Imperial College London
Department of Computing

Reconfigurable Computing for Large-Scale Graph Traversal Algorithms

Brahim Betkaoui

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London

Abstract

This thesis proposes a reconfigurable computing approach for supporting parallel processing in large-scale graph traversal algorithms. Our approach is based on a reconfigurable hardware architecture which exploits the capabilities of both FPGAs (Field-Programmable Gate Arrays) and a multi-bank parallel memory subsystem. The proposed methodology to accelerate graph traversal algorithms has been applied to three case studies, revealing that application-specific hardware customisations can benefit performance. A summary of our four contributions is as follows.

First, a reconfigurable computing approach to accelerate large-scale graph traversal algorithms. We propose a reconfigurable hardware architecture which decouples computation and communication while keeping multiple memory requests in flight at any given time, taking advantage of the high bandwidth of multi-bank memory subsystems.

Second, a demonstration of the effectiveness of our approach through two case studies: the breadth-first search algorithm, and a graphlet counting algorithm from bioinformatics. Both case studies involve graph traversal, but each of them adopts a different graph data representation.

Third, a method for using on-chip memory resources in FPGAs to reduce off-chip memory accesses for accelerating graph traversal algorithms, through a case-study of the All-Pairs Shortest-Paths algorithm. This case study has been applied to process human brain network data.

Fourth, an evaluation of an approach based on instruction-set extension for FPGA design against many-core GPUs (Graphics Processing Units), based on a set of benchmarks with different memory access characteristics. It is shown that while GPUs excel at streaming applications, the proposed approach can outperform GPUs in applications with poor locality characteristics, such as graph traversal problems.

Declaration of Originality

This document consists of research work conducted at Imperial College London in the Department of Computing, between October 2009 and December 2013. I declare that the work presented is my own, except where specifically acknowledged in the text.

Copyright Declaration

Acknowledgements

I would like to express my deepest and sincere thanks to my supervisors Professor Wayne Luk and Dr. David B. Thomas, under whom I had the privilege to work towards my PhD degree at Imperial College London. I would like to thank Professor Wayne Luk , in particular, for his guidance, support, and his extraordinary patience and trust in me during my PhD studies. I am also very grateful for David's excellent mentorship, extremely helpful suggestions, and great discussions during our weekly meetings.

Dr Natasa Przulj, Dr Oleksii Kuchaiev, and Omer Nebil Yaveroglu have been particularly helpful through their collaborations on the work in this thesis and papers on graphlet counting (Chapter 4). In addition to his collaboration on the APSP problem in Chapter 5, I would like to thank Professor Wang Yu for his many helpful comments and suggestions on how to improve my work on both BFS (Chapter 3) and APSP (Chapter 5).

I would like to thank members of the Custom Computing research group for general help towards completing my PhD, and for making my PhD time in the group enjoyable through discussions of interesting topics.

Moreover, I would like to thank my family to which I would like to dedicate this thesis. I would like to express my greatest gratitude to my parents for their support throughout my PhD studies. I would also like to thank my sister Radia for her encouragement, motivation, and advice having walked the PhD road before me.

Last but no least, a special thank to my examiners for their diligent reading of the text and suggested improvements.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In many application domains such as social networks, genomics, data mining, and bioinformatics, large datasets have been represented as large graphs or networks involving millions of vertices and billions of edges. These graph data are processed using graph traversal algorithms to discover vertices, paths, and various properties of graphs. These graph traversal algorithms are becoming more important as the graph problems grow in size and become challenging to process efficiently. For example, recent technological advances in experimental biology have yielded large amounts of protein-protein interaction (PPI) data. It has been reported in [4] that comparing 5 viral PPI networks against different network models takes more than 4 hours on an Intel Core i3 Quad-core CPU with 4GB of main memory. Moreover, the well-known graph traversal algorithm, Breadth-First Search (BFS), has been proposed recently [5] as a benchmark metric to measure and rank the performance of HPC (High- Performance Computing) systems for data-intensive HPC applications. Hence, a solution with a strong computational capability to process large-scale graph traversal algorithms will greatly benefit many application domains.

Large-scale graph traversal problems have a number of properties that make them poorly matched to computational methods applied in mainstream parallel applications [6]. Large-scale graph problems typically involve accessing high-latency off-chip memories. In addition, graph traversal algorithms tend to explore the structure of the graph while performing little arithmetic computations per memory access leading to a high data access to data compute ratio. Furthermore, the graphs are typically unstructured and highly irregular leading to spatially incoherent or random memory accesses. Irregular graph structures coupled with a high data access to data

compute ratio leads to execution times being dominated by memory latency.

These properties pose several performance challenges when these graph traversal algorithms are mapped onto conventional microprocessor-based HPC systems. In recent years clusters built from commodity cache-based microprocessors have been a common choice for HPC, as they are cheap to purchase and operate. Conventional cache-based microprocessors rely on a hierarchical memory subsystem to take advantage of high spatial and/or temporal memory access locality characteristics within an application. However, as we mentioned above, graph traversal algorithms have low memory access locality characteristics, rendering cache memories ineffective. The latency of main memory is difficult to hide due to the spatially incoherent nature of the memory accesses caused by the irregular structure found in a typical real-world graph. In addition, graph traversal algorithms perform little arithmetic operations per memory access leading to a high data access to compute ratio. This means that latency hiding techniques that rely on executing a large number of compute (arithmetic) operations between memory accesses are not effective either.

These performance challenges have recently led to a great interest in exploring heterogeneous computer systems [7, 8, 9, 10, 3, 11, 12, 13, 14], where CPUs are augmented with specialised hardware that can accelerate graph traversal kernels. Examples of specialised hardware include GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays). Previous work has shown that FPGA-based reconfigurable computing machines can achieve orders of magnitude of performance speed-ups compared to CPUs for many important computing applications [15]. However, one limitation of FPGAs that has prevented widespread usage is the requirement for software developers to learn a whole new set of skills and hardware design concepts, and application development for FPGA-based accelerators takes more time than producing a software version that runs on modern CPUs. In addition, FPGA-based acceleration solutions require regular or predictable memory access patterns (e.g. sequential access of data from memory) due to the heavily pipelined circuits in FPGA implementations. Applications with irregular memory access patterns, such as graph traversal algorithms, achieve a low random access memory bandwidth due to the increased number of page misses in DRAM memories. Consequently, this low random memory access bandwidth incurs many pipeline stalls, resulting in little acceleration from the FPGA, and possibly even deceleration.

1.2 Objectives

The primary goal of the presented work is to investigate the possible acceleration of graph traversal algorithms for large-scale graph problems using FPGA-based reconfigurable computing. In this work, the term large-scale

is used to refer to graph problems that are too large for on-chip memories, and hence, require the graph data to be stored in off-chip memories (i.e. DRAM). The size of such graphs is only limited by the storage capacity of main memory in a shared memory system. To achieve our primary goal we set the following objectives:

1. To explore reconfigurable computing in accelerating graph traversal algorithms for large-scale graph problems.
2. To develop hardware implementations which are evaluated using common graph traversal benchmarks for comparison with multi-core CPUs and GPUs.
3. To evaluate the performance of FPGAs for high-productivity computing using an FPGA-based HPC system where a CPU is augmented with a specialised FPGA co-processor, allowing the CPU and FPGA to co-operate closely while providing a programming model similar to that of traditional software.

The next section shows how these objectives are achieved.

1.3 Thesis and contributions

The work presented in this thesis proposes a reconfigurable computing approach for efficient parallel processing of large-scale graph traversal algorithms. Our reconfigurable computing approach is based on a reconfigurable hardware architecture which exploits the hardware capabilities of both FPGAs and a multi-bank parallel memory subsystem. The presented methodology to accelerate graph traversal algorithms has been applied to three case studies, revealing that application-specific hardware customisations can significantly improve performance. A summary of our main contributions is provided below, while several other contributions are listed in the introductory sections of some chapters in this thesis.

1. A reconfigurable computing approach to accelerate large-scale graph traversal algorithms is proposed. The proposed approach is based on a reconfigurable hardware architecture which decouples computation and communication while keeping multiple memory requests in flight at any given time, and so takes advantage of the high bandwidth of multi-bank memory subsystems. The effectiveness of this approach is demonstrated using the well-known Breadth-First Search (BFS) problem. Performance results show that our FPGA-based BFS design outperforms recent work in the literature, and that our BFS design was ranked 45th in the June 2012 list of the Graph500 benchmark [1], beating in the process other FPGA-based systems as well as multi-node CPU and GPU systems (Chapter 3).

2. A new hybrid graphlet counting algorithm that greatly improves performance over the conventional graphlet counting algorithm for both cache-based CPU systems and FPGA-based systems, while significantly reducing the amount of memory storage requirements. In order to process large graphs, the proposed hybrid graphlet counting algorithm uses a pseudo-adjacency-matrix representation that reduces memory storage requirements by up to 90% (Chapter 4).

3. We show how on-chip memory resources in FPGAs can be used to reduce off-chip memory accesses to accelerate graph traversal algorithms through a case-study of the well-known All-Pairs Shortest-Paths (APSP) problem. This case study has been applied to process human brain network data. Performance results show that our FPGA-based APSP solver outperforms both multi-core CPUs and GPUs for sparse networks (Chapter 5).

4. Performance comparison of FPGAs against multi-core CPUs and GPUs for high-productivity computing, using a set of benchmarks with different memory access characteristics. We also provide a discussion of the advantages and disadvantages of using FPGAs and GPUs for high-productivity computing (Chapter 6).

These contributions address the objectives mentioned in Section 1.2. The first objective is achieved mainly by the first contribution as well as the third contribution. The second objective is met through the hardware implementations and their evaluation covered by the first three contributions targeting three graph algorithms, namely BFS, graphlet counting, and APSP. Finally, the last objective is achieved through the fourth contribution.

1.4 Structure of this thesis

Chapter 2 provides background material as well as a literature review of related work for this thesis. The first part of this chapter covers graph theory and challenges in parallel processing of large-scale graph problems. The second part of this chapter survey previous work on efficient parallel processing of graph traversal algorithms on multi-core CPUs, GPUs, and FPGAs.

Chapter 3 presents a reconfigurable computing approach for efficient processing of large-scale graph traversal algorithms. This approach is based on a novel reconfigurable hardware architecture which decouples computation and communication while keeping multiple memory requests in flight at any given time. We demonstrate the effectiveness of our reconfigurable computing approach using the well-known Breadth-First Search algorithm. We provide a detailed description of our hardware design for BFS acceleration. This is followed by

design optimisations and their impact on performance. Finally, performance evaluation of our approach on an actual hardware implementation is provided including a performance comparison with multi-core CPU and GPU implementations from related work.

Chapter 4 presents a reconfigurable hardware accelerator of a common bioinformatics algorithm, namely graphlet counting. A new hybrid graphlet counting algorithm is proposed to improve the performance of the graphlet counting algorithm on both cache-based CPU systems as well as FPGA-based systems. We also propose a hybrid graph data representation to perform graphlet counting on large graphs. We then describe how we parallelise the graphlet counting algorithm using our reconfigurable hardware architecture presented in Chapter 3. An evaluation of the proposed hardware design is provided by comparing the performance of our design to an optimised multi-core CPU implementation.

Chapter 5 presents another case study of a common graph problem, All-Pairs Shortest Paths (APSP) which, unlike BFS and graphlet counting, makes use of on-chip memory resources to accelerate the APSP problem. A detailed description of an FPGA-based APSP solver, including a novel way to leverage the power of distributed memory resource on FPGAs to reduce memory bandwidth requirements, is proposed in this chapter. Next, design optimisations are presented as well as their impact on performance. We also provide an analytical model to estimate the maximum amount of achievable parallelism for a given input graph size and target FPGA device. The chapter is concluded with an in-depth performance evaluation that includes performance comparison to multi-core CPUs and GPUs from the literature.

Chapter 6 presents a comparative study of FPGAs and GPUs for high-productivity computing. This chapter begins with the characterisation of productivity before introducing a set of benchmarks with different memory characteristics that are used in this study. The FPGA-based and GPU-based system are then presented, and a discussion of how these benchmarks are implemented and optimised for each platform is provided. Finally, a discussion of the advantages and disadvantages of using FPGAs and GPUs for high-productivity computing is provided.

Finally, Chapter 7 summarises our work and explains how well this work achieves our goals, and provide a discussion of possible future research directions.

1.5 Publications resulting from this work

The following is a list of publications by the author of this thesis, related to the work contained in this thesis.

1.5.1 Conference papers

- B. Betkaoui, Y. Wang, D. B. Thomas, W. Luk, *Parallel FPGA-based all pairs shortest paths for sparse networks: a human brain connectome case study*, 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL'12), Oslo, Norway, August 2012.
- B. Betkaoui, Y. Wang, D. B. Thomas, W. Luk, *A reconfigurable computing approach for efficient and scalable parallel graph exploration*, 23rd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'12), Delft, The Netherlands, July 2012.
- B. Betkaoui, D. B. Thomas, W. Luk, N. Przulj, *A framework for FPGA acceleration of large graph problems: graphlet counting case study*, International Conference on Field-Programmable Technology (FPT'11), New Delhi, India, December 2011.
- B. Betkaoui, D. B. Thomas, W. Luk, *Comparing performance and energy efficiency of FPGAs and GPUs for high-productivity computing*, International Conference on Field-Programmable Technology (FPT'10), Beijing, China, December 2010.

1.5.2 Short paper

- B. Betkaoui, D. B. Thomas, W. Luk, *Exploring hybrid-core computing for option pricing applications*, International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2010), Epochal Tsukuba, Tsukuba, Japan, June 2010.

Chapter 2

Background

This chapter presents background material and related work for this thesis. Section 2.1 contains an introduction to graph theory, real-world graph problems, and challenges in parallel processing of graph problems. In Section 2.2, we provide an overview of current parallel computer architectures for parallel processing of graph traversal problems. We review the recent developments of efficient processing of graph traversal algorithms on various types of computing systems in Section 2.3. Finally, Section 2.4 provides our conclusion that results from our review of the recent development in parallel processing of large-scale graph algorithms.

2.1 Graph Problems

2.1.1 Graph theory

Notation

A *graph* is a collection of points with lines connecting pairs of points. The points are called *nodes* or *vertices*, and the lines are called *edges*. Figure 2.1 shows an example of a graph with 6 nodes or vertices and 7 edges. A graph is usually denoted by G, or by $G(V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges of G. We often use $|V|$ to represent the number of vertices, and $|E|$ to represent the number of edges. We also use V to represent the set of vertices of a graph G, and E to represent the set of edges of a graph G. A graph is *undirected* if its edges (node pairs) are undirected, and otherwise it is *directed* [16].

Vertices joined by an edge are called *adjacent* vertices. A *neighbour* of a vertex v is a vertex adjacent to v. We denote by $N(v)$ the set of neighbours of vertex v, and by $N[v]$ the *closed neighbourhood* of v, which is defined

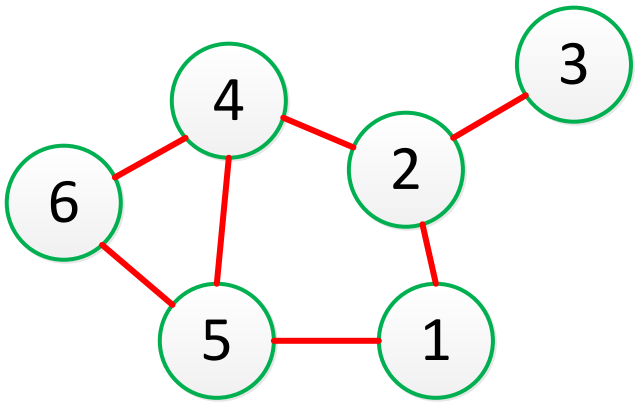


Figure 2.1: An undirected graph with 6 vertices (green numbered circles) and 7 edges (red straight lines linking the vertices).

as $N[v] = N(v) \cup \{v\}$. The *degree* of a vertex is the number of edges incident with the vertex. A *subgraph* of G is a graph whose all vertices and edges belong to G. An *induced subgraph* H of G, denoted by $H \triangleleft G$, is a subgraph of G on $V(H)$ vertices, such that $E(H)$ consists of all edges of G that connect vertices of $V(H)$.

Representation of graph data

There are two standard graph data representations $G = (V, E)$ [17]: (i) a collection of adjacency lists with a total size of $|E|$ elements, or (ii) an adjacency matrix with $|V|^2$ elements. Both representations apply to both directed and undirected graphs. The former representation is more common as it provides a compact data representation of sparse graphs, for which $|E|$ is much less than $|V|^2$. However, there are cases where the adjacency-matrix representation is preferred over the adjacency-list representation. For example, when the graph is very dense, i.e. $|E|$ is close to $|V|^2$, or when we need to quickly test the existence of an edge connecting two vertices. Note that there are some algorithms that use both representations as in the case of the graphlet counting algorithm [18].

(A) Adjacency-list representation

The adjacency-list representation of a graph $G = (V, E)$ consists of a collection of vertex lists, one for each vertex in $|V|$. For each $v \in V$, the adjacency list of v consists of all the vertices u such that $(v, u) \in |E|$. That is the u vertices are the adjacent vertices of v in G. The compressed sparse row (CSR) sparse matrix format is commonly used for the adjacency-list representation. Figure 2.2 illustrates this concept. The array C is formed by concatenating the set of adjacency lists into a single array of $|E|$ elements. The row-offset R stores the

$|V| + 1$ indices that point to the beginning of each adjacency list in C, such that $R[i]$ is the index in C of the adjacency list of vertex v_i.

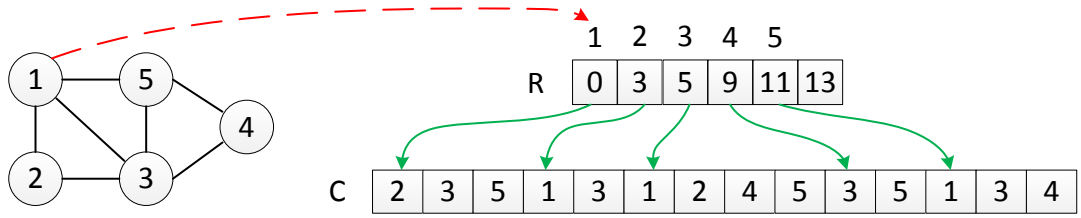


Figure 2.2: Adjacency-list representation of graph data.

(B) Adjacency-matrix representation

A drawback of the adjacency-list representation may be that it does not provide an efficient method to determine whether a given edge (u, v) is present in the graph other than performing an expensive search for u in the adjacency list of v. This search becomes expensive when the size of the adjacency list of v approaches $|V|$ vertices. A better approach for testing the existence of edges in a graph is to use the adjacency-matrix representation of the graph, albeit at the cost of using much more memory for data storage.

Assuming that the vertices in a graph $G = (V, E)$ are numbered 1,2,....,$|V|$, then the adjacency-matrix of G is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

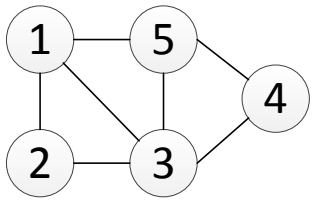
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Figures 2.3 illustrates the adjacency-matrix representation of a graph. The adjacency-matrix of a graph $G(V, E)$ requires $\mathcal{O}(|V|^2)$ memory, regardless of the number of edges $|E|$ in the graph.

2.1.2 Real-world graphs

In many application domains, data are represented using large graphs involving millions of vertices and billions of edges. Many real-world graphs are large and require significant computational and memory resources to search and store. The following are examples of real-world large graphs:

- Social Networks [19] - Graphs naturally model the relationships established by social interactions. These interactions could be on-line friendships, call-networks, etc.



	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	1	0	1	1	0

Figure 2.3: Adjacency-matrix representation of graph data

- Biological networks - Recent technological advances in experimental biology have yielded large amounts of biological network data. For example, Protein-Protein Interaction networks for mammals are expected to be very large, with humans PPI networks projected to have around 120,000 proteins with 10^6 PPIs [20]. Recent studies have been investigating associations between diseases and network topology in PPI networks and have shown that disease genes share common topological properties. Finding this relationship between PPI network topology and biological function and disease remains one of the most challenging in the post-genomic era [21].

- World Wide Web (WWW) graph [22] - Graphs that model the structure of the web often consist of vertices representing webpages and directed edges representing the hyperlinks between the webpages.

In addition, real-world graphs have typically the following properties:

- Power law - A common property of many real world graphs is a power law distribution of vertex degree. An effect of the power law degree distribution is that while the vast majority of vertices have a low degree, a select few vertices will have a very high degree. In literature, this property is often referred to as *scale-free* and can lead to a significant load imbalance when processing a scale-free graph in parallel.

- Small diameter or small world property - Although sparse, many graphs are connected into giant connected components with small diameters. The diameter of a graph is the longest shortest-path between any two vertices in the graph.

- Community Structure - Vertices that group into interconnected clusters are called communities. In a cluster, there are more interconnected edges than outgoing edges. This property leads to clusters which are highly interconnected while having only few connections outside of the group.

