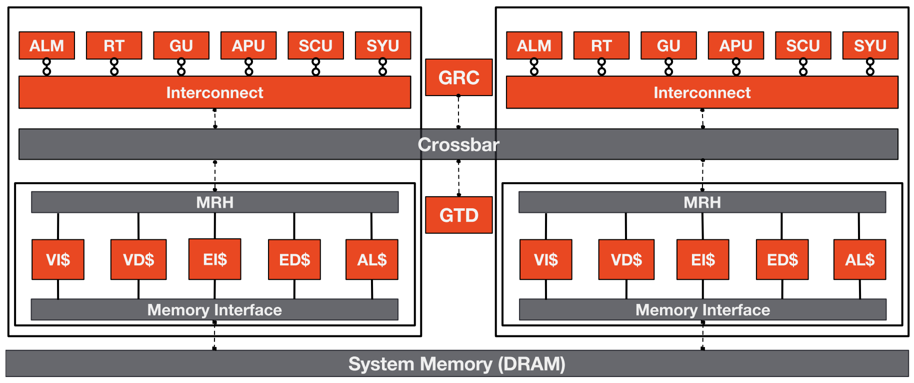


Figure 4.6: Multiple accelerator units with a crossbar.

be accessed by multiple accelerator units such as VD and ED access requests should go through a global memory request handler.

Moreover, global data accesses go through SYUs. For example, when GU is accessing the neighbour's vertex data, it first sends the request to the other AUs, which processes the neighbour vertex, and SYU resolves RAW hazards.

While there are multiple accelerator units running, we need additional runtime and synchronization managers. For this purpose, we added the following modules to our design:

- *Global Rank Counter (GRC):* While SYU handles the sequential consistency, we need unique rank assignments for all vertices. Uniqueness and monotonic increases for rank assignment is managed by global rank counter.
- *Global Termination Detector (GTD):* The runtime unit is responsible for detecting termination and checking the emptiness of Active Lists. When there are many accelerator units, we need another centralized unit that will check each AU to see whether they finished their execution or not. Basically, GTD controls all AUs and notifies the host processor when all AUs are done.

Note that, GRC and GTD are the only centralized modules in our template and both have very simple duties and implementations.

4.3 Design Methodology

4.3.1 Design Flow

In Section 4.1, we have described the application development interface which is exposed to the programmer. Basically, a user needs to provide the definitions of the data structures and template functions for the template. Moreover, in Section 4.2, we described the details of the underlying architecture and how these constructs use the user defined data structures and template functions.

In this section, we will present the methodology used for generating the final hardware with given user defined data structures, user defined template functions, and predefined hardware constructs.

4.3.1.1 User Parameters

In addition to the user defined functions and data structures, users are able to define several parameters that are used to determine the size of data type specific caches and buffers in gather and scatter units. Figure 4.7 shows the list of these parameters. Specifically, we have 4 parameters to define the size of data type specific caches which are VICacheEntries, VDCacheEntries, EICacheEntries, and EDCacheEntries that are used to specify the size of caches for VertexInfo, VertexData, EdgeInfo, and EdgeData consecutively. On the other hand, users are able to define the number of in-flight vertices and edges in gather and scatter units via the parameters VtxRowCnt and EdgeSlotCnt.

As expected, size of caches and buffers would affect the performance and the area of the accelerator. However, users can manually test their design by changing the aforementioned parameters. Since manual tuning can be time consuming, we have developed an automated methodology which will be discussed in Section 4.3.2.

GatherEdgeSlotCnt:	Number of edges stored in Gather Unit
GatherVtxRowCnt:	Number of partial vertex states stored in Gather Unit
ScatterEdgeSlotCnt:	Number of edges stored in Scatter Unit
ScatterVtxRowCnt:	Number of partial vertex states stored in Scatter Unit
VICacheEntries:	Number of cache entries for VertexInfo
VDCacheEntries:	Number of cache entries for VertexData
EICacheEntries:	Number of cache entries for EdgeInfo
EDCacheEntries:	Number of cache entries for EdgeData

Figure 4.7: Performance model parameters

4.3.1.2 Functional and Performance Validation

As any design validation process, our framework provides a fast functional simulator to the user. After a user designed their application with the given template, she can run functional simulator to validate her design. Note that, functional simulator runs nearly as fast as native C/C++ application and has no micro-architectural details. Therefore, it cannot be used to test the performance of the designed accelerator.

After functional validation, user can use the given cycle-accurate simulation to test the performance. For this purpose, a user needs to synthesize user defined functions and provide timing annotations to the template. This latency info is also parametrized in cache and buffer sizes. In this step, user can exploit capabilities of the HLS tools such as pipelining, loop unrolling etc. to make the generated hardware more efficient.

After functionality and the performance of the system is verified, same models can be used to generate final RTL for the accelerator which can be used to generate FPGA or ASIC realizations of the accelerator.

4.3.2 Design Space Exploration

As mentioned in Section 4.3.1, there are many micro-architectural parameters that can affect the performance and area of the template proposed. As shown in Figure 4.7, the first set of parameters are the size of buffers in gather and scatter units for edges and vertices. As stated previously, the buffers in gather and scatter units store temporary/partial execution states and help us to hide latency of memory operations. Specifically, increasing the size of vertex and edge buffers can improve the performance but these units occupy larger areas. Additionally, the second set of parameters, the cache sizes, can also affect the performance and area. For instance, if a data type shows good locality then increasing the cache size can improve the performance.