2.1.3 Challenges in parallel processing of graph problems

Graph problems have some inherent characteristics that make them poorly matched to current computational problem-solving approaches. In particular, the following properties of graph problems present significant challenges for efficient parallel processing [6].

- **Data-driven computations.** Generally, graph computations are dictated by the vertex and edge structure of the graph, and the execution paths are difficult to analyse and predict using static analysis of the source code. Parallelism based on partitioning the computation is a challenging task due to lack of knowledge about the structure of the computations.
- **Irregular structure of graph data** Graphs are usually unstructured and highly irregular, making it difficult to partition graph data to take advantage of small and fast on-chip memories, such as cache memories in GPUs and on-chip RAMs in FPGAs.
- **Poor locality.** Data-driven computations coupled with irregular data structures results in low locality accesses to memory. This often leads to suboptimal performance levels on conventional cache-based processors, which rely on high spatial and/or temporal locality of data access in memory.
- **High data access to computation ratio.** Many graph algorithms tend to explore the structure of the graph while performing a relatively small amount of computations. This results in a higher ratio of data access to computation compared to mainstream scientific and engineering applications, and combined with poor locality characteristics leads to execution times dominated by random memory access bandwidths.

2.2 Parallel Architectures for processing of graph traversal problems

2.2.1 Shared memory machines

Efficient single-node processing of large graph problems are drawing more attention since an efficient single-node implementation will also allow multi-node (or computing cluster) implementations to achieve better performance as the per-node performance increases. In a shared-memory system, global memory is universally accessible by each processor. Such shared memory systems can be classified in several different ways. Lumsdaine *et al.* [6] identified two main classes of shared memory computers: cache-coherent machines and massively multi-threaded machines.

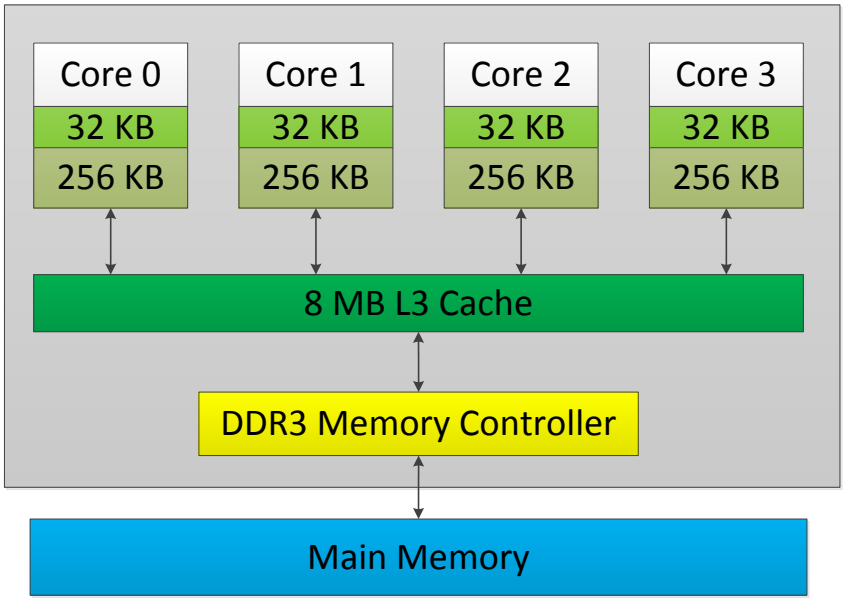


Figure 2.4: Organisation of the memory hierarchy in a quad-core CPU with three levels of cache memory.

Cache-coherent multi-core CPU systems

In cache-coherent multi-core CPU systems, global memory is universally accessible by each processor. Modern multi-core CPUs address the memory latency challenge with a hierarchical memory organisation, which employs small and fast on-chip cache memories as shown in Figure 2.4. However, such CPUs have some inherent performance limitations. These processors have a memory hierarchy in which a small amount of data is kept in faster but smaller on-chip memories, or *cache*, for quick access. When a datum is updated, the new value may not be immediately propagated to main memory (off-chip memory), but may instead be only present in cache for a while. When only a single processor is involved, it is not difficult to ensure that a read operation gets the most current value. But in a multiprocessor machine with multiple caches, the cache-coherence problem is a significant challenge. There are a variety of methods to address this problem, each with its advantages and disadvantages. However, each approach adds overhead which can degrade performance. Even for problems in which reads are much more prevalent than writes, cache coherence protocols have an impact on scalability. A second performance challenge for cache-based systems is applications with poor data access locality as in the case of graph traversal problems. Cache memories are only useful when there is enough spatial and/or temporal data locality within an application. Otherwise, caching becomes ineffective leading to poor performance results.

Massively multi-threaded systems

The massively multi-threaded approach, such as in the Cray XMT machine [23], addresses the latency challenge in a substantially different manner than cache-coherent multi-core CPU systems. Instead of trying to use cache memories to reduce the latency of memory accesses, the Cray XMT tries to tolerate latency by employing a cache-less memory subsystem and by ensuring that a processor has other work to do while waiting for a memory request to be serviced. Each processor can have a large number of outstanding memory requests. The processor has hardware support for many concurrent threads and is able to switch between them in a single clock cycle. Thus, when a memory request is issued, the processor can immediately switch its attention to another thread whose memory request has arrived. In this way, the processor tolerates latency and is not stalled waiting for memory, given sufficient concurrency in the application [6].

However, massively multi-threaded machines also have significant drawbacks. Because the processors are custom and not commodity, they are more expensive and have a much slower clock than mainstream microprocessors. Furthermore, the programming model of the Cray XMT is non-standard. The combination of high cost, non-standard programming model, and lower peak performance numbers for compute-bound computations have hindered the widespread of such machines. Nonetheless, the machines unique ability to handle large-scale graph problems has given researchers useful insights on how future architectures, in particular, on how the memory subsystem should be designed for graph traversal problems.

2.2.2 Distributed memory machines

The most widespread class of parallel machines are distributed memory computers. These machines are usually made predominantly of commodity parts, consisting merely of a set of processors and memory connected by some high speed network. The commodity nature of these machines makes them inexpensive, and they are very effective on many scientific problems and on problems that are trivially parallel. Distributed memory systems can be considered an extension to shared-memory systems as they can be obtained by connecting a number of shared-memory systems. In terms of problem size, distributed memory systems can be used to address problems sizes that are too large to fit into the main memory of a shared memory system.

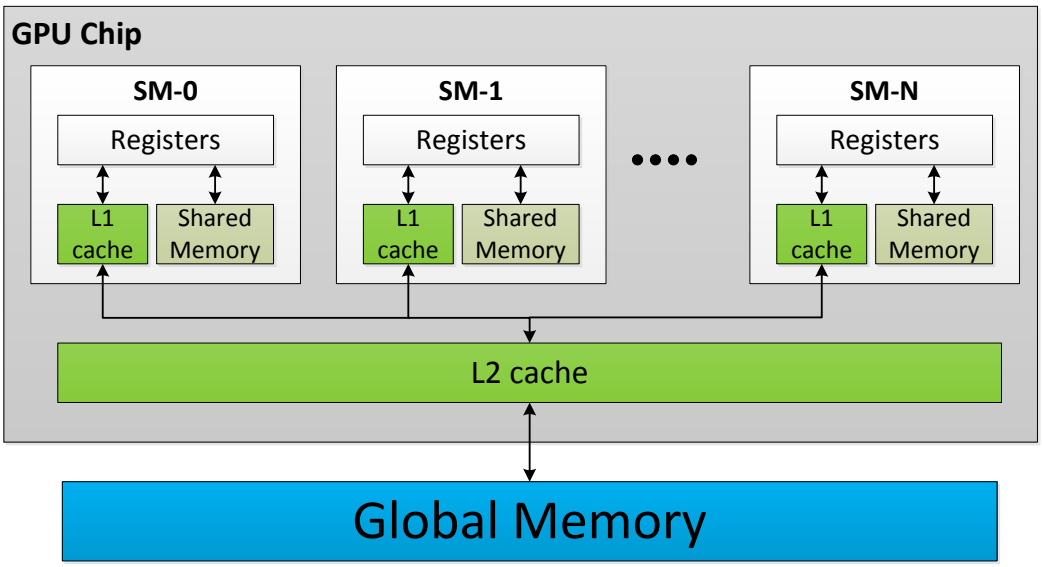


Figure 2.5: The memory hierarchy in a Fermi CUDA-based GPU.

2.2.3 GPU-based systems

In November 2006, NVIDIA introduced CUDA [24], a general purpose parallel computing architecture - with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. Modern NVIDIA GPUs are fully programmable many-core chips built around an array of parallel processors which are called Streaming Multiprocessors (SM). This is illustrated in Figure 2.5. Each SM is capable of supporting up to 1024 co-resident concurrent threads. NVIDIA's current products range in size from 1 SM at the low end to 30 SMs at the high end. A single CUDA SM contains 8 Scalar Processors (SP), each with 1024 32-bit registers, for a total of 64KB of register space per SM. Each SM is also equipped with a 64KB of shared memory and L1 cache. Newer versions of GPUs have more than one level of cache memory, making them similar to modern multi-core CPUs, but also to massively multi-threaded machines since thousands of threads can be executed on modern GPUs. A big drawback of modern GPUs is the size of main memory in GPU devices, which is much smaller than that of shared-memory systems.

2.2.4 Reconfigurable computing

Reconfigurable computing technology has been introduced to fill the gap between hardware and software based design [25]. The aim is to improve the performance of software solutions, while reducing power consumption and maintaining a great deal of flexibility through customisation. Reconfigurable computing devices consist of a

number of functional elements connected by programmable interconnect resources. These functional elements, also known as logic elements (LEs) or processing units, implement a specific function through programmable configurations. In this context, *configuration* is the process of mapping logic functions to hardware blocks on a reconfigurable device, and connecting these blocks with programmable interconnect.

In terms of hardware architecture, reconfigurable computing is fundamentally based on existing technologies: FPGAs and general purpose processors (GPPs). Occasionally, the term 'reconfigurable computing' has been used for systems which are virtually made of only FPGAs, and that barely use any GPPs, such as the Berkeley Emulation Engine [26]. An attempt to classify the variety of high performance reconfigurable computing architecture types is presented in [27], where authors identified the following five categories of HPRC architectures: (i) loosely-coupled coprocessors, (ii) tightly-coupled coprocessors, (iii) integrating FPGAs with network fabric, (iv) direct memory connection, (v) and SRC's IMPLICIT+EXPLICIT model.

2.3 Recent developments in efficient processing of large-scale graph problems

The importance of efficient processing of large-scale graph traversal problems has been increasing as datasets quickly grow past the compute capacity of current high performance computing solutions. This has motivated a substantial amount of previous work that deals with the design and optimisation of graph traversal algorithms, in particular Breadth-First Search (BFS) designs, either for commodity processors [28, 7, 29, 30, 8, 31, 32], or for dedicated hardware [33, 34, 11, 35, 14].

2.3.1 Shared-memory systems

Cache-coherent machines

In [29], an efficient BFS implementation for the Cell/BE processor is proposed. Inspired by the Bulk-Synchronous Parallel (BSP) methodology, the proposed methodology by Scarpazza *et al.* [29] is based on a high-level algorithmic design that focuses on both machine-independent aspects as well as machine-dependent optimisations for the Cell/BE processor. The authors in [29] suggest that high performance comes at the price of increased hardware complexity or human effort (roughly expressed in lines of code) required to develop the software application on the Cell/BE processor, and hence programming effort to achieve optimal performance is still very high. A major strength of the Cell/BE is the possibility of explicitly managing the memory traffic by pipelining multiple DMA requests to the main memory. This a unique feature that is not available on other

conventional multi-core processors which cannot efficiently handle working datasets that do not fit entirely in the cache memory. A major limitation for the Cell/BE processor is the extraction of the SIMD parallelism, which is a non-trivial effort when multiple activities are running concurrently.

Agarwal *et al.* [30] developed a simple and scalable BFS implementation for multi-core CPU systems with shared-memory systems, which can efficiently parse graphs with billions of vertices and edges. Their approach employs an innovative data layout that enhances memory locality and cache utilisation by defining a hierarchy of working datasets. They also presented several machine-dependent optimisations that target the latest Intel Nehalem processors. Agarwal2010 *et al.* [30] also propose a low-latency channel mechanism for inter-socket communication to tolerate expected high delays incurred due to cache-coherent protocols. Using a quad-socket 8-core CPU-based system (32 cores in total), the performance results reported in [30] outperformed previous work on BFS implementations for large graph instances, including special-purpose supercomputers designed to handle irregular applications [35, 36].

In [8], Hong *et al.*present a BFS implementation for CPUs and GPUs which exploits a fundamental property of randomly shaped real-world graph instances. They also propose a hybrid CPU-GPU method in which for each level of the BFS algorithm, one of the following implementations is dynamically selected: a sequential CPU execution, two different methods of parallel multi-core execution, and a GPU execution. Hong *et al.* [8] claim that their methodology outperforms state-of-the-art implementations by 45%, including that of Agarwal2010 *et al.* [30]. However, Hong *et al.*only compared their implementation on an 8-core CPU system to that on Agarwal *et al.*an 8-core CPU system but not on their 32-core CPU system, which is faster than Hong *et al.*'s implementation. Finally, Hong *et al.*concluded from their work that the governing factor for performance was primarily random memory access bandwidth.

Finally, a more recent work on efficient processing of graph traversal algorithms for CPUs is presented in [32]. Chhungani *et al.*propose a synchronous BFS implementation for a single shared-memory node. The proposed method is based on a lock-free and atomic-free, parallel, load-balanced, BFS traversal that aims to reduce the impact of memory latency. Rearrangement of the frontier of vertices and pre-fetching are also employed in [32] to hide latency or irregular graph accesses to fully exploit available memory bandwidth on multi-socket CPUs. In addition, Chhugani *et al.*proposed several multi-socket optimisations that keep cross-socket communication low. Their performance results outperformed previously reported multi-socket BFS implementation, including that of Agarwal *et al.* [30] by a factor of 1.5-3 using an identical system. They also achieved near-linear socket scaling (1.98x for uniformly random graphs, and 1.93x for R-MAT graphs). This represents the fastest implementation of BFS for a single shared-memory node.

Massively multi-threaded machines

Graph algorithms, including BFS, showed impressive performance on these machines [35, 37, 31]. Unfortunately, these massively multi-threaded machines have major drawbacks. The cost of such machines is just too high since they are built from custom processors rather commodity processors. In addition, these machines have been designed for irregular problems that are memory bound, and may perform poorly for other compute-bound applications due to the low clock frequency of the processors. Nevertheless, these machines have provided us with many insights into the design of algorithms and hardware architectures for graph traversal algorithms.

Umit *et al.* [31] investigated the relationship between architectures and algorithm design in shared-memory systems using a specific graph algorithm: distance-1 graph colouring. Two different methods of multi-threaded algorithms are proposed for conventional cache-coherent systems and the non-conventional massively multi-threaded machines. The authors in [31] studied the performance of the algorithm using a spectrum of multi-threaded machines. The study provides the following insights into the design of high performance algorithms for irregular problems on multi-threaded shared-memory systems:

- Memory latency may be tolerated through simultaneous multi-threading.
- The impact of lower clock frequency and smaller cache memories can be reduced by having a large number of active and concurrent threads.
- The performance of parallel graph algorithms is greatly influenced by the structure of the input graph.

2.3.2 Distributed memory systems

Another line of research has tackled the issue of very large graphs that cannot be handled by single machines using either distributed memory systems [38, 39, 40] or external disk memory [41, 42]. Distributed graph processing is considered to be challenging in general, due to the natural irregularity of the underlying graph [6]. Nevertheless, there is an important case in which distributed processing is mandatory: the graph does not fit in a single machine's memory. There are few frameworks or libraries which aim to simplify graph processing in distributed environment. PBGL [40] is a message-passing implementation of the classic boost graph library [43]. Pregel [38] is a distributed framework that encapsulates message passing and fault-tolerance in a similar manner to the MapReduce framework; traditional graph algorithms should be expressed in a description suitable for MapReduce, however.

2.3.3 GPU-based systems

Some researchers [7, 8, 9] used GPUs to accelerate graph algorithms, because modern GPUs share many architectural properties of massively multi-threaded systems such as the Cray XMT [23] that was discussed above.

Harish *et al.* [7] provided an implementation for some fundamental graph traversal algorithms such as BFS, Single-Source Shortest-Path (SSSP), and All-Pairs Shortest-Paths (APSP). The results reported in [7] showed that graph algorithms can be executed on GPUs with performance results that outperformed multi-core CPUs. The size of input graph was rather small (only up to 60 million edges) due to the size of the GPU's global memory.

A hybrid CPU-GPU approach is proposed to accelerate the BFS algorithm by Hong *et al.* [8]. In this hybrid method, the GPU only execute BFS iterations, in which there is enough parallelism to saturate the massive parallel hardware of the GPU. However, executing all BFS iterations on the GPU in [8] was only marginally slower than the hybrid CPU-GPU. The authors suggest that GPUs may be a better choice when it comes to absolute performance numbers as long as the GPU memory can fit all the graph data.

A more recent work on GPU-based acceleration of graph traversal algorithms is presented by Merril*et al.* [9]. Unlike previous works [7, 8] that focused on graphs with small diameters, Merril *et al.*proposed a BFS parallelisation that achieves an asymptotically optimal $\mathcal{O}(|V| + |E|)$ work complexity. In addition, the proposed parallelisation strategy is the first implementation that performs a parallel expansion of a single adjacency-list; i.e. multiple parallel threads are used to expand an adjacency-list for a given vertex, in contrast with previous work where only a single thread expands the adjacency-list of a given vertex. The authors made heavy usage of on-chip cache memories (local and shared memories) to avoid redundant work that may occur in the execution of a BFS. While this may greatly improve the performance, the impact of such on-chip memory optimisations weakens as the graph size increases. Moreover, a multi-GPU implementation is presented in [9], where up to 4 GPUs are utilised to execute the BFS algorithm. While using 4 GPUs allows for the processing of graphs which are about 4 times larger, the scalability of this multi-GPU implementation was rather poor. A 4-GPU implementation was only about two times faster than a single GPU implementation, but the four GPUs would still consume four times more power than a single GPU. Finally, the authors in [9] reported performance results that are faster than other CPU and GPU implementations in the literature, but could only compares their performance results up to a limited input graph size due to the size limitation imposed by the GPU's global memory.

Other related work on using GPUs to accelerate graph problems are those reported in [10] and [3]. Both works tackle the acceleration of the APSP problem using GPUs. However, in both [10] and [3], the Floyd-Warshall algorithm is used to solve the All-Pairs Shortest-Paths problem. Unlike most graph traversal algorithms, this algorithm has a rather regular structure of computations and memory accesses that are data independent, and is very similar to dense matrix multiplication.

2.3.4 FPGA-based systems

Much previous work on using FPGAs to solve graph problems has used low-latency on-chip memory resources to store graph data [44, 33, 34, 11]. An early approach to implementing graphs on FPGAs is part of the RAW project [44]. In contrast to our solution, RAW's graphs are stored in the FPGA by creating a circuit that directly resembles the graph edges are routes connected to logic representing vertices. As a consequence, changing nodes and edges requires completely re-routing and reconfiguring the entire FPGA circuit, which is extremely costly in time.

A similar approach is proposed by [34], where a circuit represents a graph of a particular size, while edges are state bits that can be changed quickly at runtime. This approach admits graph modifications at run-time and performs them in a single write (clock cycle) to the FPGA. One drawback of this approach is that graphs can only be represented as an adjacency matrix, which as explained in sectionsec:, uses asymptotically more memory as opposed to the adjacency-list representation. Adjacency-matrix representation coupled with the graphs being stored in on-chip memory, leads to very limited size of input graphs. Finally, using the adjacency-matrix representation for some graph algorithms, such as BFS, will be very inefficient.

However, many real world graphs are too large to fit into on-chip RAMs of FPGAs, requiring the use of off-chip memories such as DRAM. Due to significant differences in access times between on-chip memory and off-chip memory, many efficient FPGA-based solutions are not suitable for high-latency off-chip storage. Some recent publications have described successful parallelisation strategies of graph problems on reconfigurable hardware [14] and [11]. Wang *et al.* [14] propose a multi-softcore architecture on FPGAs for the BFS problem. The proposed architecture utilises message passing for parallel processing among soft cores, including both data transmission and synchronisation. The graph data is stored in off-chip DRAM allowing for the processing of large graph sizes. However, Wang *et al.*employ on-chip memory to store certain data structures (e.g. bitmaps to mark visited vertices), which are implemented on BRAM resources. These data structures consume considerable amounts of BRAM resources which leads to graph sizes being limited by the amount of available BRAM

resources. In [14], the maximum size of the input graph size is set to 256K vertices. In addition, there was no actual hardware implementation of this multi-softcore architecture and the performance results in [14] are restricted to simulation results.

2.4 Conclusions resulting from this review

- The performance of graph traversal algorithms has been primarily dominated by random memory access bandwidth. This favours multi-banked memory subsystems that support a large number of parallel memory accesses as opposed to a hierarchical memory subsystem.