In our template, the trade-off between the performance and area/power depends on the application since each application can have different computations and memory access patterns. For example, if we consider PageRank application, we can see that gather will cause a performance bottleneck since it will do many non-sequential memory accesses and performs most of the computation. Therefore, it is reasonable to increase the buffer sizes for vertices and edges in gather unit. In contrast, accesses to neighbour vertex data will show poor locality. Thereby, it is better to keep the vertex data cache small and save area/power. For a single application, i.e. PageRank, we saw that even a single unit has many parameters to consider in order to optimize the performance and area/power trade-off. For this reason, a user should select the parameters carefully according to the application characteristics and consider how much extra area/power she is willing to sacrifice for a certain amount of performance gain.

If we consider all possible micro-architectural parameters, the design space for manual tuning becomes very large. Additionally, manual tuning of parameters requires in depth knowledge of the applications and the template constructs. However, this is against our motivation that a software developer should be able to create an accelerator without the knowledge of the underlying hardware. For these reasons, we propose an automated design space exploration methodology which can generate a Pareto curve for performance and area/power. This approach allows users to select the design point which fits best for their design limitations.

In the automated design space exploration tool, we have used the throughput (number of edges processed per second) as a proxy for the performance and the area as a proxy for the energy consumption. While estimating the throughput, we run the application long enough to avoid problems due the the warm-up period. To estimate the area of gather and scatter units, we first calculated the area of these units for multiple design points (different VtxRowCnt and EdgeSlotCnt pairs) and applied curve fitting techniques to calculate the area for remaining design points. For cache area calculations, we have used a well-known tool, Cacti [39].

For generating the Pareto curve, we have used a linear function of throughput (T) and area (A): $T - \alpha A$. The aim of this methodology is to maximize the given objective function where we can generate multiple design points by changing α value. One challenge we faced was the different units and value ranges for T and A. Therefore, selecting α values is not a trivial task. On the other hand, aforementioned linear function has an advantage such that the slope of the Pareto curve for a particular point will have the slope of α.

Specifically, for investigating the area-performance trade-off, we propose a two step methodology designed for the needs of graph applications. Since the search space is large and choosing an arbitrary point as a starting point for Pareto curve generation is challenging, we start with a greedy optimization for a well-known metric, area-delay product. In the first step, we try to optimize aforementioned metric by a randomized greedy approach. More specifically, we iteratively increase the size of each component which gives us the best area-delay product. As a result, we get an area, throughput pair that we can use as a normalization point in the second step (a throughput area pair is found: $< T_0, A_0 >$).

When we have the initial point for normalization, then a different heuristic will try to optimize a trade-off between throughput and area increase with the metric: $R = T - \alpha A$ which T, and A values are calculated with respect to T_0 and A_0. In this step, we explore different design points by sweeping α value which controls the impact of area increase relative to the increase in throughput. The details of our methodology can be found in Figure 4.8. At each iteration of the second step, we explore both directions for every parameter available (both increasing and decreasing the size of corresponding hardware unit). Among all options, algorithm selects the best move which gives the highest R value. Corresponding parameter is updated with its new value according to the selected move. Similarly, next iteration uses updated parameter values and selects another parameter to update which gives the best R value. The second step continues

```
Design_Space_Exploration()

1.  Metric $m_1 \leftarrow$ maximize $T/A$
2.  $(T_0, A_0) \leftarrow$ OptimizeParams(m1)
3.  $i \leftarrow 1$
4.  for a set of different $\alpha$ values do
5.      Metric $m_2 \leftarrow$ maximize $(T/T_0 - \alpha A/A_0)$
6.      $(T_i, A_i) \leftarrow$ OptimizeParams($m_2$)
7.      Add $(T_i, A_i)$ to the Pareto curve
8.      $i \leftarrow i+1$
9.  return Pareto curve
```

Figure 4.8: The high-level algorithm for the design space exploration.

until there is no move that can improve the R value calculated in the previous iteration. Finally, $< T_i, A_i >$ pair found as result for a single α value is added to the Pareto curve. Note that, second step is executed for multiple α values. The details of selected α values can be found in Section 5.2.4.

Chapter 5

Experiments[1]

5.1 Experimental Setup

Using the proposed architecture template, we generated accelerators for 4 applications (outlined in Section 5.1.1), and compared with a state-of-the-art IvyBridge server system. Details of the execution environments are as follows:

- *CPU:* This is the baseline against which we compare our accelerators. The system is composed of two sockets. Each socket has 12 cores. Each core has private L1 and L2 caches, and the L3 cache is shared by cores on the same socket. Total cache capacity is 768KB, 3MB, and 30MB for L1, L2 and L3 respectively. Total DRAM capacity of the system is 132GBm while the software is implemented in OpenMP/C++. Applications are either hand optimized, or reused from existing benchmark suites. Each application is compiled using gcc 4.9.1 version with -O3 flag enabled. When needed, we set the NUMA policy to divide the memory allocation for an application to two different sockets on the system to maximize the memory bandwidth utilization. The applications in our experiments cannot effectively utilize the vector extensions of the CPU due to the reasons explained in Section 3.

[1]**This chapter is reproduced from [38] ©2016 IEEE, for copyright information see [38]**

Table 5.1: Parameters used for accelerators constructed.

		Gather Unit		Scatter Unit		Cache
	# AUs	# vtxs	# edges	# vtxs	# edges	size
PR	4	32	128	16	128	9.9 KB
SSSP	4	32	4	16	128	8.9 KB
LBP	4	16	64	16	64	34.8 KB
SGD	4	16	64	16	64	9.6 KB

- *ACC:* This is the accelerator generated by the proposed architecture template for each application. The architectural parameters are customized per application. The main parameters are listed in Table 5.1. Observe that 4 Accelerator Units (AUs) are used for all applications. The number of vertices and edges concurrently processed in each AU are also listed for the Gather and Scatter Units. Finally, the total cache storage in the memory subsystem of each AU is listed in the last column.

As discussed in Section 3, GPUs are not well-suited for irregular graph applications. There are several existing works that have compared GPU performance with CPUs. For example, it is reported that a GPU implementation of Stochastic Gradient Descent (SGD) performs as good as 14 cores on a 40-core CPU system [15]. For Single-Source Shortest Path (SSSP) problem, it is reported that an efficient serial implementation can outperform highly parallel GPU implementations for high-diameter or scale-free graphs [16]. A GPU-based sparse matrix-vector multiplication implementation of PageRank has been proposed recently [13], where 5x speed-up is observed with respect to a 4-core CPU. However, this work ignores the work-efficiency advantages of asynchronous execution and asymmetric convergence (Section 3). It has been shown that a synchronous implementation can be up to 3x less work efficient compared to an implementation that keeps track of active vertices and performs asynchronous computation [31].

5.1.1 Graph Applications

In order to test our framework, we have selected widely used graph applications from different domains, such as machine learning, computer vision, and data mining, which are briefly described below.

PageRank (PR): It is an important graph application used to order web pages according to their importance. The pseudo code of the algorithm is given in Figure 3.1. As a baseline, we use the multi-core CPU implementation from the Berkeley GAP Benchmarks [40]. We extended the existing implementation to improve convergence behaviour by adding a bit vector to keep track of active vertices. In our ACC implementation, the *PageRank (PR)* value of a vertex and $1/out\_degree$ of a vertex is stored as vertex data. In a vertex program execution, the current vertex collects and accumulates the *PR* values from its neighbours and updates its own *PR* value.

Loopy Belief Propagation (LBP): It is a well-known image stitching algorithm that works on a grid graph. Each vertex in the graph represents a pixel of a given image. More specifically, each vertex has a belief vector where each entry represents the probability of the corresponding label for the vertex. The following 3 stages are performed for each vertex: 1) The messages from neighbours are accumulated. 2) The belief of the vertex is updated based on the accumulated value. 3) A new message is generated using min convolution and sent to each neighbouring vertex. The CPU implementation uses a synchronized execution model. Specifically, a ”2-colouring” scheme is implemented, where the vertices with the same color are executed in parallel, avoiding the need for locks. Our ACC implementation is similar to GraphLab, where edges store the messages in both directions and vertices keep belief values. Initially, a vertex program visits all incident edges of the vertex and calculates the log-sum (product) of all incoming messages. Then, the vertex updates the belief value of its own. Finally, for each incident edge of this vertex, the outgoing message values are updated.

Stochastic Gradient Descent (SGD): It is an iterative machine learning algorithm used in recommender systems. SGD operates on a bipartite graph, and tries to estimate a feature vector for both user and item vertices of the graph. The dot product of a user and item vector is expected to give the estimated rating of the user for that item. We used the DSGD algorithm [41] as the baseline CPU implementation because it is shown to be the most efficient implementation of SGD in [5].

Single Source Shortest Path (SSSP) As a baseline CPU implementation, we used the SSSP implementation from the Berkeley GAP benchmarks [40]. Special bucket-based data structures are used in that implementation to achieve high parallel performance. In our accelerator implementation, distance of a vertex to a source node is stored as vertex data. If the distance value of a vertex is updated, it sends a message to its neighbours. Each edge is assigned a unit weight in the input data that we use in our experiments.

5.1.2 Power, Performance, and Area Estimation

5.1.2.1 Methodology for CPU

We used the time measurement function calls that exist in the OpenMP library. To calculate the energy and power consumption of the native system, we used Running Average Power Limit (RAPL) [42], which provides energy measurements for core, uncore and DRAM by allowing us to read MSR registers. The baseline CPU system uses DDR3 as the system memory. For fair power comparisons with accelerators, we estimated DDR4 power consumption separately from the CPU system with DRAMSim2 [43]. For that purpose, we generated DDR4 access traces that result in the same bandwidth as DDR3 of the CPU system and then applied them to DRAMSim2 with a DDR4 device model.

5.1.2.2 Methodology for Accelerator Compute Blocks

We used a commercial high-level synthesis (HLS) tool to generate RTL from our SystemC-based performance models in order to estimate area, performance and power for each block. HLS was run for each application on five main blocks of the accelerator unit: Gather, Scatter, Apply, Sync and ALM. Then, the generated RTL was run through a commercial physical-aware logic synthesis tool to confirm absence of timing violations and to measure area and power at the gate-level. We used 22*nm* technology library for standard cells and metal layers and a 1GHz clock frequency. Significantly large arrays (about 1Kb and larger) were implemented using synthesizable latch-based register files (RF). For the crossbar block, we estimated the wire length of interconnects between all the blocks in the accelerator unit based on an approximate floorplan and the area of the blocks. We estimated area and power for the crossbar using the wire length, routing resources and physical parameters of the metal layers.

Most of the SystemC template-related functions were modelled at cycle-accurate level to provide accurate performance estimates. Application-specific functions, such as scatter, gather and apply, were pipelined using the pipelining feature of the HLS tool. We were able to achieve a throughput of 1 function call per cycle for every user function except the Gather function of SGD, which has throughput of 1/4 due to significantly more computation done per vertex. The latencies were also 1 except for the *Apply* function of PageRank, *Scatter* function of LBP, and the *Gather* function of SGD. Their latencies are 3, 6, and 4 cycles, respectively.

The latency/throughput values for user functions are back-annotated to the original SystemC model for performance measurements. For power measurements, we used a hybrid SystemC-Verilog simulation methodology, where RTL for the block of interest and annotated SystemC models for the rest of the blocks were used to generate power traces. During simulation, we captured switching activity for all inputs and sequential elements of the RTL block in SAIF format. Then, we used a commercial power analysis tool that takes the SAIF file as input and produces power values for the given switching activity file.