- Conventional cache-based CPU systems are ill-equipped to deal with the random memory access bandwidth demands and the rather high data access to computation ratio found in graph traversal algorithms.

- Unlike cache-based CPU systems, massively multi-threaded machines which employ simpler processor architectures and have next to little memory hierarchy, are better placed to deal with memory bandwidth requirements of graph traversal algorithms. However, the relatively high cost of these machines prevented their widespread usage.

- GPUs are architecturally similar to massively multi-threaded machines, but are much cheaper. However they greatly suffer from the global memory size limitation (only several gigabytes). CPU- and FPGA-based systems, on the other hand, can have access to much more off-chip memory (by more than an order of magnitude).

- There have been previous work on accelerating graph problems on FPGAs, but most previous work dealt with relatively small graph problems. Thus, there is little work on reconfigurable computing acceleration of large-scale graph traversal algorithms.

- Breadth-First Search (BFS) is an important graph kernel that is representative of most graph problems as it has random accesses, irregular computational structures, and the issue of load imbalance. In fact, most previous work on graph traversal problems used BFS as a benchmark for evaluation and comparison with other results from the literature.

2.5 Summary

In this chapter, we present background material and related work for this thesis. We provide some background on graph problems including graph theory basics and the challenges of efficiently processing graph traversal problems. We also review related work on efficient processing of graph problems, and provide a summary of the key concepts that will form the basis of our work that is presented throughout the rest of this thesis. The following chapters will take us through our work on using reconfigurable computing to achieve efficient processing of large-scale graph traversal algorithms.

Chapter 3

High Performance Reconfigurable Computing for Efficient Parallel Graph Traversal

3.1 Introduction

Large-scale datasets are often represented as large graphs or networks. These graph data are processed using graph analysis algorithms, such as Breadth-First Search (BFS), to provide information about vertices, paths, and various properties of graphs. As the time and space complexity grows linearly with the number of graph vertices, efficient parallel processing becomes critical when dealing with hundreds of millions of vertices. Unfortunately, traditional software and hardware solutions that are used to parallelise mainstream parallel applications do not necessarily work well for large-scale graph problems. As we mentioned in Section 2.1.3, graph problems have a number of properties that make them poorly matched to computational methods applied in mainstream parallel applications [6].

Previous work has shown that FPGA-based reconfigurable computing machines can achieve order of magnitude speed-ups compared to microprocessors for many important computing applications [15]. However, one limitation of FPGAs that has prevented widespread usage is the requirement for regular or predictable memory access patterns (i.e., sequential streaming of data from memory) due to the heavily pipelined circuits in FPGA implementations. Applications with irregular memory access patterns, such as graph-based algorithms, achieve much lower memory bandwidth due to the increased number of page misses in DRAM memories. Conse-

quently, this low memory bandwidth incurs many pipeline stalls, resulting in little acceleration from the FPGA, and possibly even deceleration.

In this chapter, we present a novel reconfigurable hardware methodology for efficient parallel graph traversal. Our approach is evaluated on a high-performance reconfigurable computing platform, using a case study to compare it with related work. This chapter provides five main contributions:

- A reconfigurable hardware architecture for efficient parallel processing of large-scale graph problems, which decouples computation and communication while keeping multiple memory requests in flight at any given time, and so takes advantage of the high bandwidth of a multi-bank memory subsystem (Section 3.3).

- A detailed case-study using the well-known BFS problem, providing a practical demonstration of our reconfigurable hardware methodology on a commercial FPGA-based HPC system (Section 3.4).

- Design optimisation techniques that include exploiting the small-world property of randomly-shaped real-world graphs, as well as reducing memory bandwidth requirements via efficient graph data encoding (Section 3.5).

- An in-depth performance evaluation that considers different classes of graphs and analyses impact of design optimisations, performance scalability and sensitivity to graph size, and I/O effects (Section 3.7).

- A performance comparison to related work on the implementation of BFS using state-of-the-art CPUs and GPUs, showing that our reconfigurable hardware solution is able to not only outperform high performance multi-core systems, but also to achieve better performance scaling with respect to graph size. We also compare the performance results of our BFS design against those of the Graph500 list (Section 3.7).

The rest of this chapter is organised as follows. The next section provides background material on the BFS problem. The reconfigurable hardware methodology is presented in Section 3.3. Section 3.4 describes an FPGA-based BFS accelerator that uses our reconfigurable hardware methodology, with design optimisations of this accelerator presented in Section 3.5. Section 3.6 details the methodology for our experiments, and we conclude this chapter with the results of these experiments in Section 3.7.

3.2 Background

3.2.1 Breadth-First Search

The BFS problem is to traverse the vertices of a graph $G(V, E)$ in breadth-first search order starting at a source vertex v_s. Each newly-discovered vertex v_i is marked by its distance from v_s, i.e. the minimum number of edges from v_s to v_i. All the vertices with the same distance value belong to the same BFS level, with the source vertex being in BFS level 0, its neighbours in BFS level 1, the neighbours of the neighbours of v_s in BFS level 2, and so on.

The BFS problem is one of the most common algorithms, and is a building block for a wide range of higher-level graph traversal algorithms. For example, BFS can be used on a given graph to identify all of the connected components, to determine the graph diameter, and to perform a bipartiteness test [45]. BFS has been employed in brain network analysis of very sparse brain network data [2].

3.2.2 Sequential BFS algorithm

Algorithm 1 describes the standard sequential BFS algorithm. CQ (queue for current BFS level) is used to hold the set of vertices that must be visited at the current BFS level. At the beginning of a BFS, CQ is initialised with v_s (Line 5). As vertices are dequeued (Line 10), their neighbours are examined (Line 11). Unvisited neighbours are labelled with their distance, or BFS level (Line 13), and are enqueued for later processing in NQ, the queue for next BFS level (Line 14). After reaching all nodes in a BFS level, CQ and NQ are swapped (Line 16).

3.2.3 Parallel BFS algorithm

Most parallel BFS algorithms are *level-synchronous*: each BFS level is processed in parallel while the sequential ordering of levels is preserved. One common approach to parallelising the BFS algorithm is the quadratic parallelisation or read-based parallelisation of the BFS algorithm [8]. This approach, illustrated in Algorithm 2, is common in BFS implementations for high memory bandwidth machines such as GPUs [8, 7]. In algorithm 2, the distance array, $distance[]$, is used: (1) to determine if a vertex belongs to the current BFS level (Line 8), (2) to check if a vertex has been visited (Line 10), and (3) to mark vertices for processing in the next BFS level (Line 11).

Algorithm 1: Simple sequential BFS

Input: $G(V, E)$, source vertex v_s
Output: Array *distance*$[1..n]$ with *distance*$[i]$ holding the minimum distance of v_i from v_s
Data: CQ: queue of vertices to be explored in current level,
NQ: queue of vertices to be explored in next level

```
1  CQ ← ∅
2  foreach v_i ∈ V do
3      distance[i] ← ∞
4  distance[s] ← 0
5  CQ ⟵ {v_s}
6  bfs_level ← 0
7  while (CQ != ∅) do
8      NQ ⟵ ∅
9      for all v_i ∈ CQ do
10         v ← Dequeue(CQ)
11         foreach u_j adjacent to v do
12             if distance[j] == ∞ then
13                 distance[j] ⟵ bfs_level + 1
14                 Enqueue (NQ,u_j)
15     bfs_level ← bfs_level + 1
16     Swap(CQ, NQ)
```

Algorithm 2: Level-synchronous read-based BFS

Input: $G(V, E)$, source vertex v_s
Output: Array *distance*$[1..n]$ with *distance*$[i]$ holding the minimum distance of v_i from v_s

```
1  parallel foreach v_i ∈ V do
2      distance[i] ⟵ ∞
3  distance[s] ⟵ 0
4  bfs_level ⟵ 0
5  repeat
6      done ⟵ true
7      parallel foreach v_i ∈ V do
8          if distance[i] = bfs_level then
9              foreach u_j adjacent to v_i do
10                 if distance[j] = ∞ then
11                     distance[j] ⟵ bfs_level + 1
12                     done ⟵ false
13     bfs_level = bfs_level + 1
14 until done
```

A primary disadvantage of the read-based method is that the distance array is repeatedly accessed at each BFS level, even if only a few vertices belong to that BFS level. In the worst case, the read-based parallel BFS performs $\mathcal{O}(|V|^2 + |E|)$ work, in particular for graphs with large diameters. However, this rarely happens with randomly-shaped real-world graphs which are governed by the small-world property [46]. Due to this property, the diameter of the graph is generally small, and hence, the number of BFS levels is much smaller than $|V|$. In addition, for such graphs, most vertices belong to one of a few critical BFS levels where the number of these vertices approaches $\mathcal{O}(|V|)$. Since the execution time of the BFS algorithm is dominated by these critical levels, reading the whole $\mathcal{O}(|V|)$-sized array will not be wasteful for these critical levels. Moreover, the memory access pattern of the distance array is sequential, and hence, accessing this array can be achieved efficiently on a high memory bandwidth machine such as GPUs [8].

3.3 Efficient Parallel Graph Traversal on Reconfigurable Hardware

As we have discussed in Section 2.1.3, the computational and memory access requirements of large-scale graph problems are significantly different from mainstream parallel applications, requiring new architectural solutions for efficient parallel graph processing. In this section we propose a reconfigurable computing solution for efficient parallel graph traversal algorithms.

Algorithm 3 shows a general template for the graph traversal algorithms targeted by our reconfigurable hardware solution. In terms of algorithm coding, this property translates into a loop that iterates through all the vertices in the graph. Each loop iteration can be performed as a separate kernel by a processing element (PE). The outer-loop (line 2) represents the coarse-grained parallelism required for our reconfigurable hardware solution, while further fine-grained parallelism may be available within the graph kernel itself (line 3).

Algorithm 3: Graph traversal algorithm template

```
1: INPUT: a graph G(V, E)
2: for each vertex v of G in parallel do
3:     graph_kernel(v)
4: end for
5: OUTPUT: statistical data of G(V, E)
```

3.3.1 Parallelisation strategy

Typically, mapping an algorithm onto a custom hardware accelerator requires extracting parallelism from the algorithm to take advantage of the hardware resources. In the case of FPGAs, designers usually rely on heavily

pipelined designs to compensate for the relatively slow operating frequencies on these devices. However, the irregular memory access pattern requirements of large graph problems result in many pipeline stalls, leading to limited or no FPGA performance speed-up. Instead of attempting to increase throughput by pipelining the PEs , we aim to tolerate off-chip memory latency by pipelining accesses to a multi-bank memory subsystem. In particular, a set of architectural design features and techniques, not necessarily new ideas, are put together to achieve efficient parallel processing of large graph problems. These features and techniques are described in detail in the following.

1. **Custom Graph Processing Element (GPE)**. Designing application-specific GPEs will result in more efficient utilisation of hardware resources, compared to a more general-purpose processing element. For optimised performance levels, GPEs are designed using a hardware description language. However, given that operations performed in graph algorithms are simple compute operations (e.g. no floating-point operations) that map to relatively small hardware implementations, high-level synthesis tools should be able to generate efficient implementations of GPEs.

2. **High coarse-grained parallelism**. This is achieved by instantiating a large number of GPEs in hardware, which operate in parallel in a massively multi-threaded machine fashion. Having a large number of GPEs allows us to take advantage of the abundant parallelism that is often available in graph algorithms (see Algorithm 3, line 2).

3. **Multiple concurrent memory requests**. Instead of using cache memories to hide memory latency, we adopt a *latency masking threads* technique [47]. The GPEs are directly connected to a shared off-chip multi-bank memory subsystem via with no memory hierarchy. Each GPE is capable of issuing multiple outstanding memory requests to shared off-chip memory. Given a large number of parallel GPEs, multiple concurrent memory requests are pipelined leading to superior memory access performance.

4. **Trading speed for area**. Since the execution times of graph traversal algorithms are dominated by memory latency, the processing elements will be idle for most of the time. In other words, a GPE will spend most of its time waiting for memory requests to return from main memory. Hence, having a GPE operating at 500MHz or 100MHZ will make little difference since over 90% of the time the GPE is stalled. So by running at slow frequencies, say 75MHz, the design can be optimised for area, leading to higher parallelism (i.e. higher number of GPEs) compared to designs targeting higher clock rates.

5. **Decoupling access and execution units**. This improves the re-usability of the reconfigurable architecture template, as only the GPEs (execution units) will need to change from one graph algorithm to another.

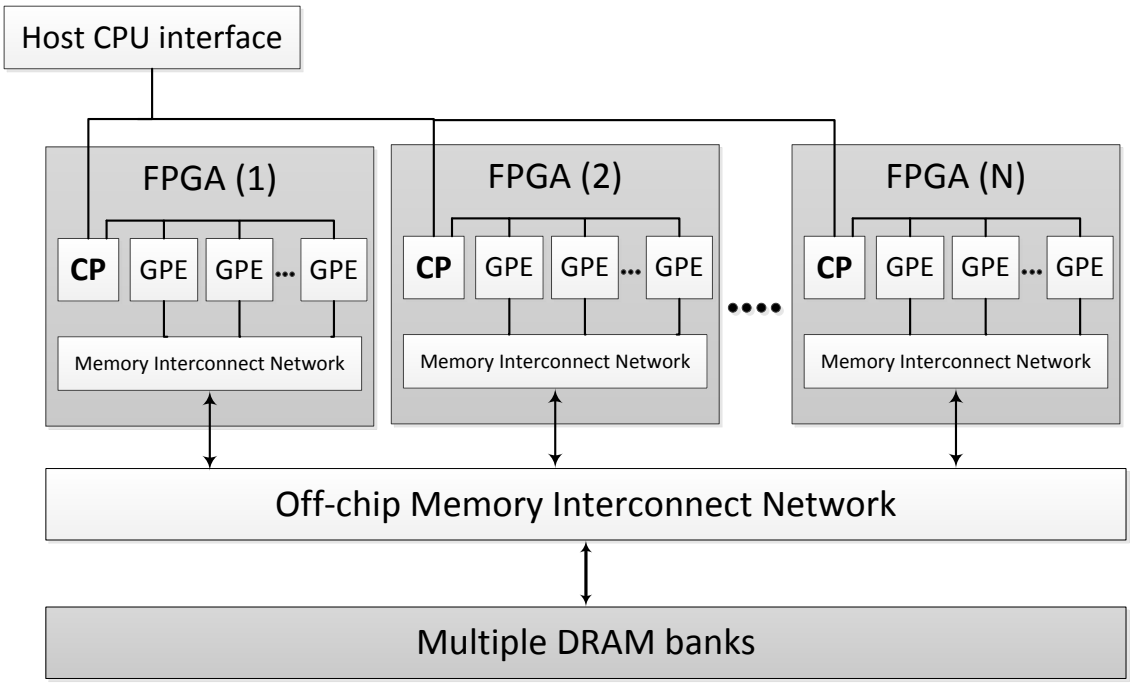


Figure 3.1: Reconfigurable hardware architecture template for graph traversal algorithms.

It will also benefit the hardware synthesis process while improving the productivity of the template user. For example, a GPE (the execution unit) can be generated using a high-level synthesis tool, while the memory interconnect network (the access unit) can be obtained from a library of hand-crafted hardware components.

3.3.2 Reconfigurable hardware architecture template

The overall architecture of the reconfigurable computing solution, as illustrated in Figure 3.1, resembles a scalable, many-core style processor architecture, comprising a ***Control Processor*** (**CP**), multiple ***Graph Processing Elements*** **(GPEs)**, and a ***memory interconnect network***. The GPEs are a collection of replicated and parallel processing elements that are application-specific. Each GPE can independently execute a graph kernel (see Algorithm 3, line 3). The CP acts as a control processor that manages the operation of the GPEs, including initialisation, task assignment and synchronisation of the GPEs. It also provides interfacing to the host CPU processor in the case of high performance reconfigurable systems with an FPGA-based coprocessor architecture. Finally, the memory interconnect network links the GPEs to an off-chip shared memory subsystem via a memory crossbar that provides a point-to-point connection between each GPE and all the off-chip memory banks.

3.4 FPGA-based Parallel BFS

In this section, we describe how we parallelise the BFS algorithm using our reconfigurable hardware architecture template presented in Section 3.3.2. We choose to use a refined version of the read-based BFS algorithm (Algorithm 2) which is proven to be suitable for: (1) platforms with a high memory bandwidth, and (2) randomly-shaped real-world graphs governed by the small-world property [46] as discussed in Section 3.2.3. As for graph representation, we use the popular CSR (Compressed Sparse Row) format which merges the adjacency lists of all vertices into a single $\mathcal{O}(|E|)$-sized array, with the beginning location of each vertex's adjacency list stored in a separate $\mathcal{O}(|V|)$-sized array. The BFS-level of each vertex is stored in a separate $\mathcal{O}(|V|)$-sized array, the *distance* array.

We start by breaking the read-based BFS algorithm into two parts: one part running on the *Control Processor (CP)* (Algorithm 4), and the other part on the GPEs (Algorithm 5).

3.4.1 The CP design

The CP design (Algorithm 4) consists of three main steps:

1. **Initialisation of the GPEs**. The CP initialises the GPEs by partitioning the set of vertices V in disjoint sets V_i (Line 1), one per GPE, such that each GPE owns the vertices in its partition V_i. Each V_i is only explored by its designated GPE_i, but any GPE can mark any vertex in any V_i.

2. **Concurrent computation on GPEs**. After the initialisation step, asynchronous execution of the BFS kernel (Algorithm 5) takes place on each GPE for a given BFS level (line 7). Once a GPE_i has explored all the vertices in its V_i set, it sends a termination signal to the CP, indicating whether there have been any vertices marked for the next BFS level through the *done* signal.

3. **Synchronisation of the GPEs**. Each GPE waits for all GPEs to finish their assigned vertices (line 8). The termination of the BFS algorithm depends on the consensus between all GPEs, and is reached when there are no marked vertices for next BFS level (line 10).

Algorithm 4: Read-based BFS algorithm running on the CP

```
1  Partition set of vertices V into disjoint sets V_i, with |V_i| = |V|/np
2  bfs_level ⟵ 0
3  distance[s] ⟵ 0
4  repeat
5      done ⟵ true
6      in parallel foreach GPE do
7          Invoke BFS_KERNEL (V_i, bfs_level)
8      Synchronise all GPEs
9      bfs_level = bfs_level + 1
10 until done
```

3.4.2 The GPE design: serial execution, parallel access

Since the execution time of the BFS algorithm is dominated by memory latency, the GPE is likely to be idle for most of the time while waiting for data from memory. So to achieve good performance levels, we must deal with the memory latency bottleneck. Our GPE design approach is based on serialising execution and processing of data within the GPE, and parallelising access to off-chip memory. A serial implementation of the BFS kernel leads to an area-efficient design, and hence more GPEs can be instantiated on the FPGA. In addition, the inner loops in Algorithm 5 (Lines 3, 4, 6, and 9) have a data-dependent iteration count, and so cannot be efficiently parallelised through loop unrolling. Parallelising memory accesses is achieved by designing the GPE in such a way that it can sequentially issue multiple outstanding memory requests to a parallel memory subsystem, and use on-chip RAM resources to store data from memory for subsequent processing.

Figure 3.2 presents a schematic overview of the GPE design for the BFS kernel (Algorithm 5). The GPE consists of four functional units that execute the BFS kernel serially. These functional units have access to local storage in the form of dedicated registers that are implemented using distributed RAM. A detailed description of each functional unit follows:

1. **Read distance of** v_i. This unit reads the distance of a given vertex v_i stored in the $distance$ array. If the distance of v_i is equal to the current BFS level (Line 2), it means that the vertex v_i belongs to this level, and hence its neighbours will be explored in the current iteration (Functional units 2-4). Otherwise, the GPE processes the next v_i. This process is repeated until all vertices belonging to V_i have been processed.