5.1.2.3 Methodology for Memory Subsystem

The accelerator memory subsystem is composed of internal memories such as caches and light-weight load/store queues, and DRAM. We estimate the power and area of internal memories using Cacti 6.5 [39]. Since Cacti only supports down to 32*nm* technology, we apply three different scale factors to convert them to 22*nm* technology. For area, we used the scaling factor 0.5 based on [44, 45]. For dynamic power, we used the scaling factor 0.569 as in [46]. Finally, for the leakage power, we used the scaling factor 0.8 as in [47]. In order to estimate dynamic power consumption, we first compute the dynamic energy consumption by measuring access count of each memory component and then multiplying it by the energy per access provided by Cacti. For example, we collect energy per access through Cacti for a cache and run simulation to get the access count of the cache. Then, we multiply them together to estimate the dynamic energy consumption of the cache. For leakage energy, since leakage current is always consumed as long as power is turned on, we simply multiply the total execution time by leakage power. By summing up the dynamic and leakage energy, we can compute the total energy consumption. Total power consumption is simply computed by dividing total energy with total execution time. DRAM power is computed using DRAMSim2[43] with a DDR4 memory model.

5.1.3 Datasets

We tested each application with several datasets, either taken from existing graph datasets or created synthetically. For SSSP and PageRank applications, we selected 3 different directed graphs from the SNAP datasets [33]: *WebGoogle(wg)*, *soc-Pokec(pk)*, and *soc-LiveJournal(lj)*. We generated three large graphs using Graph500 [48] with 16, 32, and 67 millions vertices. For LBP, we synthetically generated three different graphs using GraphLab's synthetic image generator [2]. Each image has 4 different colors, and hence, there are 4 different possible labels for each pixel. Images generated for LBP tests include 1000x1000, 2000x2000, and 3000x3000 pixels (vertices). For the SGD application, we selected two different movie datasets from MovieLens [49]. The first movie dataset (1M) includes approximately 1 million ratings and the second one (10M) includes approximately 10 million ratings. Table 5.2 shows the detailed description of each dataset with its respective properties.

Table 5.2: Datasets used in our experiments.

Application	Dataset	# Vert.	# Edges
PageRank SSSP (Directed)	wg	916K	5.1M
	pk	1.6M	30M
	lj	4.8M	69M
	g24	16.8M	268M
	g25	33.5M	536M
	g26	67M	1000M
LBP (Undirected)	1M	1M	2M
	4M	4M	8M
	9M	9M	18M
SGD (Undirected)	1M	9.7K	1M
	10M	80K	10M

5.2 Experimental Results

In this section, we present our experimental results in terms of execution time, power and area. We provide results for 17 different test cases, where each test case is an application-dataset pair.

5.2.1 CPU and Accelerator Comparison

5.2.1.1 Execution Time and Throughput

As mentioned in Section 5.1, we used a 24-core server system as our baseline for these experiments. We used identical convergence conditions for the CPU and accelerator implementations so that the execution time comparisons make sense. In this section, we report the performance results in terms of total execution times (Figure 5.1) and throughput values (Figure 5.2). Throughput is defined as the number of edges processed per second. Note that throughput is a raw performance metric, because it does not take into account the convergence behaviour. As shown in [31], an implementation can have higher throughput, but worse execution time, especially if the properties described in Section 2 are not taken into account.

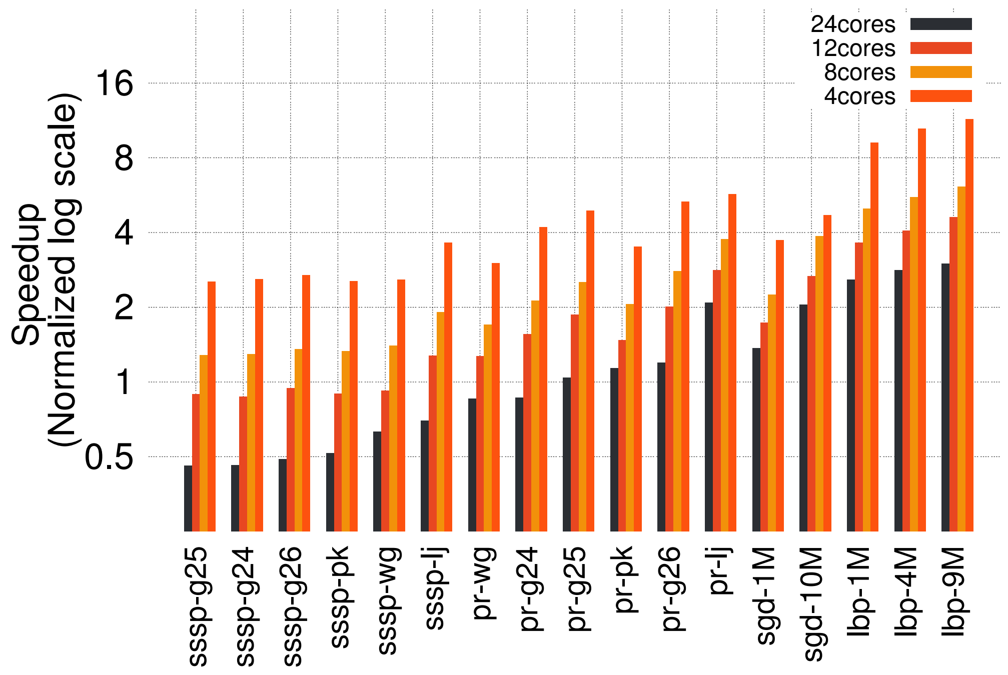


Figure 5.1: Execution time comparisons. The y-axis is the speed-up of the proposed accelerators with respect to multi-core execution.