2. **Neighbour gathering**. In this unit, the neighbours of v_i are retrieved from memory and stored in local Neighbour registers (Nid registers in Algorithm 5). For area-efficiency reasons, these registers are implemented using distributed RAM instead of Slice registers, and neighbours are retrieved from memory

Algorithm 5: BFS kernel executed by each GPE

Input: V_i: set of vertices to be explored by the GPE,
Array *distance*$[1..n]$ with *distance*$[i]$= *minimum_distance*(v_s, v_i),
bfs_level: current BFS level,
$R[1..n]$: offsets of adjacency lists,
$C[1..m]$: CSR adjacency lists
Output: Array $distance[1..n]$,
$done$: set to false if any vertex has been marked for next BFS level
Data: $Nid[1..q]$: local 32-bit registers to store Neighbour IDs
$bitmask[1..q]$: 1-bit array to store visitation status of the vertices current loaded in the Nid registers,
q: number of Nid registers, size of $Bitmask$ in bits

```
1  foreach v_i ∈ V_i do
2      if (distance[i] == bfs_levels) then
3          for (offset ← R[v]; offset < R[v+1]; offset += q) do
4              foreach i ∈ 0..q do
5                  Nid[i] ← C[offset + i]
6              foreach i ∈ 0..q do
7                  u ← Nid[i]
8                  bitmask[i] ← (distance[u] == ∞)
9              foreach i ∈ 0..q do
10                 u ← Nid[i]
11                 if bitmask[i] == 1 then
12                     distance[u] ← bfs_level + 1
13                     done ← 0
```

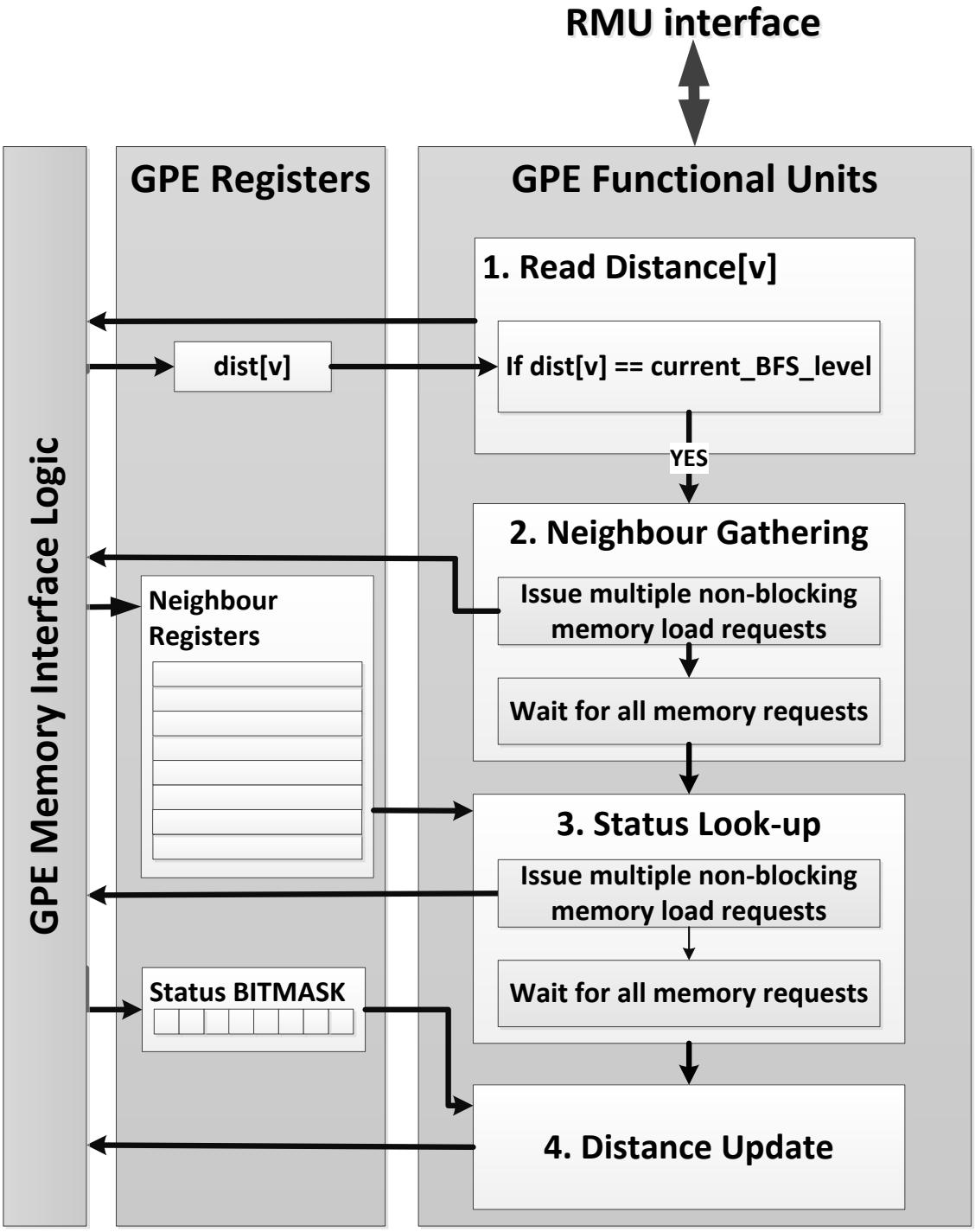


Figure 3.2: *Graph Processing Element (GPE)* design for the BFS kernel (Algorithm 5)

in batches. After each batch is retrieved from memory, steps 3 and 4 are executed (Functional unit 3 and 4) before the next batch of neighbours is read from memory.

3. **Status look-up**. The visitation status of the gathered neighbours is checked in this step (Lines 6-8). Similarly to the previous step, the distances of the neighbours are read using up to q multiple non-blocking memory requests. If the distance has not been set before, then the vertex is marked for the next step using a q-bit *bitmask*.

4. **Distance update**. In this step, the distances of the neighbouring vertices are updated based on the values stored in the *bitmask* in the previous step (Lines 9-12). Vertices marked in previous step will have their distance value updated to the current BFS level plus one (Line 12). By updating a vertex's distance, the vertex is also marked for the next BFS iteration.

3.5 BFS Design Optimisations

In this section, we present three optimisation techniques to improve the performance of our FPGA-based BFS design: Hybrid-BFS, graph data encoding, and multiple non-blocking memory requests. The impact of these optimisations is evaluated in Section 3.7.2.

3.5.1 Hybrid BFS

Beamer *et al.* [48] presented a hybrid approach to the BFS algorithm that combines the conventional top-down approach from Algorithm 1 with a bottom-up approach. This hybrid approach takes advantage of the small-world property of real-world graphs [46] to significantly reduce the number of edges examined, and hence speed up the BFS kernel. Because of the small-world phenomenon the number of vertices in each BFS level grows very rapidly, leading to most edges being examined in one or two BFS levels, the *critical* BFS levels. Algorithm 17 describes the bottom-up algorithm that replaces Algorithm 5 when processing critical BFS levels. Due to the similarities between the top-down and bottom-up BFS algorithms, much of the hardware circuitry used for the top-down BFS algorithm can be used for the bottom-up BFS algorithm.

Algorithm 6: Bottom-up algorithm for BFS kernel

```
1  foreach v ∈ V_i do
2      if (distance[v] == ∞) then
3          for (offset ← R[v]; offset < R[v+1]; offset += q) do
4              foreach i ∈ 1..q do
5                  NID[i] ← C[offset + i];
6              foreach i ∈ 0..q do
7                  u ← Nid[i];
8                  bitmask[i] ← (distance[u] == bfs_level);
9              foreach i ∈ 0..q do
10                 u ← Nid[i];
11                 if bitmask[i] == 1 then
12                     distance[v] ← bfs_level + 1;
13                     done ← 0 ;
14                     break;
```

3.5.2 Graph data encoding

This optimisation is platform-dependent: we consider the size of the native memory word of the target platform in bits, and encode our adjacency list such that for each memory load operation we get more than one neighbouring vertex from the adjacency list. So if the native memory word is k-bit wide, we can obtain $\lfloor k/w \rfloor$ neighbouring vertices per memory operation if we use w bits to represent neighbouring vertices. Figure 5.6 illustrate this idea with k=64 and w=32. An adjacency list must be padded to multiples of 64 bits if the number of neighbouring vertices is not a multiple of 2.

In addition, the switching mechanism from top-down approach to bottom-up approach requires knowledge about the sum of the degree of the vertices marked for the next iteration. So this means that when neighbouring vertices are marked for the next BFS iteration, their degrees need to be obtained, leading to extra random memory accesses. In order to avoid these random memory accesses, the encoding of the $distance$ array is modified as follows: 64 bits are used to store the distance and the degree of each vertex, with bits 0-31 used to store the distance from the source vertex, while bits 32-63 store the degree of the vertex.

3.5.3 Multiple non-blocking memory requests

Instead of issuing one memory request, and then waiting for the response from memory, the GPE issues multiple non-blocking memory requests in an attempt to take advantage of the capabilities of the multi-bank memory subsystem. Assuming that the requests are destined for different memory banks, the high off-chip memory la-

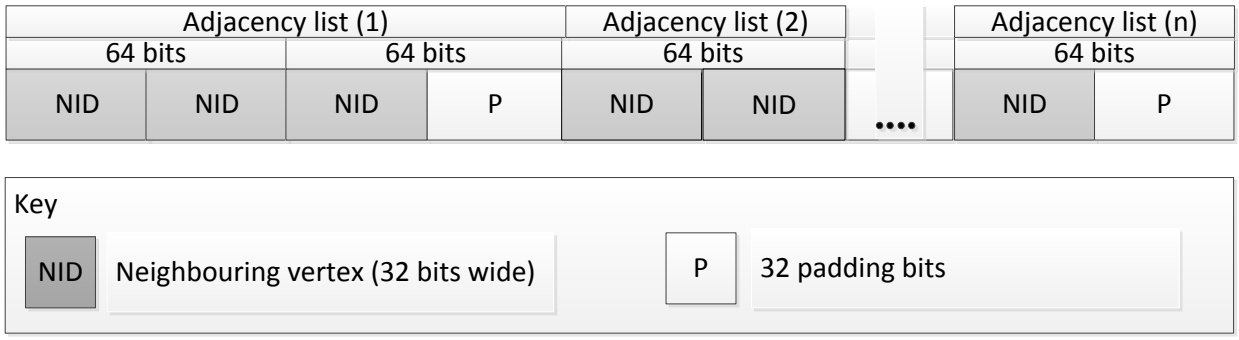


Figure 3.3: The encoding format of the adjacency list.

tency of a single memory request is amortised over multiple memory requests as they get serviced concurrently. The maximum number of non-blocking memory requests is bounded by q, the number of *NID* registers available in the GPE. If the number of neighbours, d, (i.e. degree of v_i) is greater than q, then neighbours are read from memory in batches of size q (Algorithm 5, lines 4-5).

3.6 Implementation and Measurements

This section provides details of our experiments. For the graph data, we use two common classes of graphs: Uniformly Random graphs, and R-MAT graphs. Uniformly Random graphs are generated with n vertices each with degree d, where the d neighbours of a vertex are chosen randomly. The uniformly random graph data are generated using *GTgraph* [49], a synthetic graph generator suite. R-MAT (Scale-free) graphs [50] are characterised by their skewed degree distribution and fractal community. The scale-free graphs are generated using the Graph500 benchmark suite [5] based on the Recursive-Matrix (R-MAT) graph model. For the parameters of the R-MAT graph, we use the default values of the Graph500 benchmark (A=0.57, B=0.19, C=0.19).

Our performance is measured by taking the average execution time of 64 BFS runs from 64 different source vertices which are randomly chosen. To avoid trivial searches, all source vertices must belong to the same connected component whose size is $\mathcal{O}(|V|)$. As in previous related work [8, 30], our BFS performance is reported as the number of traversed edges per second, which is computed by dividing the actual number of traversed edges, $|E|_t$, by the BFS execution time. In our experiments, we notice that $|E|_t$ is about 3% less than $|E|$, the desired number of edges used to generate the graph data by the tools.

To develop for Convey HC-1, we use the Convey Personality Development Kit (PDK) [51], which is a set of makefiles to support simulation and synthesis design flows. Convey provides a wrapper that allows the

Table 3.1: Device utilisation on a Virtex-5 LX330 device

Num. of GPEs	Data encoding	Num. of NID reg.	Slice LUTs	BRAM	f_{max} (MHz)
112	32 bit	16	82%	87%	100
112	64 bit	16	73%	87%	100

Table 3.2: The specification of the machines used in our experiments and related work.

	SC10-EP[30]	SC10-EX[30]	PACT11-NEH[8]	Convey HC-1
Core architecture	Intel Nehalem	Intel Nehalem	Intel Nehalem	Xilinx Virtex-5
Model No.	Xeon X5570	Xeon X7560	Xeon X5550	XC5VLX330
Fabrication process	45 nm	45 nm	45 nm	65 nm
Launch year	2009	2010	2009	2006
Core frequency	2.93 GHz	2.26 GHz	2.66 GHz	100 MHz
Total Num. of CPU Cores	8	32	8	-
Total Num. of FPGA devices	-	-	-	4
Total Num. of threads or GPEs	16	64	16	448
Main memory	48 GB	256 GB	24 GB	16 GB
Maximum memory bandwidth	100 GB/	266 GB/s	100 GB/s	80 GB/s

user to interface the FPGA design with both the host CPU and the memory controllers. Our BFS hardware design is expressed in RTL using Verilog HDL, and was compiled using Xilinx ISE v13.1. Hardware resource utilisation is provided in Table 3.1. Our FPGA-based BFS design has 112 GPEs clocked at 100 MHz. Note that using 32-bit data encoding of the adjacency list requires extra logic circuitry to access the *NID* registers (see Section 3.4).

Table 3.2 provides the configuration details of the machines used in our experiments and in the previous work [8] and [30]. For the high performance reconfigurable computing system, we use the Convey HC-1 server [52] which has four user-programmable Virtex-5 FPGAs with a total of 448 GPEs, and each FPGA is connected to a shared memory subsystem via a memory crossbar. Each memory controller is implemented on its own FPGA and is connected to two Convey-designed scatter-gather DIMM modules. Each AE has a 2.5 GB/s link to its corresponding memory controller, giving a theoretical aggregate peak memory bandwidth of 80 GB/s. However, the effective memory bandwidth of the AEs varies according to their memory access pattern.

3.7 Experimental Results

In this section, we validate the effectiveness of our reconfigurable computing methodology presented in Section 3.3.2. To begin we measure the performance of our BFS design on the Convey HC-1 machine (See Table 3.2), using both uniformly random and R-MAT graph instances with different sizes. We then compare our BFS performance results to those of Agarwal *et al.* [30] and Hong *et al.* [8]. We conclude this section by presenting I/O effects on the performance.

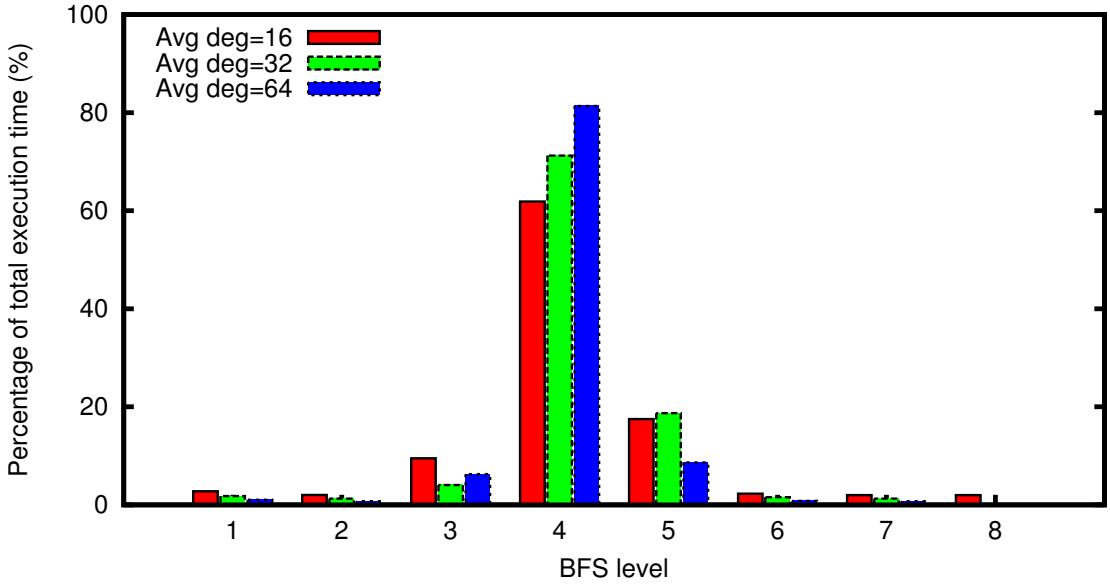


Figure 3.4: Level-wise breakdown of total execution time.

3.7.1 Micro-benchmark analysis

As discussed in Section 3.4.2, the BFS algorithm consists of four steps: (1) Read distance of v_i, (2) neighbour gathering, (3) status-lookup, and (4) distance update. In this section, we use hardware counters to measure the number of clock cycles taken by each step during the execution of a BFS to gain. In our measurements, we measure the cycle count of Step (3) and Step (4) using a combined hardware counter. Figure 3.4 shows the level-wise break down of the execution time of the BFS algorithm on our FPGA implementation, obtained from a BFS run on an R-MAT graph with 1 million vertices and 8-64 million edges. We observe that the execution time of BFS is dominated by two BFS levels (4 and 5), which are known as the critical BFS levels. These critical levels occupy more than 90% of the total execution time of a BFS.

Next, we look onto the execution time breakdown of the critical BFS level 4 for an R-MAT graph with 1 million vertices and varying average degree. Figure 3.5 shows that *neighbour gathering* (Step 2) takes about 40-45% of the total execution time, while Steps (3) and (4) consume about 55-60% of the total execution time of the critical BFS level. These results suggest that any performance improvement will come from optimising Steps (2), (3) and (4), since Step (1) takes only about 1-2% of the total execution time. Step (2), or *neighbouring*

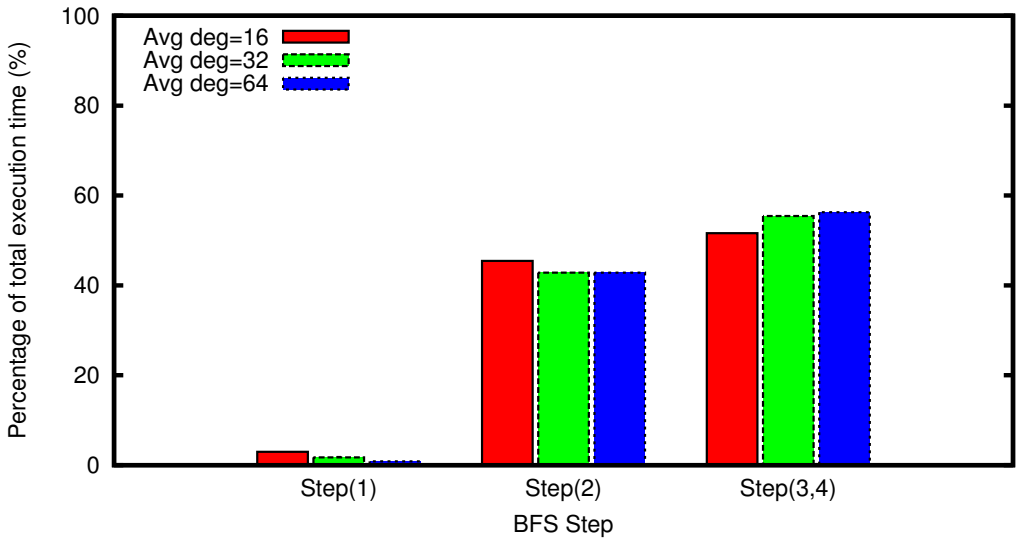


Figure 3.5: Breakdown of execution time of a critical BFS level for the four steps (see Section 3.4.2): Read distance of v_i (Step 1), Neighbour gathering (Step 2), Status lookup (Step 3), and Distance update (Step 4).

gathering, can be optimised through efficient encoding of the adjacency lists that are read from memory in Step (2) (see Section 3.5.2). A *hybrid BFS algorithm* can be used to optimise Steps (2), (3) and (4) as explained in Section 5.5.2. The impact of these optimisations will be discussed in the following section.

3.7.2 Impact of design optimisations

Figures 3.6 and 3.7 show the effects of two design optimisations over the baseline design for uniformly random and R-MAT graphs: 32-bit adjacency list encoding, and hybrid BFS (see Section 3.5). The number of vertices is fixed at 8 million, while the number of edges varies from 64 million to 512 million. Using 32 bits to encode the adjacency lists improves the performance by about 20%. This can be explained by the fact that the number of memory read operations to gather neighbouring vertices is almost halved in comparison with the baseline design that uses 64 bits to encode the adjacency list. For the hybrid BFS optimisation, significant improvements are observed as the average degree of the graph increases, in particular when the average graph degree is 64, where 8 times and 4 times speed-ups are achieved over the baseline design for uniformly random graphs and R-MAT graphs respectively.

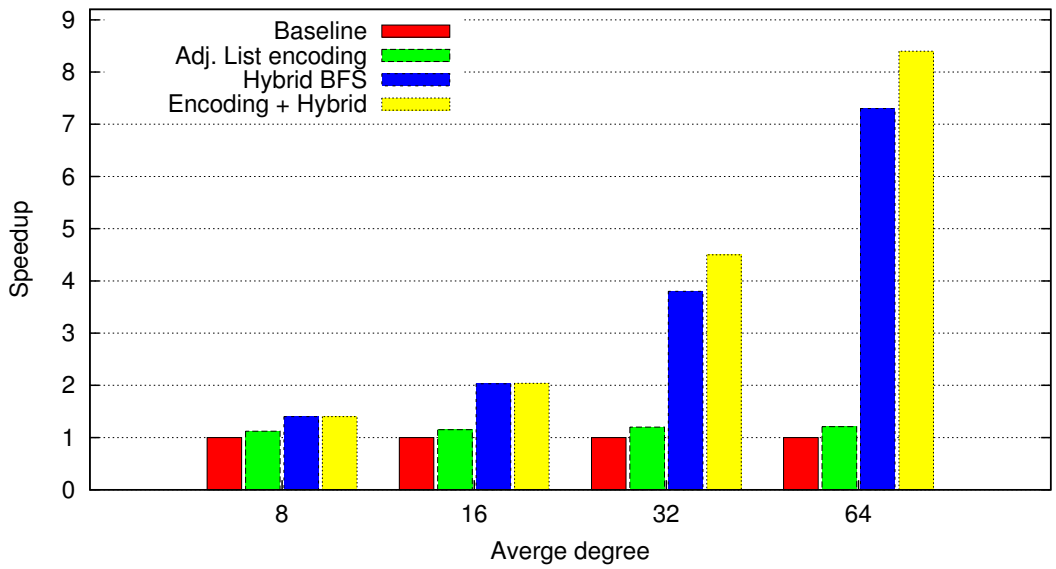


Figure 3.6: Optimisation effects on FPGA-accelerated BFS performance for uniformly random graphs.