PageRank is one of the best examples of an iterative, converging graph application, which benefits from asynchronous execution as shown in [2]. Although the baseline CPU implementation has some asynchronous execution support, its vertex scheduling is not asynchronous [2]. Note that the performance benefits of our accelerators are higher when the execution time metric is considered (Figure 5.1) compared to the raw throughput metric alone (Figure 5.2). This shows the importance of the asynchronous mode support in our architectures. Compared to the 24-core system, our accelerators have better or equivalent execution times in 4 out of 6 test cases[3]. Compared to 12 or fewer cores, the speed-up observed is in the range of 2x to 20x.

For the LBP application, observe that our throughput values are comparable to the throughput of 12 cores. However, when the total execution time is considered, our accelerator is between 2.5x and 3x faster than 24-cores. We believe the reason for this

[2]Fully asynchronous multi-core implementation would require more synchronization, which would lead to worse execution times.

[3]The remaining two test cases are smaller, and CPU has better LLC utilization for these cases. We would also expect better performance if our accelerators were connected to an LLC instead of directly to DRAM.

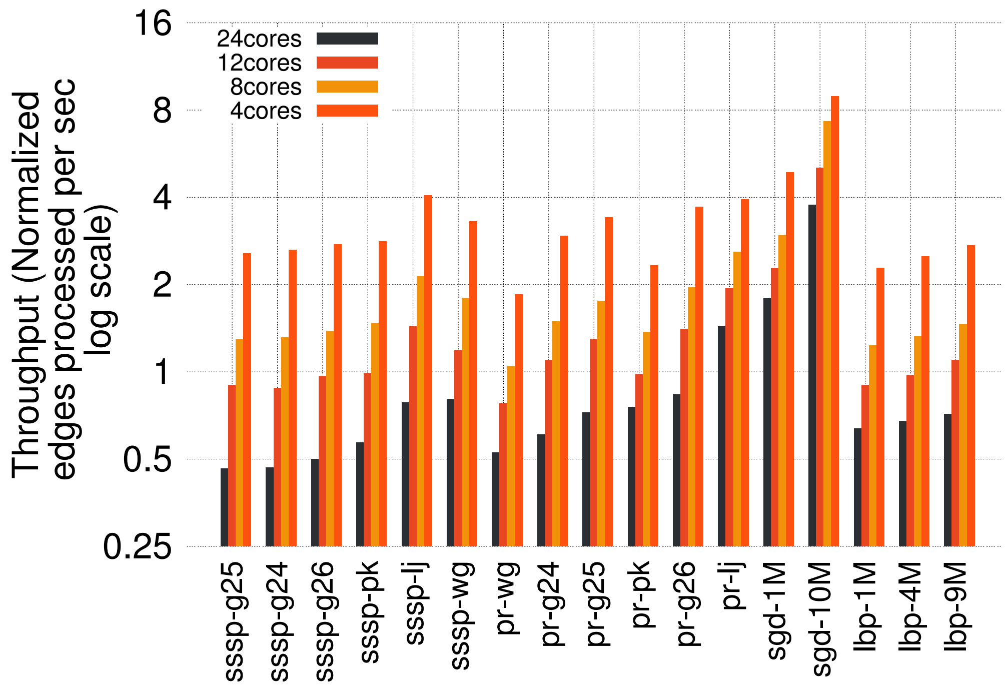


Figure 5.2: Throughput comparisons. The y-axis is the ACC throughput divided by the CPU throughput.

is the sequential consistency support provided in our accelerators. It was shown in [2] that LBP-like applications have much better convergence behaviour when sequential consistency is enabled. However, as shown in [31], implementing sequential consistency on a CPU can slow down the execution by up to an order of magnitude due to extra locking overheads.

For SGD, our accelerators perform better than a 24-core CPU in terms of both execution time and throughput metrics. The reason is the large number of arithmetic operations performed per vertex, which is done more efficiently with custom hardware.

SSSP is the only application where our accelerators do not outperform 24-core performance. The baseline CPU implementation is highly optimized with special data structures that cannot be modelled as a vertex-centric program alone. As future work, such data structures can be added to our accelerator templates. The performance of our accelerators is similar to the performance of 12-core CPU. However, as will be shown in Section 5.2.1.2, our accelerators consume significantly less power than 12 cores.

5.2.1.2 Power Efficiency

Power consumption of our accelerators is dominated by the DDR4 power, which is around 3W for all test cases. This is about 8x higher than the power consumed by the rest of the system, including all accelerator units and cache structures. Other studies have also observed that accelerator power is dominated by DRAM access [50]. However, for CPU executions, core+uncore power consumption is much larger than the projected DDR4 power values.

Figure 5.3 shows the power consumption of the baseline CPU with respect to our accelerators. Note that the CPU power includes core, uncore, and the projected DDR4 power values. The accelerator power includes all accelerator units, caches, and DDR4. As can be seen from this figure, our accelerators have up to 65x better power efficiency compared to the CPU system. Most importantly, even if our SSSP accelerator does not perform as fast as 24 cores (Section 5.2.1.1), we observe about 64x lower power for most of the SSSP test cases.