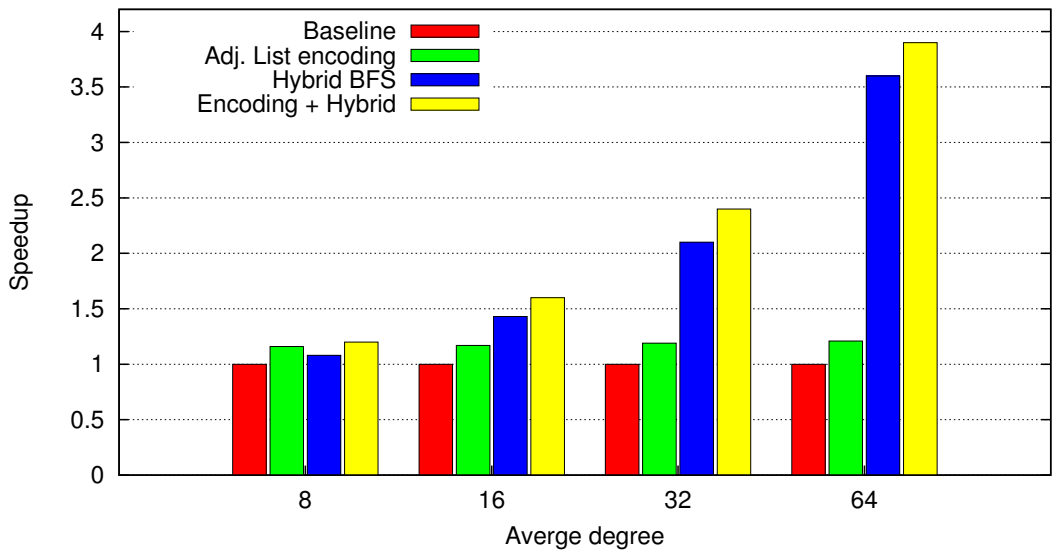


Figure 3.7: Optimisation effects on FPGA-accelerated BFS performance for R-MAT graphs.

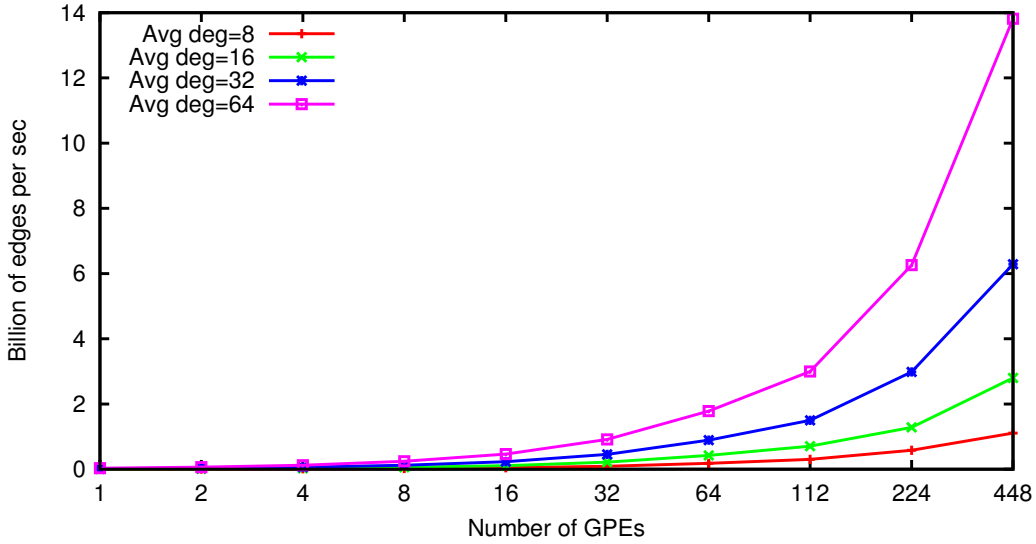


Figure 3.8: FPGA-accelerated BFS performance: processing rate for uniform graphs.

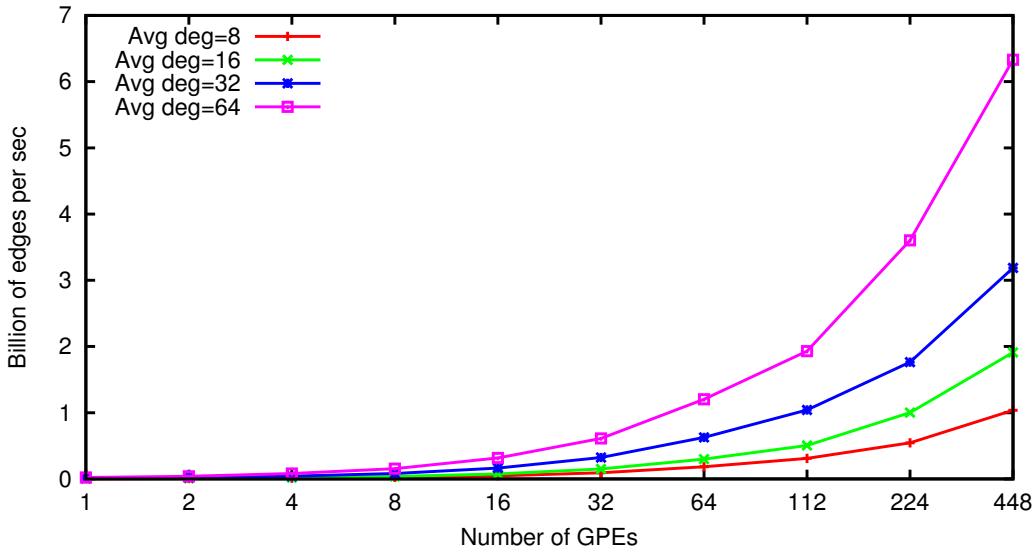


Figure 3.9: FPGA-accelerated BFS performance: processing rate for R-MAT graphs.

3.7.3 Performance scalability

Figures 3.8, 3.9, 3.10, and 3.11 show the processing rate and scalability of our BFS design for both uniformly random graphs and R-MAT graphs as the number of GPEs varies from 1 to 448. The number of graph vertices

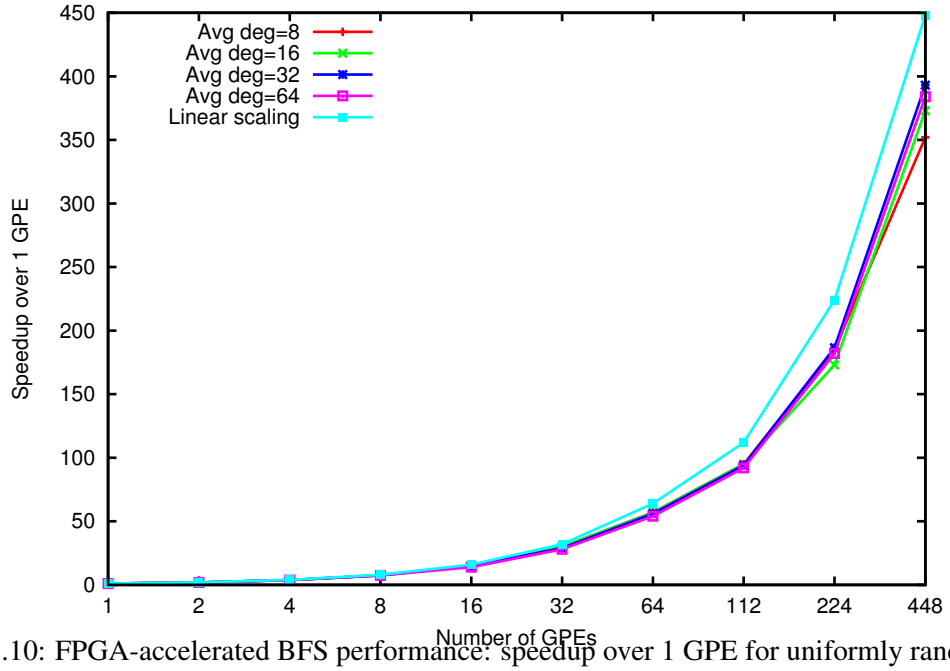


Figure 3.10: FPGA-accelerated BFS performance: speedup over 1 GPE for uniformly random graphs.

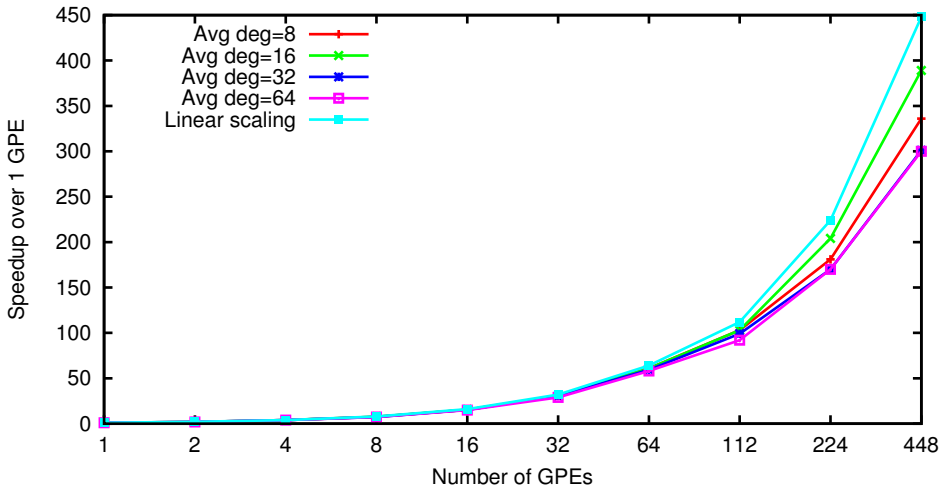


Figure 3.11: FPGA-accelerated BFS performance: speedup over 1 GPE for R-MAT graphs.

is set to 16 million vertices with an average vertex degree of 8, 16, 32, and 64. We define the efficiency as the ratio of speedup of g GPEs over 1 GPE, divided by the linear or ideal speedup, g. In our current design we are able to fit up to 112 GPEs per Virtex5 LX330 device, so we use 2 and 4 FPGA devices for 224 GPEs and 448 GPEs respectively. For large uniformly random graphs, we observe that our design not only scales well on one FPGA device giving an efficiency of about 85%, but also on multiple FPGA devices as we are able to reach efficiency rates of about 94% and 92% for 2 and 4 FPGA devices respectively. For large R-MAT graphs, we see a similar efficiency pattern as to that of uniformly random graphs, albeit with slightly lower efficiency rates: 80% for a single FPGA device, and 92% and 81% for 2 and 4 FPGA devices respectively.

3.7.4 Performance sensitivity to graph size scaling

Figures 3.12 and 3.13 show the average processing rate obtained on four Virtex-5 LX330 FPGAs for uniformly random graphs and R-MAT graphs respectively as the graph size varies. The number of vertices is varied from 1 million to 16 million, while the average vertex degree varies from 8 to 64. We see that the performance increases as the graph size increases in terms of vertex count and/or average vertex degree for both uniformly random and R-MAT graphs. This is due to the fact that our architecture does not make use of low-latency, but small capacity, on-chip memories to hide memory latency. In contrast, the BFS performance of cache-based systems, as in [30], tends to decrease as the graph size is scaled up due to the increased rate of last-level cache misses.

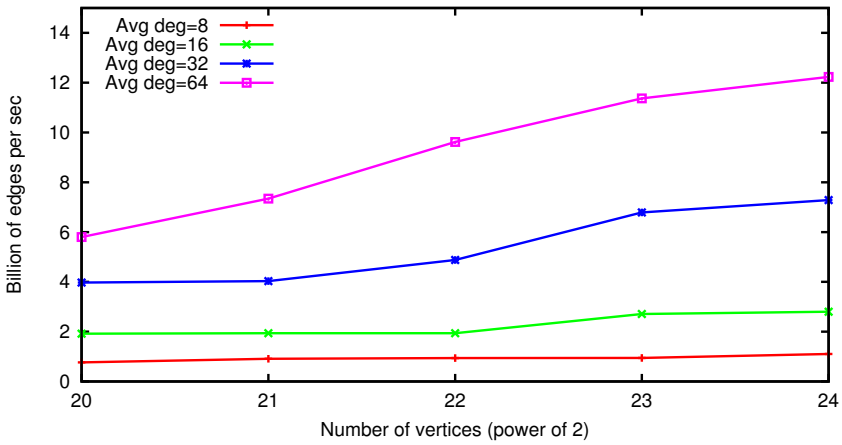


Figure 3.12: FPGA-accelerated BFS: performance sensitivity to graph size scaling for uniformly random graphs: performance scaling with respect to graph size in terms of both vertex count and average vertex degree (Avg deg).

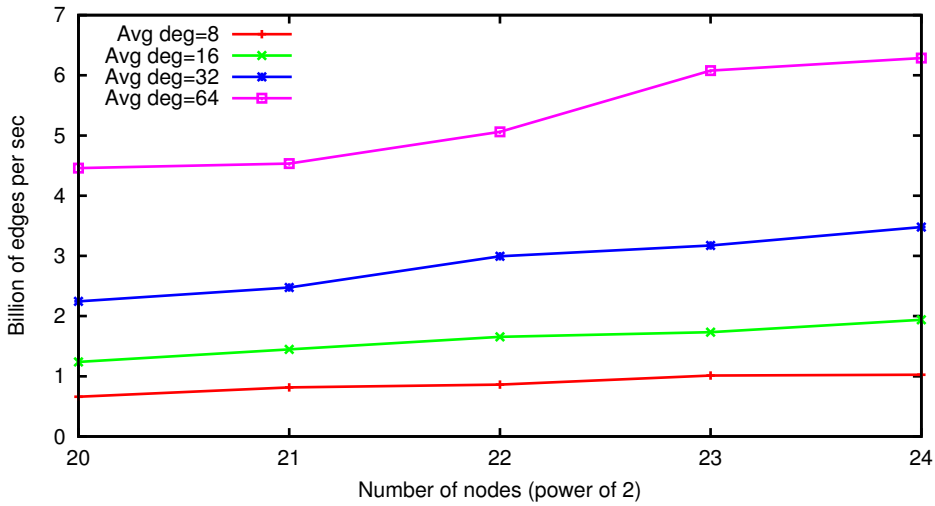


Figure 3.13: FPGA-accelerated BFS: performance sensitivity to graph size scaling for R-MAT graphs.

3.7.5 Performance comparison

Figures 3.14 & 3.15 compare our BFS performance to the performance figures reported in the previous work of Agarwal *et al.* [30] and Hong *et al.* [8], which are reported to be the fastest BFS implementations in comparison with other related work [35], [29], [28], and [53]. The last three sets of bars on the left of each figure represent the measured performance of our optimised hardware implementation of 112 GPEs per FPGA device.

First we compare our performance results to PACT11-NEH (8 Nehalem cores). Using a single Virtex-5 FPGA device based on 65nm technology and operating at 100 MHz, we are able to match the performance of a 2-socket quad-core CPU based on 45nm technology and running at 2.66 GHz for R-MAT graphs. For uniformly random graphs, our FPGA design outperforms PACT11-NEH by a factor of 1.4 times. With Four Virtex-5 FPGAs, we are able to achieve a speedup of 9.6 times and 4.5 times for large instances of uniformly random graphs and R-MAT graphs respectively. For SC10-EX (32 Nehalem cores), our design is able to outperform this high-end 32-core CPU with two Virtex-5 FPGAs by a factor of 3.3 times for uniformly random graphs, and 1.8 times for R-MAT graphs. Using all four Virtex-5 FPGAs, our FPGA-based BFS design performed about 6.3 times and 3.5 times faster than SC10-EX for uniformly random and R-MAT graphs with 16 million vertices and 512 million edges.

From these comparison results, we can say that our FPGA design consistently outperforms the multi-core CPU implementations, as it is able to exploit higher parallelism through 448 custom GPEs compared to 16 and 64

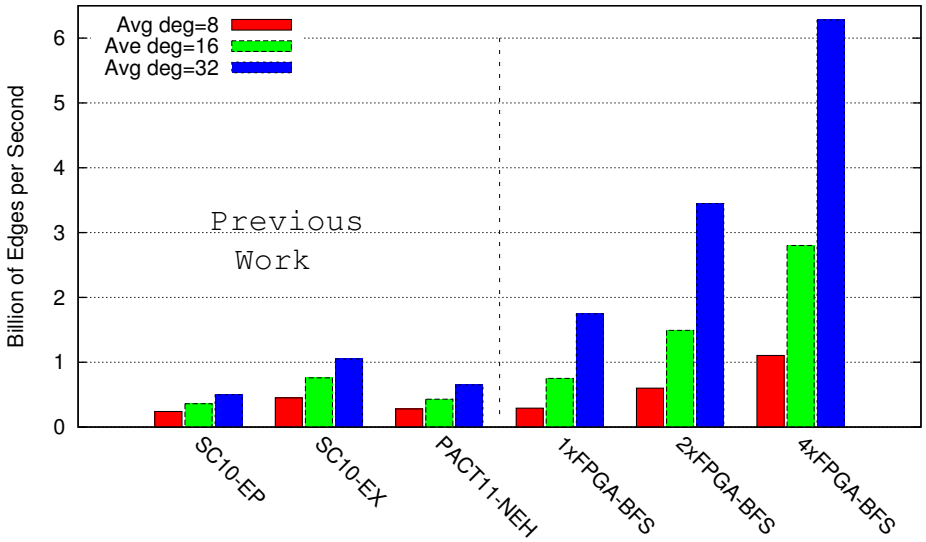


Figure 3.14: Performance comparison of BFS execution on various machines using uniformly random graph instances with 16 million vertices and an average vertex degree (Avg deg) of 8, 16, and 32.

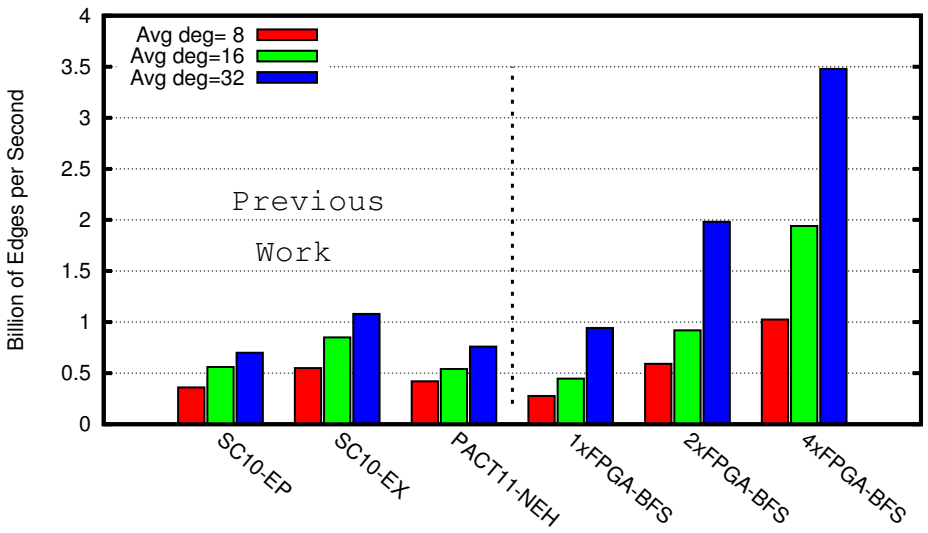


Figure 3.15: Performance comparison of BFS execution on various machines using R-MAT graph instances with 16 million vertices and an average vertex degree (Avg deg) of 8, 16, and 32.

Table 3.3: Extract from Graph500 List for June 2012 [1]..