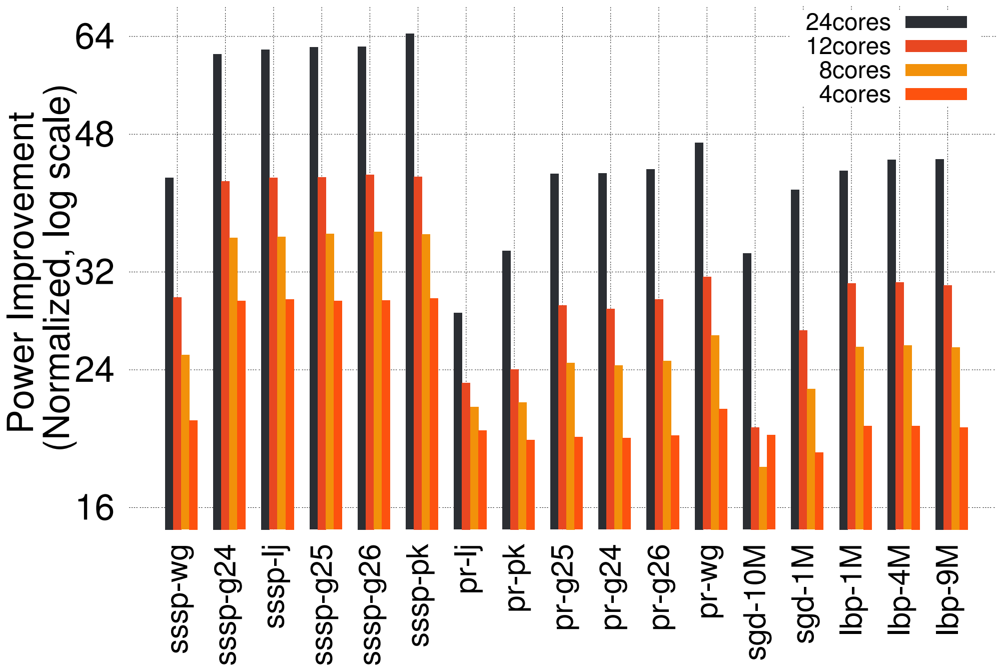


Figure 5.3: Power consumption comparisons. The y-axis is the CPU power divided by the ACC power.

Table 5.3: Power breakdown of accelerator units (in W).

	Pagerank		LBP		SGD		SSSP	
	Power	%	Power	%	Power	%	Power	%
gather	0.029	33	0.045	23	0.438	80	0.008	13
scatter	0.015	16	0.066	34	0.012	2	0.022	39
apply	0.011	12	0.006	3	0.007	1	0.001	2
sync	0.014	16	0.035	18	0.062	11	0.007	13
alm	0.013	15	0.014	7	0.014	3	0.012	22
runtime	0.000	1	0.001	0	0.000	0	0.001	1
crossbar	0.006	7	0.029	15	0.013	2	0.006	10

5.2.2 Area and Power Analysis of Accelerator

In this subsection, we provide the detailed power and area breakdown of accelerator units and cache units. As stated in Section 5.1.1, different applications have different computational requirements. As shown in Tables 5.3 and 5.4, area and power consumption of individual blocks depend on the application. For example, for PageRank and SGD the *gather unit* occupies the most area while the *scatter unit* takes most of the area for LBP. Beside computational units, depending on the application requirements, different basic blocks in the accelerator unit can occupy different areas. For example, LBP and SGD both implement support for *sequential consistency*. Their synchronization unit occupies a larger area than PageRank and SSSP.

In addition to computational units, cache components also depend on the data structures used in the application definition (see Section 4.2.1.7 for the acronyms used for different data structures). Tables 5.5 and 5.6 show the details for each cache unit. The data structure that has the maximum amount of storage has generally the highest amount of power consumption and area. For example, when we consider the PageRank application, vertex data (VD) is the only storage that the application has and we observe that 32% of power consumption belongs to this cache. The same characteristics are also valid for other applications such as LBP and its edge data (ED) cache, SGD and its VD cache. In addition to caches that are used for the application data storage, active list (AL) caches consume significant amount of power and area in our accelerator architecture. Yet, the power consumption of the memory subsystem is still negligible compare to the 3W DRAM power.

Table 5.4: Area breakdown of accelerator units (in mm^2).

	Pagerank		LBP		SGD		SSSP	
	Area	%	Area	%	Area	%	Area	%
gather	0.238	54	0.192	25	0.484	42	0.090	31
scatter	0.096	22	0.247	32	0.101	9	0.121	42
apply	0.030	7	0.010	1	0.012	1	0.005	2
sync	0.032	7	0.244	31	0.504	43	0.032	11
alm	0.030	7	0.029	4	0.029	3	0.029	10
runtime	0.002	0	0.002	0	0.002	0	0.002	0
crossbar	0.011	3	0.051	7	0.024	2	0.010	4

Table 5.5: Power breakdown for cache structures (in W).

	Pagerank		LBP		SGD		SSSP	
	Power	%	Power	%	Power	%	Power	%
VI	0.0019	6	0.0028	4	0.0007	2	0.0015	5
EI	0.0021	7	0.0080	13	0.0038	12	0.0008	3
VD	0.0098	33	0.0108	18	0.0069	23	0.0007	3
ED	0.0000	0	0.0199	33	0.0000	0	0.0000	0
AL	0.0160	54	0.0141	24	0.0191	63	0.0245	89
L/S Unit	0.0000	0	0.0044	7	0.0000	0	0.0000	0

5.2.3 Scalability and Sensitivity Analysis

The default architecture parameters for the proposed accelerators are listed Table 5.1. In this section, we change one parameter at a time and measure the change in performance.