Rank	Machine	Installation site	Number of nodes	Number of cores	Problem scale	GTEPS
1	Mira (IBM)	DOE/SC/Argonne National Laboratory	32768	524288	38	3541
1	Sequoia (IBM)	LLNL	32768	524288	38	3541
2	(IBM - Power 775 POWER7 8C 3.836 GHz)	DARPA Trial Subset IBM Development Engineering	1024	32768	35	508.05
27	thunder4 (Convey HC-2ex)	Convey Computer Corporation	1	12	27	7.85353
45	**Convey HC-1**	**Imperial College London**	**1**	**6**	**24**	**1.72364**
46	Dingus (Convey-HC-1)	SNL	1	4	28	1.71799
46	Wingus (Convey-HC-1)	SNL	1	4	27	1.71799
47	Vortex (Convey-HC-1)	Convey Computer Corporation	1	4	28	1.61061
50	Cray - XMT	Pacific Northwest National Laboratory	128		29	1.30997

threads for PACT11-NEH and SC10-EX respectively. In addition, as the average vertex degree is increased, the performance gap between HC-1 and the CPUs increases too. This is mainly due to the increased number of random memory accesses issued in the BFS algorithm to read the distance of neighbouring vertices (see Algorithm 2, line 10). By issuing a large number of concurrent memory requests (up to 4x112x16=16384 concurrent requests), our FPGA design can cope better with irregular memory accesses. On the other hand, the CPU-based systems attempt to hide memory latency by using cache memories, which is ineffective for random and irregular memory access patterns.

3.7.6 Graph500 ranking

Since June 2010, Graph500 [5] has been established as a benchmark for data-intensive supercomputer applications. The authors of Graph500 argue that their benchmark is an important step to augment the Top 500 list with data-intensive applications. The Graph500 benchmark consists of two kernels: (1) a kernel to construct the graph from the input tuple list, and (2) a computational kernel, namely BFS, to operate on the graph. Both kernels are timed, and the results of the BFS kernel, expressed in terms of Traversed Edges Per Second (TEPS), are used for the Graph500 ranking of computing machines.

Using our reconfigurable hardware BFS design presented in Section 3.4, we ran the Graph500 benchmark on the Convey HC-1 system and submitted our results to the Graph500 list for June 2012 [1]. Table 3.3 is an extract from the Graph500 list of June 2012. Graph500 uses R-MAT graphs as input graphs with an average vertex degree of 32, while the number of the vertices is given as $2^{ProblemScale}$. Our FPGA-based designed was ranked 45th, and was the fastest single-node FPGA-based system with Virtex-5 FPGAs. The Convey HC-2ex has Virtex-6 FPGAs and DDR3 memory modules, and is considered to be a significant upgrade over both the Convey HC-1 and Convey HC-1ex systems.

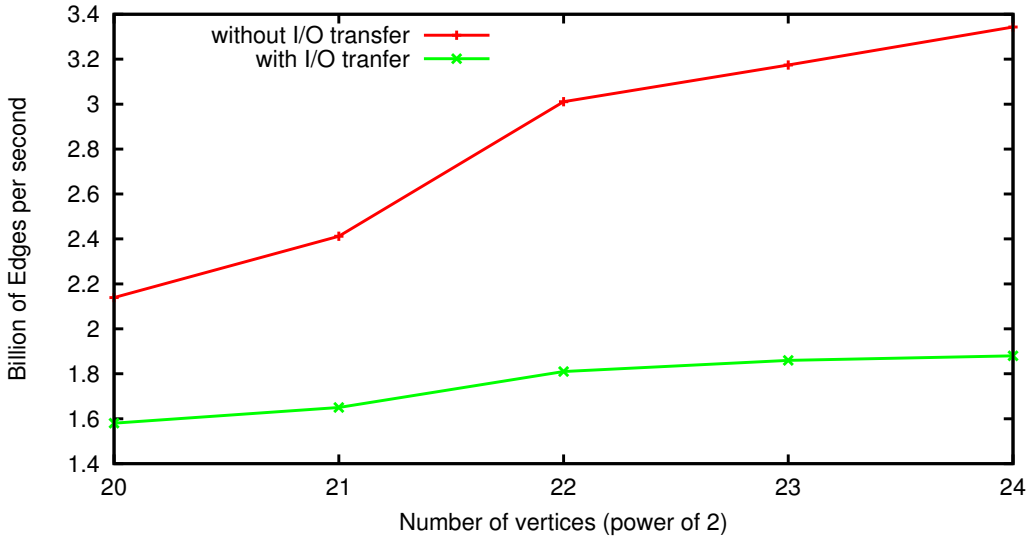


Figure 3.16: Effect of I/O transfer on the performance of the FPGA-accelerated BFS for uniformly random graphs with average degree of 32 and 1, 2, 4, 8, and 16 million vertices.

3.7.7 Effect of I/O transfer on performance

In this section, we examine the impact of data communication between the host processor (CPU) and the coprocessor (FPGA engines). In our work, the Convey HC-1 server is used, which allows the host processor and the coprocessor to communicate over the Front Side Bus (FSB). In theory, the bandwidth of FSB is around 8 GB/s; however, the effective bandwidth of Convey HC-1 is reported to be just below 2.5 GB/s [54]. In order to estimate the effect of I/O transfer of data on the performance, we measure the BFS performance of our design with and without data communication cost. Note that we only consider the transfer of output data, or $distance$ array which, for each source vertex (or a BFS execution), (i) is initialised by host processor, (ii) transferred to the coprocessor memory, (iii) processed by our FPGA-based BFS, and (iv) copied back to the host processor memory. The adjacency list data also needs to be copied to the coprocessor memory once, but can be re-used to execute multiple BFSs on different source vertices that belong to the same graph. So the cost of transferring the adjacency list data is amortised over multiple BFS executions.

Figures 3.16 and 3.17 show the performance of our FPGA-based BFS design using R-MAT graphs with varying number of vertices and edges. From these plots, we observe that I/O transfer of data affects the performance of our FPGA-based BFS by 25%-40%. One way to avoid this I/O transfer penalty is to overlap data com-

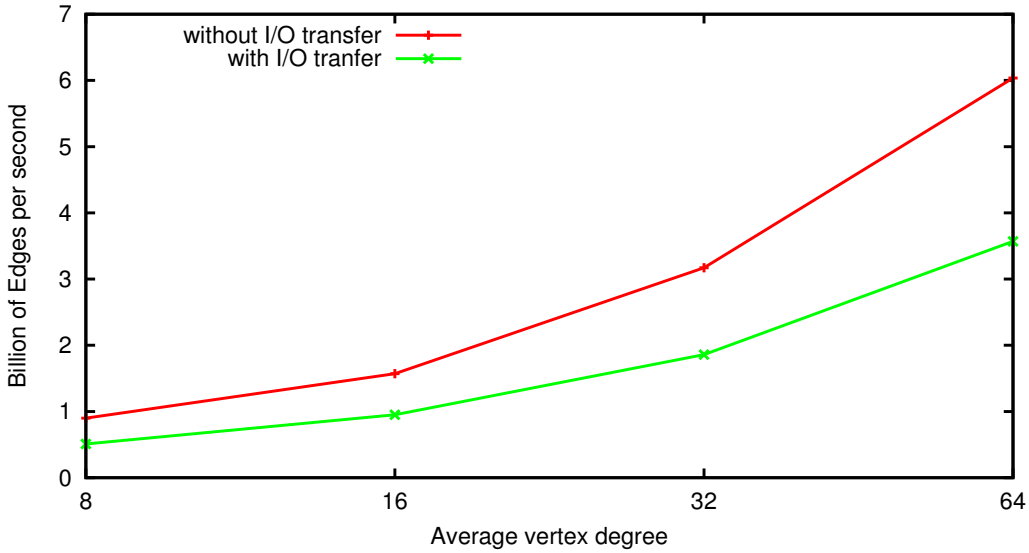


Figure 3.17: Effect of I/O transfer on the performance of the FPGA-accelerated BFS for R-MAT graphs with average degree of 32 and 1, 2, 4, 8, and 16 million vertices.

munication with computation when running multiple BFSs, where performance will be measured in terms of throughput instead of latency, i.e. execution time of a single BFS.

Figure 3.18 compares BFS compute time and BFS I/O transfer time, where for R-MAT graphs with varying number of vertices, the compute time is always greater than the I/O transfer time. Hence, by overlapping the computation of a BFS with the data communication of another BFS, the effect of I/O transfer becomes negligible.

3.8 Summary

In this chapter, we propose a reconfigurable computing methodology for efficient parallel graph traversal of large-scale graph problems. Our approach is based on exploiting the capabilities of both custom hardware computing and the high-bandwidth offered by a multi-bank memory subsystem. To validate our methodology, we provide a detailed description of an FPGA-based accelerator for the BFS algorithm using a high performance reconfigurable computing system. Using power-law graphs found in real-word problems, we are able to achieve performance results that are superior to those of high performance multi-core systems in the recent literature for large graph instances, with a throughput in excess of 6.3 billion traversed edges per second on R-MAT

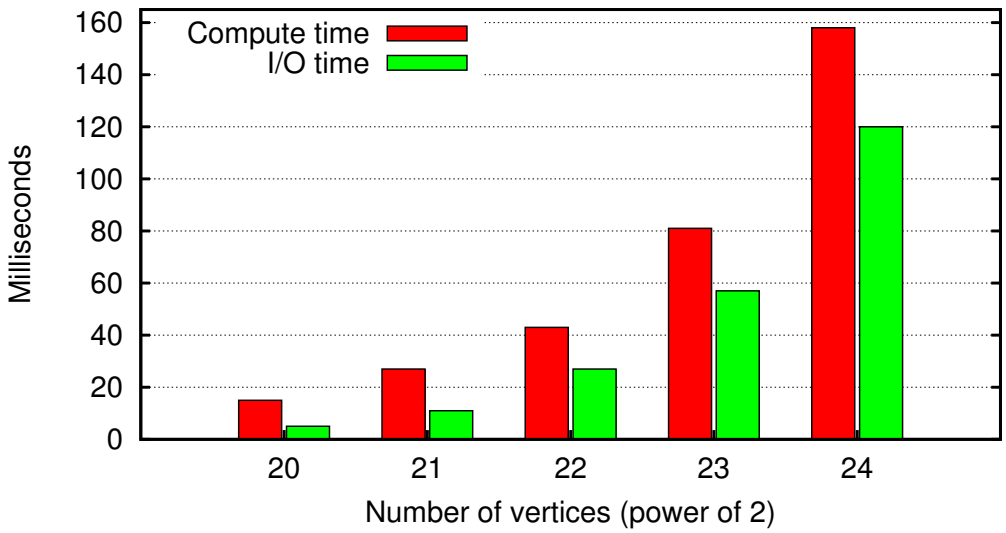


Figure 3.18: Compute time and I/O transfer time for FPGA-accelerated BFS on R-MAT graphs with an average degree of 32 and 1, 2, 4, 8, and 16 million vertices.

graphs with 16 million vertices and over a billion edges. In the following two chapters, we explore other graph traversal problems that can benefit from using on-chip memory resources of FPGAs (Chapter 5), as well as the usage of a different graph data representation than the one used in this chapter (Chapter 4).

Chapter 4

Reconfigurable Hardware Acceleration of Graphlet Counting

In the previous chapter, we presented our approach for reconfigurable computing acceleration of graph traversal algorithms. We also used the well-known BFS problem to demonstrate the effectiveness of our approach. In this chapter, we present our reconfigurable hardware accelerator for the *graphlet counting* algorithm, which is an important graph algorithm for bioinformatics applications. For example, the graphlet counting algorithm is a key kernel in GraphCrunch [55], an open source tool for biological network analysis developed at UC Irvine. In [4], it has been reported that it takes about 2 hours to cluster nodes in a yeast PPI (Protein-Protein Interactions) network (with 9,074 interactions amongst 1,622 proteins) on a multi-core CPU with most of the time being spent on counting graphlets. This a relatively small network if compared with with human PPI networks which are projected to have around 120,000 proteins with 10^6 PPIs [20]. Hence, a solution with a strong computational capability will greatly benefit the PPI research.

Unlike in the case of BFS where only the adjacency-list representation of graphs is used, efficient processing of the graphlet counting algorithm requires both the adjacency-list representation as well as the adjacency-matrix representation of the input graph data. In addition, we consider the graphlet counting algorithm to be more complex than the BFS algorithm. Our main contributions in this chapter are:

- A hybrid graphlet counting algorithm that uses both the adjacency lists and a pseudo-adjacency-matrix to perform adjacency tests. The aim of such an algorithm is to process larger graph problems efficiently as well as improving the performance over the standard graphlet counting algorithm.

- A detailed description of a reconfigurable hardware design for the graphlet counting algorithm providing another practical demonstration of our reconfigurable hardware approach on an actual FPGA-based HPC system.
- An in-depth performance evaluation that analyses the impact of using the proposed hybrid graphlet counting algorithm showing that up to 5 times performance speedup can be obtained over the conventional graphlet counting algorithm.
- A performance comparison of the software and hardware graphlet counting implementations, including all hardware overhead and IO costs, showing that more than two times performance speed-up can be achieved using our reconfigurable hardware graphlet counter.

The rest of this chapter is organised as follows. Section 4.1 provides background material on the graphlet counting algorithm. A performance analysis of the graphlet counting algorithm for a cache-based CPU is presented in Section 4.2. In Section 4.3, we propose a hybrid graphlet counting algorithm that allows us to process larger input graphs compared to the conventional graphlet counting algorithm. Section 4.4 presents our reconfigurable hardware graphlet counter. Section 4.5 details the methodology for our experiments, with the results of these experiments presented in Section 4.6.

4.1 Graphlet Counting Algorithm

The graphlet counting algorithm enumerates all connected graphlets with size $k \in \{3, 4, 5\}$ in an undirected, unweighted graph $G(V, E)$. A graphlet is a small connected induced subgraph a graph $G(V, E)$ [56]. Figure 4.1 shows 3-, 4-, and 5-node connected graphlets. Enumerating all the graphlets in a graph has proved to be extremely useful in many network analysis algorithms such as the GRAph ALigner (GRAAL) [57], and *Graphlet Degree Signatures (GDS)* [58] but is also considered to be a computational bottleneck in network analysis applications.

The pseudo code of the graphlet counting algorithm is shown in Algorithm 7. A detailed C implementation of the graphlet counting algorithm is provided in Appendix A. The basic operations of this algorithm can be described as follows. For each vertex or node a of G (outer loop in Line 1), list all adjacent nodes b of a (Line 2). Then for each node b, list all adjacent nodes c of node b, such that the c nodes are different from the a node (Line 3). At this stage all 3-node graphlets (graphlets $G1$ and $G2$ in Figure 4.1) can be enumerated using an adjacency test of nodes c and a. If node c is connected to node a, then we have a triangle (graphlet

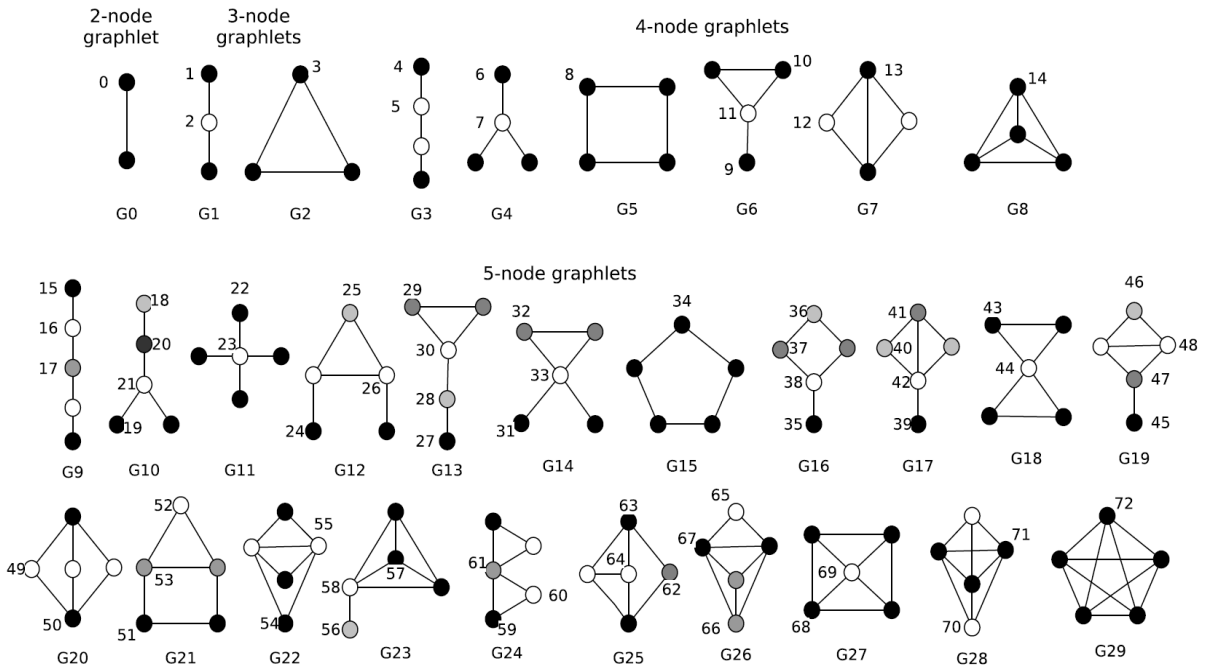


Figure 4.1: 2-, 3-, 4- and 5-node connected graphlets

Algorithm 7: The graphlet counting algorithm for k-node graphlets, with $k \in 3, 4, 5$

```
for all nodes a of G do
    for all adjacent nodes b of a do
        for all adjacent nodes c of b do
            {*finding 3-node graphlets*}
            for all adjacent nodes d of c do
                {*finding 4-node graphlets*}
                for all adjacent nodes e of d do
                    {*finding 5-node graphlets*}
                end for
            end for
        end for
    end for
end for
```

$G2$); otherwise, we have a path (graphlet $G1$). Similarly, using the adjacency tests, we can deduce the different 4-node and 5-node graphlets in the subsequent inner for loops (Lines 5 and 7).

In terms of computational complexity, counting all graphlets in a graph G has a time complexity of $\mathcal{O}(|V|^5)$. However, for very sparse graphs, such as PPI networks, the computational cost is much less prohibitive than in dense graphs. In terms of memory access time, the graph data are often represented in software using pointer-based data structures, which requires un-predictable fine-grained memory access operations to perform node adjacency tests and update graphlet counters. For large-scale graphs, the performance of the graphlet counting kernel is dominated by the wait for memory fetches.

4.2 Performance analysis of graphlet counting

The graphlet counting algorithm is part of the GraphCrunch tool [55] which is an open source tool for large biological network analyses and comparison. GraphCrunch was created by researchers at the School of Information and Computer Science at UC Irvine, and currently implements the latest research on biological network analysis. The graph counting algorithm is at the heart of many of the algorithms used in GraphCrunch. The graphlet counting code is written in C++, and was compiled with gcc version 4.4.6 and run on an Intel Core 2 Duo CPU E8400 running at 3.00GHz with 6MB of L2 cache, and 4 6GB of RAM.

In order to determine the effect of graph size on the performance of the graphlet counting algorithm, we run the following experiments. We generate networks of different sizes $|V|$ and average vertex degree $|E|/|V|$. We measure the execution times as the number of vertices and the average vertex degree vary, and the results of these experiments are reported in Figure 4.2. The execution time increases almost linearly with graph size $|V|$, whereas a non-linear increase of execution time is observed as the average vertex degree is increased. This non-linear increase in execution time can attributed in an increase in last-level cache misses.