As described in Section 4.2, processing multiple vertices and edges allows us to achieve high levels of memory level parallelism and tolerate long latencies. Figure 5.4 illustrates the performance sensitivity with respect to the number of concurrent edges in Gather and Scatter Units of a single AU. Here, the y-axis value of 1.0 corresponds to the execution time for the parameters in Table 5.1, and values larger than 1.0 correspond to slower executions due to parameter change. Note that a certain number of concurrent vertices and edges are needed to achieve the best performance, after which the performance saturates. This is due to Little's Law, which states that the number of in-flight requests need to be at least throughput times latency to be able to fully utilize the available DRAM bandwidth.

Table 5.6: Area breakdown for cache structures (in mm^2).

	Pagerank		LBP		SGD		SSSP	
	Area	%	Area	%	Area	%	Area	%
VI	0.0105	9	0.0187	5	0.0077	6	0.0082	9
EI	0.0050	4	0.0473	14	0.0095	8	0.0026	3
VD	0.0175	15	0.0647	19	0.0167	14	0.0037	4
ED	0.0000	0	0.1054	30	0.0000	0	0.0000	0
AL	0.0822	72	0.0830	24	0.0849	72	0.0799	84
L/S Unit	0.0000	0	0.0288	8	0.0000	0	0.0000	0

5.2.4 Design Space Exploration

To study the effectiveness of our design space exploration methodology, we have selected the PR application as case study. Pareto curve generated by the heuristic shown in Figure 4.8 is given in Figure 5.5. In this experiment, we swept the α values between 0 and 5.

When we consider $\alpha = 0$, area does not have any impact on the metric $T - \alpha \cdot A$, thus we expect that only throughput will be optimized. On the other hand, larger α values increase the significance of the area term. Smaller area means smaller buffers and caches, and this causes two problems: (1) smaller number of edge slots and vertex rows imply less parallelism in GU and SCU; and it is harder to hide memory access latency, (2) smaller caches increase the number of memory accesses and impose higher memory latency. Therefore, we observe smaller throughput for large α values.

While high throughput values are desirable, different projects may have different design constraints. As shown in Figure 5.5, our heuristic provides a range of design points that a user can select based on specific constraints. However, we can say that design points that reside in the lower right corner of this figure would represent most desirable points in practice.

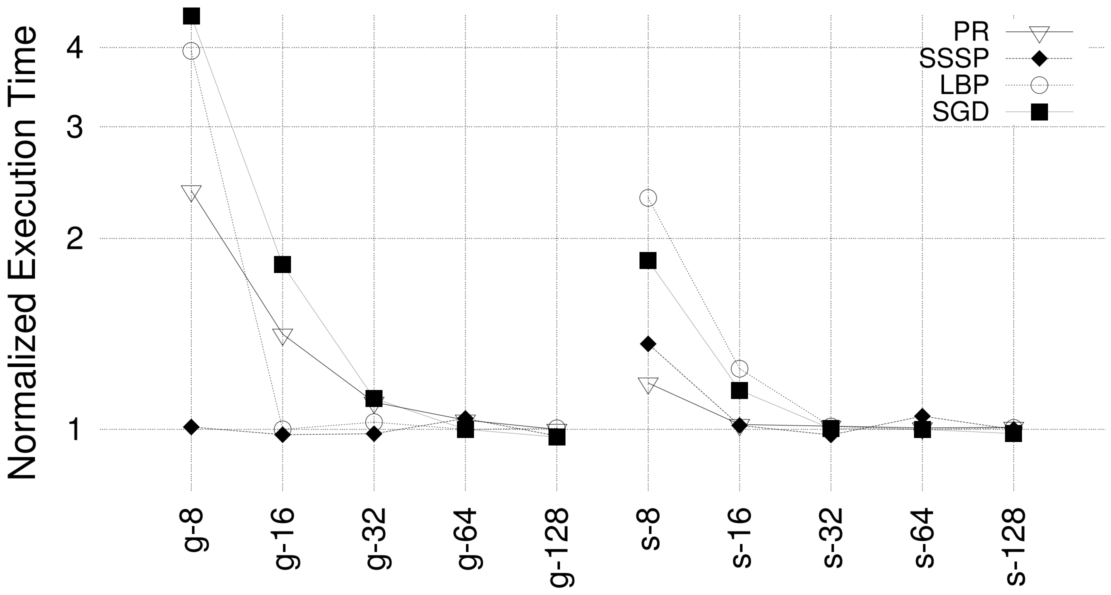


Figure 5.4: Sensitivity analysis for the number of concurrent edges in (g-XX) Gather Unit and (s-XX) Scatter Unit of a single AU (XX is the number of concurrent edges in the corresponding unit).

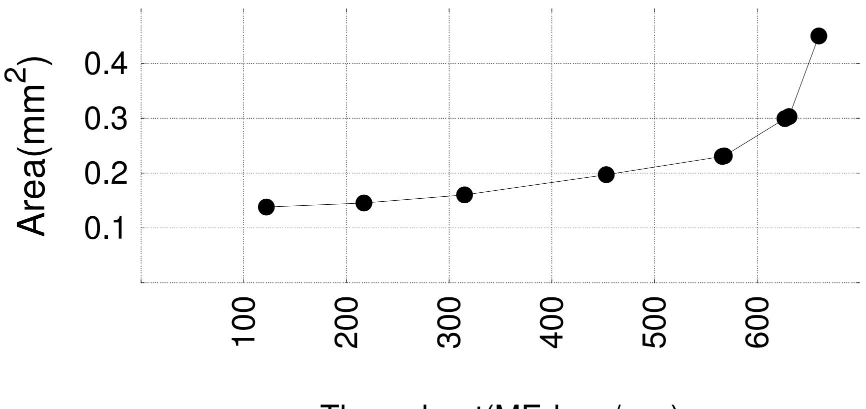


Figure 5.5: Pareto curve generated for PageRank using the proposed design space exploration algorithm.

Chapter 6

Conclusions

While Moore's Law is still valid, we observe that Dennard scaling does not hold. Therefore, chips become more power limited and researchers are trying to find a good solution to overcome the power limitation.

Custom hardware accelerators offer new opportunities to develop highly energy efficient chips. However, implementing these custom hardware requires both hardware design and application domain knowledge.

In this work, we propose a template based solution in order to free the programmer from the burden of hardware design. The proposed template enables fast custom hardware accelerator design and exploration. The main contributions of this work can be summarized as follows: (1) We analyse the work efficiency that can be extracted from different characteristics of graph applications. (2) We propose a template and an accelerator architecture targeted for iterative, vertex-centric graph applications with irregular execution and memory access patterns. (3) We provide a detailed experimental study which compares generated hardware with a high end 24 core CPU system. (4) We provide an automated design space exploration methodology which enables programmers to optimize storage parameters without detailed knowledge of the underlying hardware.

Furthermore, we show that the proposed accelerator provides similar or better performance compared to a 24 core high end CPU system. It is observed that 3x performance gains can be achieved in specific applications. Additionally, the proposed template can generate significantly smaller and more power efficient hardware by up to a factor of 65x.

Although our work is limited with graph analytics applications, similar techniques can be extended to different applications. The template is only able to generate fixed function accelerators. However, future work can apply similar concepts for programmable hardware. Moreover, the proposed template can be implemented on FPGAs.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[4] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, June 2011.

[5] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 979–990, ACM, 2014.

[6] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proc. of the Seventeenth International Conference*

on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012.

[7] "Giraph." http://giraph.apache.org/.

[8] "The combinatorial BLAS: Design, implementation, and applications." http://gauss.cs.ucsb.edu/ aydin/combblas-r2.pdf.

[9] M. S. Lam, S. Guo, and J. Seo, "Socialite: Datalog extensions for efficient social network analysis," in *Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, (Washington, DC, USA), pp. 278–289, IEEE Computer Society, 2013.

[10] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," *SIGPLAN Not.*, vol. 46, pp. 267–276, Feb. 2011.

[11] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *SIGMOD Rec.*, vol. 43, pp. 35–40, Dec. 2014.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Efficient and simplified parallel graph processing over CPU and MIC,"

[13] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proc. VLDB Endow.*, vol. 4, pp. 231–242, Jan. 2011.

[14] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, and H.-D. Nguyen, "Parallel pagerank computation using GPUs," in *Proc. of the Third Symposium on Information and Communication Technology*, SoICT '12, (New York, NY, USA), pp. 223–230, ACM, 2012.

[15] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on GPUs," in *Proc. of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, (New York, NY, USA), pp. 81–89, ACM, 2015.

[16] A. Davidson, S. Baxter, M. Garland, and J. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 349–359, May 2014.

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

[18] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 141–151, Nov 2012.

[19] M. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular gpu kernels," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 130–139, Oct 2014.

[20] S. McGettrick, D. Geraghty, and C. McElroy, "An fpga architecture for the pagerank eigenvector problem," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 523–526, Sept 2008.

[21] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "Cygraph: A reconfigurable architecture for parallel breadth-first search," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 228–235, May 2014.

[22] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on fpga for breadth-first search," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 70–77, Dec 2010.

[23] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pp. 8–15, July 2012.

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

[25] G. R. Jagadeesh, T. Srikanthan, and C. M. Lim, "Field programmable gate array-based acceleration of shortest-path computation," *IET Computers Digital Techniques*, vol. 5, pp. 231–237, July 2011.

[26] J. Y. Kim and C. Batten, "Accelerating irregular algorithms on GPGPUs using fine-grain hardware worklists," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 75–87, Dec 2014.

[27] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[28] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin, "Graphgen: An FPGA framework for vertex-centric graph computation," in *Proc. of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, (Washington, DC, USA), pp. 25–28, IEEE Computer Society, 2014.

[29] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, J. Knight, T.F., and A. DeHon, "Graphstep: A system architecture for sparse-graph algorithms," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pp. 143–151, April 2006.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *CoRR*, vol. abs/1204.6078, 2012.

[31] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Architectural requirements for energy efficient execution of graph analytics applications," in *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 2015.

[32] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, Apr. 1998.

[33] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June 2014.

[34] A. Arasu, J. Novak, J. Tomlin, and J. Tomlin, "Pagerank computation and the structure of the web: Experiments and algorithms," 2002.

[35] D. A. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *2005 International Conference on Parallel Processing (ICPP'05)*, pp. 547–556, June 2005.

[36] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pp. 56–65, October 2015.

[37] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, pp. 12–25, June 2011.

[38] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proc. of the 43Rd Annual International Symposium on Computer Architecture, to appear*, ISCA '16, IEEE, 2016. ©2016 IEEE, Reprinted, with permission.

[39] "Cacti." `http://www.hpl.hp.com/research/cacti`.

[40] "Gap benchmark suite code." https://github.com/sbeamer/gapbs.

[41] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), pp. 69–77, ACM, 2011.

[42] "Running average power limit." `https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl`.

[43] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, 2011.

[44] M. Bohr, "Silicon technology leadership for the mobility era," in *IDF*, 2012.

[45] M. Bohr, "14nm process technology: Opening new horizons," in *IDF*, 2014.

[46] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," in *IEEE Micro*, 2011.

[47] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the power wall: A path to exascale," in *Proc. of Supercomputing*, 2014.

[48] "Graph500." `www.graph500.org`.

[49] "Movielens dataset." http://grouplens.org/datasets/movielens/.

[50] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, pp. 269–284, Feb. 2014.