In order to investigate the effect of cache misses on the execution time as the graph grows is size and density, we use Cachegrind from the Valgrind instrumentation suite, to measure the effect of the CPU stalls due to L1 and L2 cache misses as the graph size increases. Cachegrind is a cache profiler that performs detailed simulation of L1 and L2 caches in the CPU, and returns the number of cache misses, memory references and instructions executed. The total number of cycles can be approximated by the following equation:

$$Total\ Cycles \simeq Ir + p1 \times L1m + p2 \times L2m \tag{4.1}$$

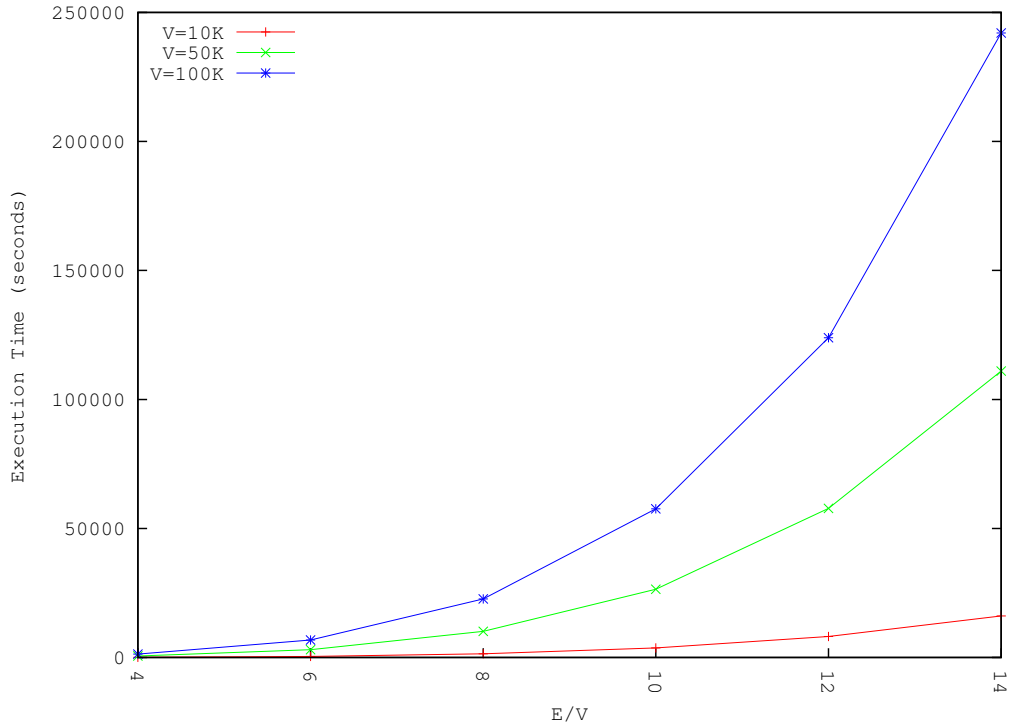


Figure 4.2: Profiling results of the software implementation of the graphlet counting algorithm

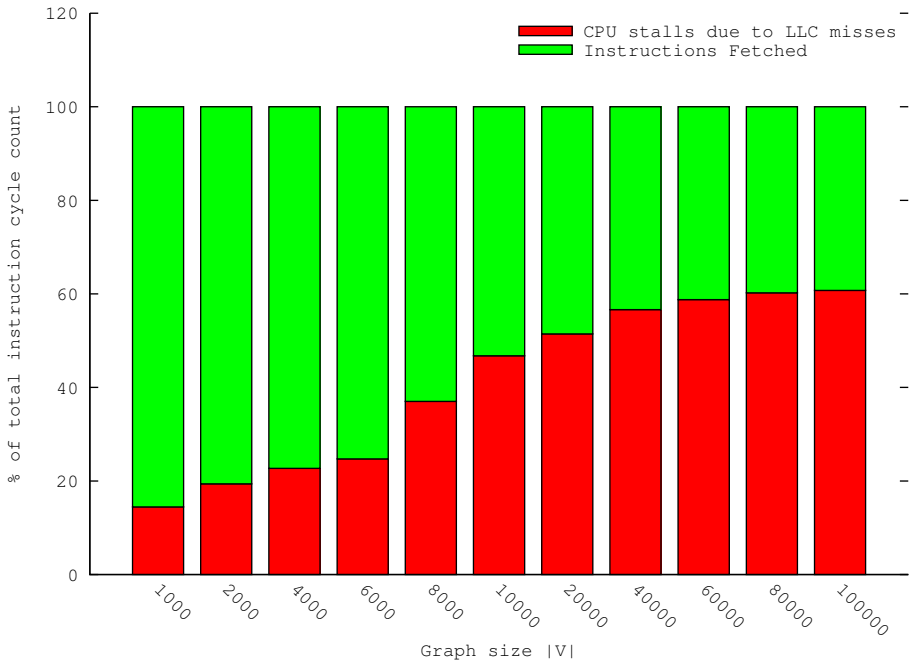


Figure 4.3: Effect of cache misses on the running time of graphlet counting kernel, with an edge density, $|E| / |V| = 10$.

where Ir is the number of instructions retired,

$L1m$ is the number of L1 misses,

$L2m$ is the number of L2 misses,

$p1$ and $p2$ are the miss penalty in CPU cycles for L1 misses and L2 misses respectively.

In an Intel Core 2 Duo processor, an approximate penalty for L1 misses and L2 misses are estimated to be about 12 CPU cycles and 165 CPU cycles respectively [59]. So Eq 4.1 becomes

$$Total\ Cycles \simeq Ir + 12 \times L1m + 165 \times L2m \tag{4.2}$$

and the percentage of CPU stall cycles due to L1 and L2 cache misses can be estimated as follows

$$CPU\ Stall\ Cycles(\%) = \frac{12 \times L1m + 165 \times L2m}{Total\ Cycles} \times 100 \tag{4.3}$$

The results of the Cachegrind profiling tool are shown in Figure 4.3. We fix the average vertex degree and vary the graph size, $|V|$, to determine the effect of cache misses on the running time of the graphlet counting kernel. For large graphs, the CPU is stalled for about two thirds (60%) of the total execution time, compared to about 15% for small graphs. This suggests that if we are to accelerate the graphlet counting algorithm, we need to reduce the number of last-level cache misses. In the following section, we propose a new hybrid graphlet counting algorithm that aims to reduce the number of last-level cache misses, and hence improves the overall performance of the graphlet counting algorithm.

4.3 A Hybrid Graphlet Counting Algorithm

There are several ways to represent graph data using different data structures. Efficiency of a given representation is dependent on the type of operations the algorithm perform on the graph data. In the graphlet counting algorithm, we have two types of operations that requires information from the input graph $G(V, E)$: (i) discovering neighbours of a given vertex, and (ii) performing adjacency tests between any two vertices from $G(V, E)$.

In order to explore the neighbours of a vertex, we represent graphs in the common compact adjacency list form, where the adjacency lists of all vertices are packed into a single array. Each vertex points to the first vertex of its own adjacency list in this large single array of adjacency lists. The vertices are represented as an array

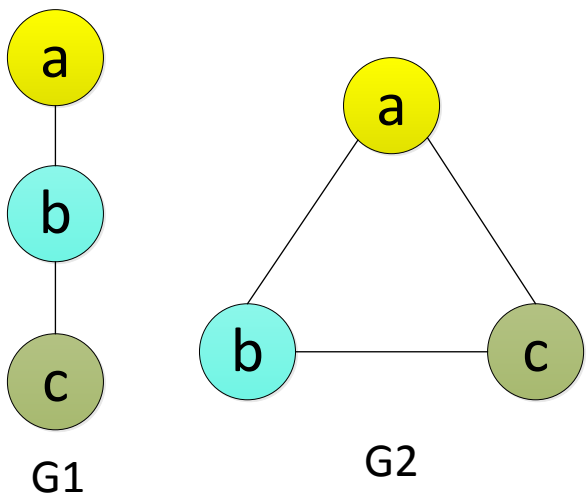


Figure 4.4: 3-node graphlets.

that stores the vertex name and a pointer to its own adjacency lists. Another array of adjacency lists stores the adjacent vertices with neighbouring vertices of vertex i immediately following the neighbouring vertices of vertex $i - 1$.

For adjacency test operations, there is more than one data structure that can be used to perform adjacency tests. In addition to adjacency lists data structure, adjacency matrix can also be used to find out if two vertices are connected. We present each approach in the context of the graphlet counting algorithm, and discuss the advantages and disadvantages of each approach. We then propose a new hybrid approach for performing adjacency lists and demonstrate how it can benefit the graphlet counting algorithm.

To explain this hybrid approach to perform adjacency tests, we use a simplified version of the graphlet counting algorithm (Algorithm 8) in which 3-node graphlets (Figure 4.4) are enumerated. So for each node a of G, its neighbour nodes b are fetched. Then, for each b node, we fetch its neighbours, the c nodes. We know that a is connected to b and that node b is connected to c, since b is a neighbour of a and c is a neighbour of b. This means we have a 3-node graphlet which can be either a path (graphlet $G1$ in Figure 4.4) if node a is not connected to node c, or a triangle (graphlet $G2$ in Figure 4.4) if node a is connected to node c. To determine if a is connected to c, we perform an adjacency test between node a and node c (Line 5), and update the graphlet counters accordingly.

Algorithm 8: The graphlet counting algorithm for 3-node graphlets

Input: $G(V, E)$
Output: Array *gcount*[1..2] with *gcount*[i] holding the number of grahplets G_i (see Figure 4.1)

```
1 gcount[ ] ← 0
2 for all nodes a of G do
3     for all adjacent nodes b of a do
4         for all adjacent nodes c of b do
5             {*perform adjacency test on a and c*}
6             if (a is connected to c) then
7                 Increment G2 counter by 1
8             else
9                 Increment G1 counter by 1
```

4.3.1 Naïve approaches to perform adjacency tests

We have two obvious approaches to perform adjacency tests: (i) using the adjacency matrix, and (ii) using the adjacency list.

(i) The adjacency-matrix approach

In this approach, the adjacency matrix is used to perform adjacency tests. This is illustrated in Algorithm 9 (Line 5). We use an adjacency matrix that requires $|V|^2$ bits of memory storage. Accessing the adjacency-matrix requires only one memory read operation. A great advantage of using the adjacency-matrix is the execution time since accessing the adjacency-matrix requires only one memory read operation. However, the adjacency matrix requires at least $|V|^2$ bits of memory space which limits the size of the input graphs, while wasting a significant amount of memory space for very sparse graphs (performing graphlet counting on dense graphs may be impractical due to its time complexity of $\mathcal{O}(|V|^5)$).

(ii) The adjacency-list approach

An alternative to performing adjacency tests is to use the adjacency list instead of the adjacency matrix. For example, to test whether nodes a and c, we can read the adjacency list (i.e. neighbours) of node a and search for node c. If we find node c in the adjacency list of a, then a is connected to c. Algorithm 10 illustrates this approach, where the adjacency test of nodes a and c is performed by searching for node c in the adjacency list of node a (Line 6).

Algorithm 9: The graphlet counting algorithm for 3-node graphlets

Input: $G(V, E)$
Output: Array *gcount*[1..2] with *gcount*[i] holding the number of graphlet G_i (see Figure 4.1)

```
1  gcount[] ← 0
2  for all nodes a of G do
3      for all adjacent nodes b of a do
4          for all adjacent nodes c of b do
5              if adjmatrix[a][c] then
6                  gcount[2] ← gcount[2] + 1
7              else
8                  gcount[1] ← gcount[1] + 1
```

Algorithm 10: The graphlet counting algorithm for 3-node graphlets: using the adjacency-list to perform adjacency tests

Input: $G(V, E)$: input graph
Output: Array *gcount*[1..2] with *gcount*[i] holding the number of graphlets G_i (see Figure 4.1)

```
1  gcount[ ] ← 0
2  for all nodes a of G do
3      for all adjacent nodes b of a do
4          for all adjacent nodes c of b do
5              a_is_connected_to_c ← 0
6              for all adjacent nodes n of a do
7                  if n = c then
8                      a_is_connected_to_c ← 1
9                      break
10             if a_is_connected_to_c then
11                 gcount[2] ← gcount[2] + 1
12             else
13                 gcount[1] ← gcount[1] + 1
```

A great advantage of such approach is that using the adjacency list results in great savings of memory storage, as opposed to the adjacency-matrix approach. This allows for processing of much larger graphs in comparison with the adjacency-matrix approach. However, a great drawback of the adjacency-list approach is the prohibitive execution time that results from reading the adjacency-list of a node to perform an adjacency test. This is more accentuated when the degree of the nodes to be tested for adjacency have a large degree, with the worst case approaching $\mathcal{O}\,|V|$ memory read operations.

4.3.2 A new hybrid approach to perform adjacency tests

We propose a new approach that uses both the adjacency lists and the adjacency matrix to perform adjacency tests between vertices in the graph. The idea here is to use the adjacency list to perform adjacency tests for any two vertices if at least one of the vertices has a small degree. Otherwise, if both vertices have a large degree, then the adjacency matrix is used to find out if the two vertices are adjacent or not. This approach is shown in Algorithm 11. A buffer (on-chip memory) is used to store the adjacency list of node a if the degree of a (number of neighbouring nodes) is less than or equal to the size of the buffer reserved for storing the adjacency lists (Line 4). The buffered adjacency list of a can be used to perform the adjacency test (Lines 10-13), otherwise the adjacency matrix is used for the adjacency test (Line 14). The hybrid approach presents two main advantages over the other two approaches that we presented above.

(A) Performance improvement

Using a buffered adjacency list to perform adjacency tests reduces the number of memory operations required to read the adjacency matrix. Memory accesses to the adjacency matrix are random and may lead to a significant increase in last level cache misses in particular for large graphs (See section 4.2). By buffering the adjacency lists of small degree nodes, we can perform adjacency tests for almost free compared to accessing the adjacency matrix stored in off-chip memory. Figures 4.5 and 4.6 shows the reduction in the number of memory accesses to adjacency matrix when the hybrid approach is used to perform adjacency tests. For a buffer size of 10, we obtain of a reduction of 15-25% in memory accesses.

(B) Memory storage requirements

Using the adjacency list to perform adjacency tests for small degree nodes, means that we only need to use the adjacency matrix for large degree nodes. This allows us to adopt a more efficient storage scheme for the

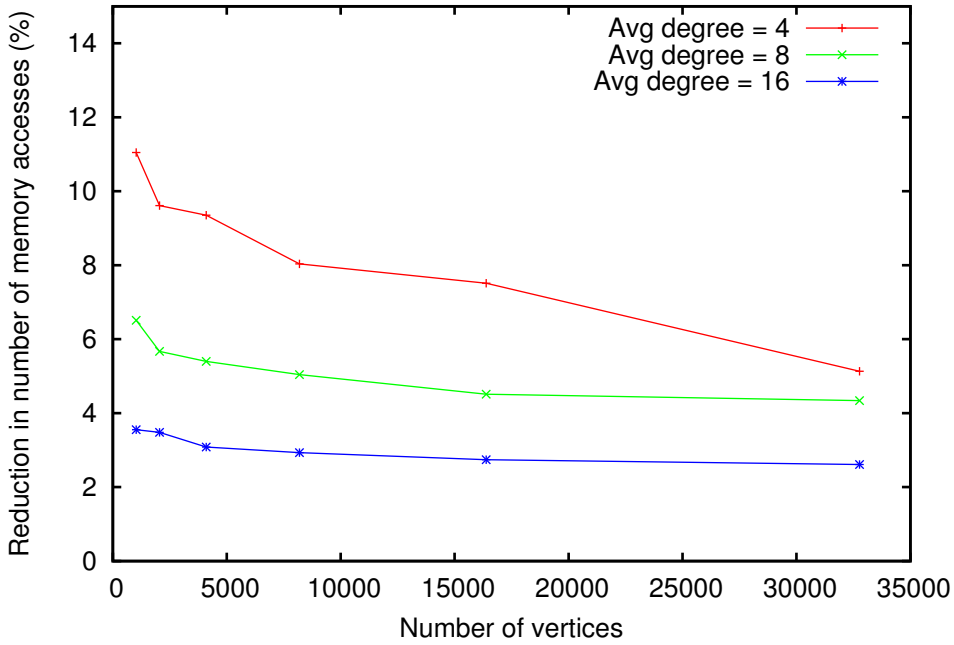


Figure 4.5: Percentage of the reduction in memory accesses to the adjacency matrix using the hybrid adjacency test approach in R-MAT graphs. The size of the adjacency list buffers is set to 5.

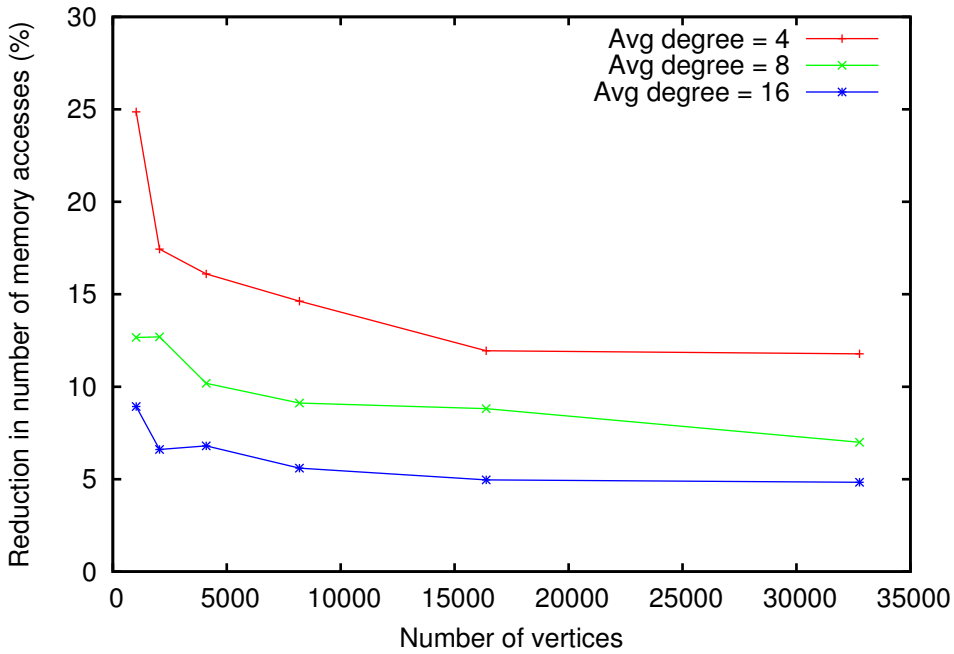


Figure 4.6: Percentage of the reduction in memory accesses to the adjacency matrix using the hybrid adjacency test approach in R-MAT graphs. The size of the adjacency list buffers is set to 10.

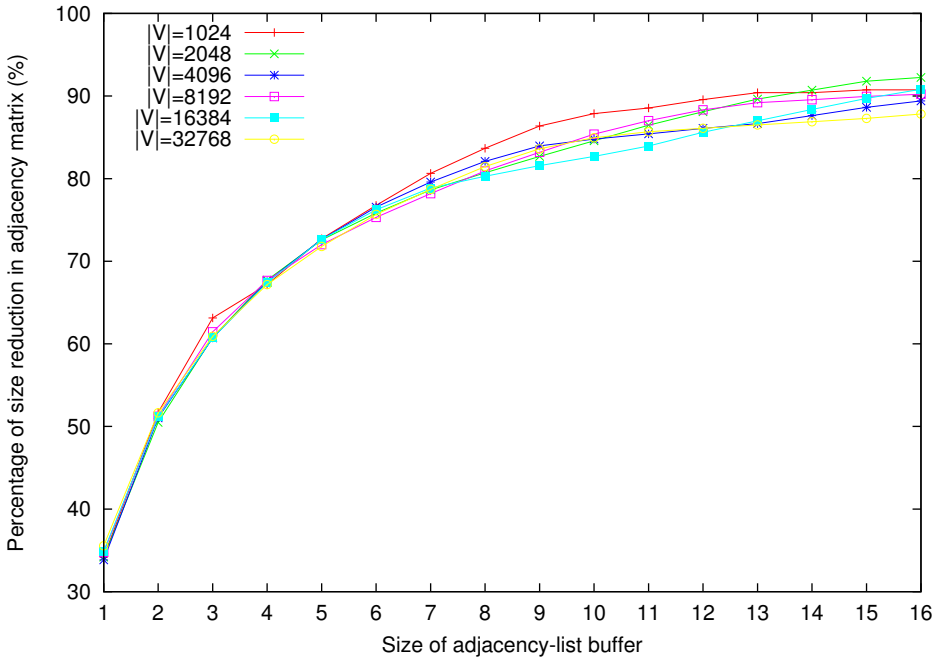


Figure 4.7: Percentage of size reduction in the adjacency matrix for an R-MAT graph with an average degree of 4 (i.e. $|E|/|V| = 4$).

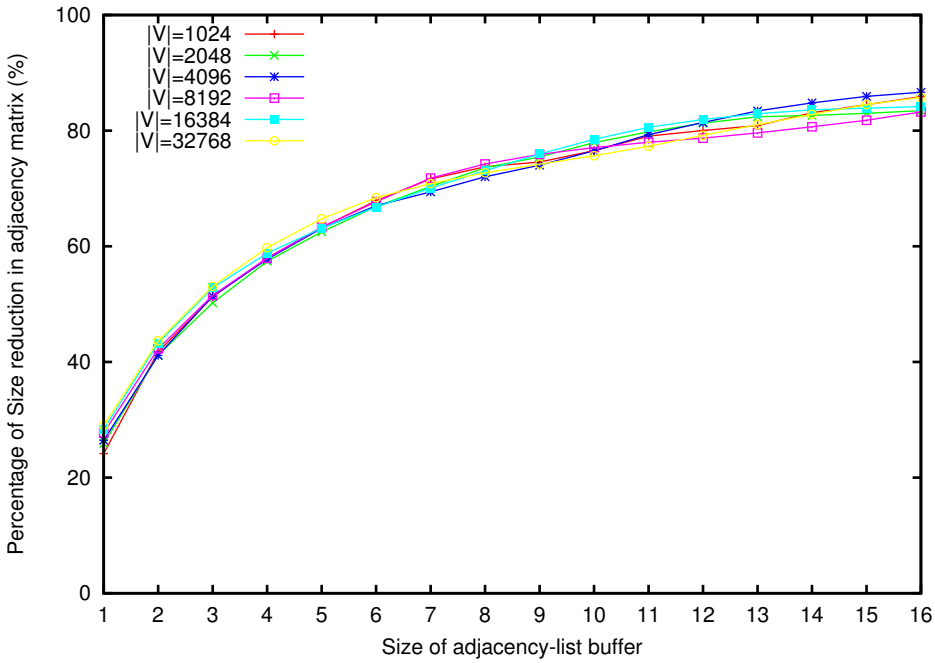


Figure 4.8: Percentage of size reduction in the adjacency matrix for an R-MAT graph with an average degree of 8 (i.e. $|E|/|V| = 8$).

Algorithm 11: The hybrid graphlet counting algorithm for 3-node graphlets

Input: $G(V, E)$: input graph
$adjmatrix[\,][\,]$: the adjacency matrix
Output: Array *gcount*[1..2] with *gcount*[i] holding the number of graphlets G_i (see Figure 4.1)

```
1  gcount[ ] ← 0
2  for all nodes a of G do
3      if degree(a) < BUFFER_SIZE then
4          for all adjacent nodes b of a do
5              A_BUFFER[ ] ← b
6      for all adjacent nodes b of a do
7          for all adjacent nodes c of b do
8              a_is_connected_to_c ← 0
9              if degree(a) < BUFFER_SIZE then
10                 for all nodes n in A_BUFFER do
11                     if n = c then
12                         a_is_connected_to_c ←= 1
13                         break
14             else
15                 a_is_connected_to_c ← adjmatrix[a][c]
16             if a_is_connected_to_c then
17                 gcount[2] ← gcount[2] + 1
18             else
19                 gcount[1] ← gcount[1] + 1
```

adjacency matrix. We create what we call a *pseudo adjacency-matrix* for the high-degree nodes. A node degree is considered to be high if it is larger than the size of the on-chip memory buffer. Rows of the pseudo adjacency matrix only represent the high degree nodes, since most values in a row representing low degree nodes are 0s while very few of these values are 1s. Using a pseudo adjacency matrix allows us to process larger graphs than with a traditional adjacency matrix. Figures 4.7, 4.8, and 4.9 show that using the pseudo adjacency matrix leads to huge savings (up to 90%) of memory storage requirements for very sparse R-MAT graphs.

4.4 FPGA-based Graphlet Counter

In this section, we describe how we parallelise the graphlet counting algorithm using our reconfigurable hardware architecture template (Figure 4.10) presented in Section 3.3.2. For graph representation, we use the popular CSR (Compressed Sparse Row) format which merges the adjacency lists of all vertices into a single $\mathcal{O}(|E|)$-sized array, with the beginning location of each vertex's adjacency list stored in a separate $\mathcal{O}(|V|)$-sized array. The BFS-level of each vertex is stored in a separate $\mathcal{O}(|V|)$-sized array, the *distance* array. We also use

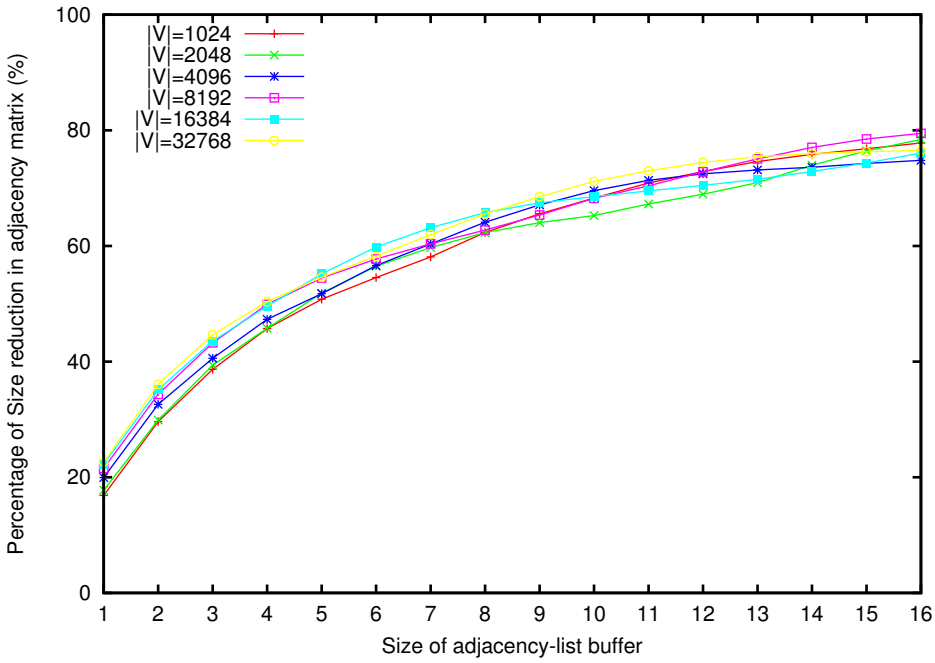


Figure 4.9: Percentage of size reduction in the adjacency matrix for an R-MAT graph with an average degree of 16 (i.e. $|E|/|V| = 16$).

an adjacency matrix that is stored in an $\mathcal{O}(|V|^2)$-sized array.

4.4.1 Reconfigurable hardware parallelisation

In section 2.1.3, the main challenges in parallel graph processing are outlined. In this section, we briefly present instances of these challenges in mapping the graphlet counting algorithm onto hardware:

- *Unstructured problem:*. This is demonstrated by the variable number of iterations of the inner loops, which strongly depends on the degree of the graph nodes: the number of iterations varies between 1 and $|V| - 1$. As a result, it is not obvious where to introduce parallelism in the inner loops.

- *Poor data locality*: The graphlet counting algorithm explores the structure of a graph by performing fine-grained and random memory accesses such as retrieval of neighbouring vertices, or adjacency tests. These memory operations often exhibit poor temporal and spatial memory access locality characteristics.

- *Synchronisation issues*: In the graphlet counting algorithm, we require 29 counters to enumerate all the 3-, 4-, 5-node graphlets (see Figure 4.1) for a given graph. These counters needs to be accessed by all

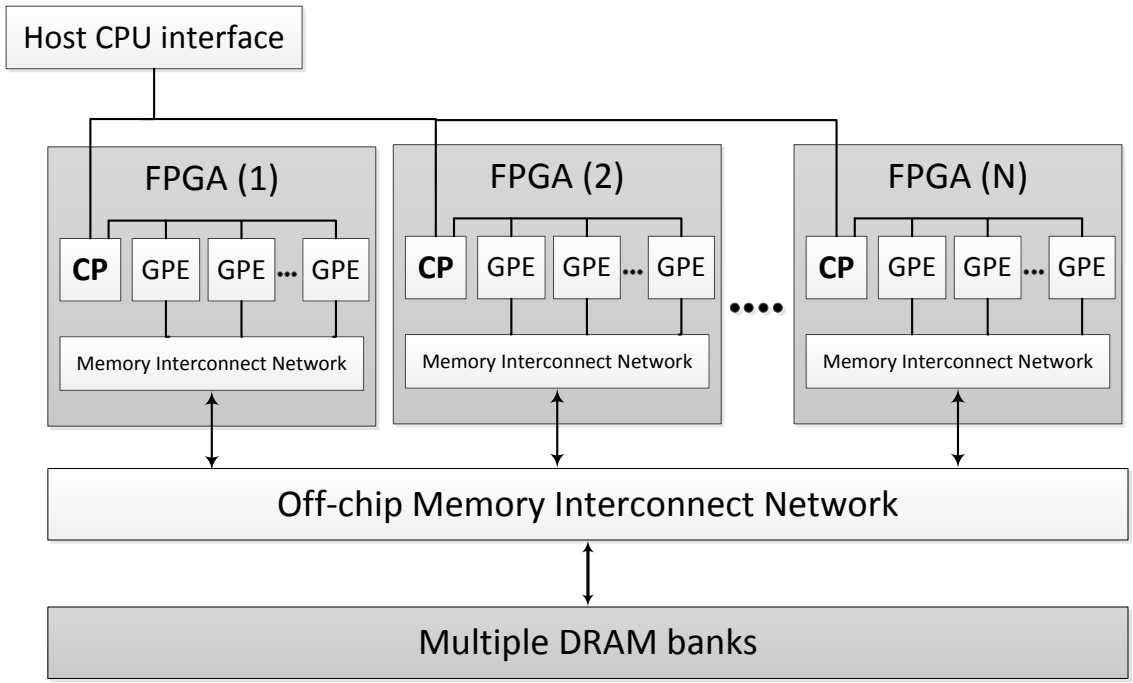


Figure 4.10: Reconfigurable hardware architecture template for graph traversal algorithms

GPEs. A counter may be incremented by two or more processing elements simultaneously requiring a synchronisation mechanism to avoid data hazards due to race conditions. In the case of a system with a large number of parallel threads or processing elements, synchronisation due to high-contention situations can become a performance bottleneck because of the additional delays introduced by contention.

Having mentioned the main design issues of the graphlet counting algorithm, we now present how we can address these issues.

- *Parallelising graphlet counting kernel.* This issue is addressed by a combination of a parallel implementation of the outer loop (line 1 in Algorithm 7), and a serial implementation of all the inner loops (lines 2, 3, 5, and 7 in Algorithm 7). The serial implementation takes the form of a GPE, while the parallel implementation comes from the replication of this GPE. In other words, several GPEs operate in parallel, and each GPE processes a graph node at a time; i.e. an iteration of the outer-loop. Implementing the inner loops sequentially on hardware would most likely result in slower execution times compared to software implementations. These slow execution times can be overcome by having the number of processing units large enough so that the overall execution time of all processing elements is smaller than the software execution time.

- *Hiding memory latency*. The issue of poor data access locality of the algorithm can be addressed by directly connecting the GPEs to a parallel memory subsystem, omitting general purpose cache memories. We use a memory interconnect network to route memory requests between the GPEs and multiple memory banks for parallel access. Latency is tolerated by instantiating a large number of GPEs that can issue multiple outstanding memory requests.

- *Synchronising graph counters*. Finally for the synchronisation of the counter updates, each GPE has its own set of counters (29 counters). These counters are stored in on-chip memory resources (registers). Once all nodes have processed, these counters are copied to off-chip memory where they can be summed up together to obtain the final results of the graphlet counting algorithm. This approach will avoid atomic operations required to prevent read-after-write (RAW) hazards that may be caused by two GPEs trying to update the same counter. Having atomic-free operations will not only simplify our design (and hence improve device resource utilisation), but it will also avoid a significant performance penalty that may have been imposed by the usage of atomic operations.

4.4.2 Reconfigurable hardware design

In this section, we describe the implementation details of the three main components in our graphlet counting hardware accelerator: the GPEs, the memory interconnect network, and the control processor. We start by breaking the graphlet counting algorithm into two parts: one part running on the *Control Processor (CP)* (Algorithm 12), and the other part on the GPEs (Algorithm 13).

Algorithm 12: Graphlet counting algorithm running on the Control Processor (CP)

```
1 Initialise GPEs
2 in parallel foreach v_i ∈ V do
3     Invoke GRAPHLET_COUNTING_GPE(v_i)
4 Synchronise all GPEs
```

The Control Processor (CP) design

This unit manages the execution of the GPEs at run-time by using a dynamic scheduler, which dynamically assigns nodes to the GPEs for processing. The CP design (Algorithm 4) consists of three main steps:

1. **Initialisation of the GPEs**. The CP initialises the GPEs by setting all GPEs to an idle state where they will be ready to receive nodes for processing. The CP also provides information about the graph data

such as memory addresses and the size of the input graph. The values of the on-chip counters are cleared (set to 0). Initially all GPEs receive one node for processing.

2. **Concurrent computation on GPEs**. After the initialisation step, asynchronous execution of the graphlet counting kernel (Algorithm 13) takes place on each GPE for a given node (Algorithm 12, line 2). Once a GPE has processed the assigned node, it sends a request to the CP indicating that it is ready to process another node. The CP will assign a new node (that has not been processed yet) to the GPE.

3. **Synchronisation of the GPEs**. Once there are no more nodes to process, any GPE that sends a termination signal to CP will be requested to write the results stored in its counters to memory and wait for the other GPEs to finish their task. Once all the GPEs have completed their assignment a termination signal is sent to the host CPU via the host CPU interface to indicate that the graphlet counting algorithm has finished executing.

GPE design

Since the outerloop in the graphlet counting algorithm (Algorithm 7, line 1) is implemented by the CP, the GPEs implement the inner loops of Algorithm 7 (lines 2, 3, 5, and 7 in Algorithm 7). Figure 4.11 presents a schematic overview of the GPE design for the graphlet counting kernel (Algorithm 13). The GPE consists of three main functional blocks that execute the graphlet counting kernel serially. These functional blocks have access to local storage in the form of dedicated registers for storing local counters as well as buffering the adjacency list of small degree nodes. These local memories are relatively small and are implemented using distributed RAM. Each functional block enumerate graphlets with a specific size k, with $k \in 3, 4, 5$. Each functional block has three functional units that perform the following three tasks: neighbour gathering, adjacency tests, and counter updates. A brief description of each functional unit is provided below.

- **Neighbour gathering**.The neighbours (or adjacent nodes) of a given node are retrieved from off-chip memory. If the size of the adjacency list (node degree) is less than or equal to the size of the adjacency list buffer than the adjacency list is buffered for adjacency test purposes.

- **Adjacency tests**. In this unit, adjacency test are performed between nodes. Figure 4.12 provides a schematic diagram of an adjacency test unit that determines if node a is connected to node c. Depending on the degree of node a, adjacency tests can be performed using either the buffered adjacency list of node a, or by accessing the adjacency matrix from off-chip memory (see Section 4.3.2). Note that for

4-node graphlets and 5-node graphlets, two and three parallel adjacency tests are performed respectively by replicating the functional unit presented in Figure 4.12.

- **counter updates**. Once adjacency tests are performed, the on-chip graphlet counters are updated accordingly.

Algorithm 13: GRAPHLET_COUNTING_GPE(a)

Input: $G(V, E)$, node a
Output: Array *gcount*[1..29] with *gcount*[i] holding the number of grahplets G_i (see Figure 4.1)

```
1  for all adjacent nodes b of a do
2      for all adjacent nodes c of b do
3          Perform adjacency test (a,c)
4          Update 3-node counters
5          for all adjacent nodes d of c do
6              Perform adjacency test (a,d)
7              Perform adjacency test (b,d)
8              Update 4-node counters
9              for all adjacent nodes d of c do
10                 Perform adjacency test (a,e)
11                 Perform adjacency test (b,e)
12                 Perform adjacency test (c,e)
13                 Update 5-node counters
```

Memory interconnect network

The memory interconnect network provides each GPE with access to all off-chip memory banks via a memory crossbar. The memory crossbar consists of two main parts: one for memory access requests, and the other for memory access responses. Each part consists of three different component: FIFOs, an arbiter, and multiplexer. For memory requests, each GPE has its own FIFO to queue up memory requests. The arbiter controls the multiplexer using an N-way round robin scheduler, where N is the number of GPEs. For the memory responses, FIFOs are also used to queue up memory responses before being transferred to GPE through a multiplexer that is controlled by an round-robin arbiter. In order to achieve timing closure for large numbers of GPEs, the memory crossbar is organised into two or three pipelined stages, which results in a reduced critical path in exchange for a negligible increase in memory access latency.

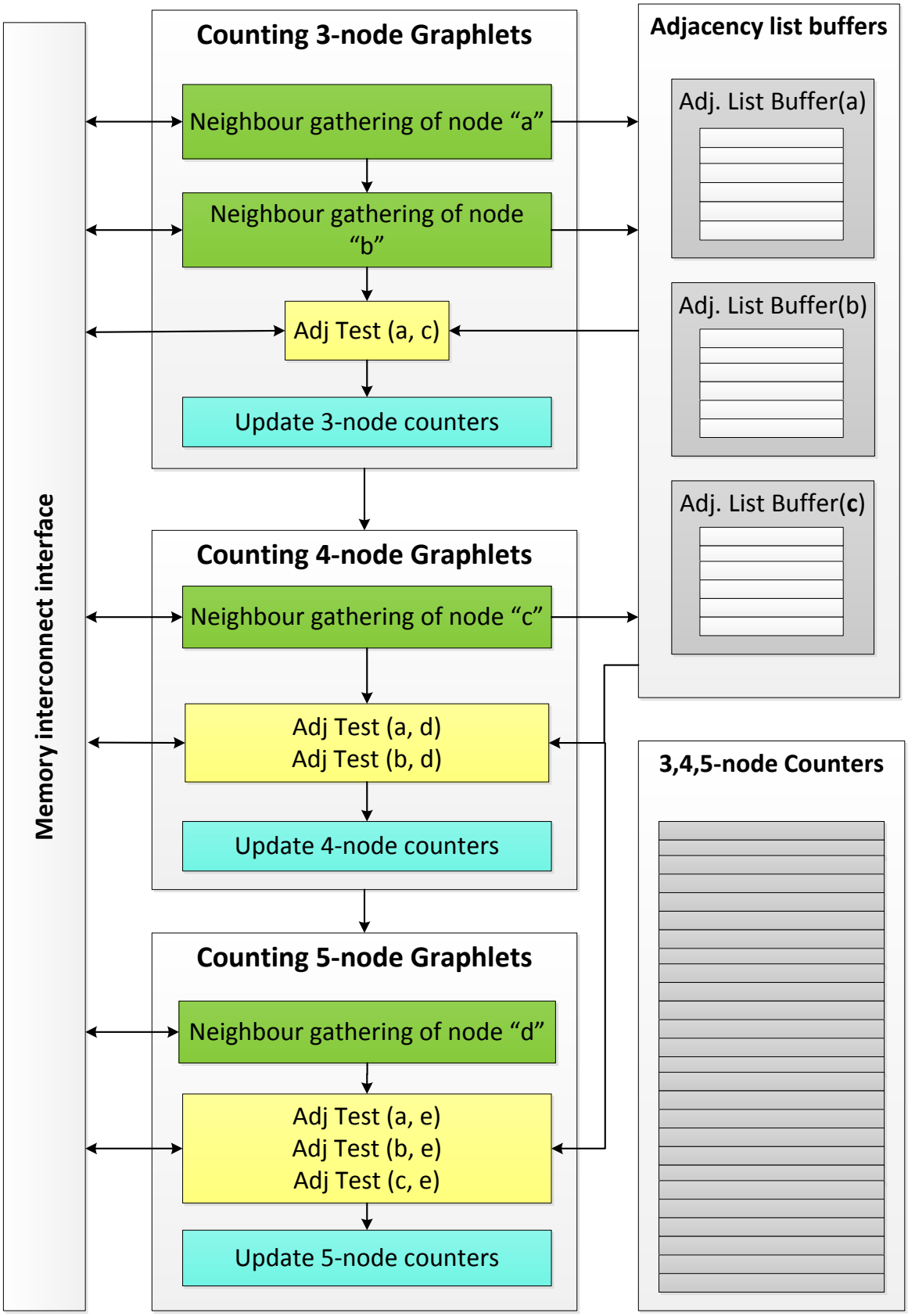


Figure 4.11: *Graph Processing Element (GPE)* design for graphlet counting.

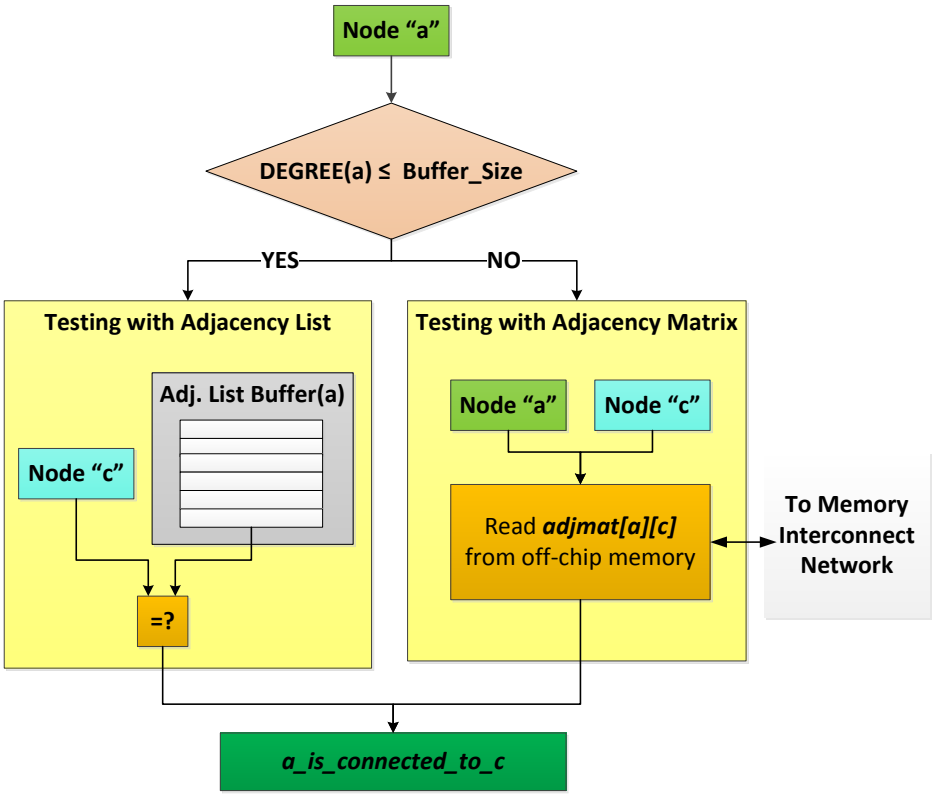


Figure 4.12: Adjacency test between nodes a and c.

4.5 Implementation and measurements

This section provides details of our experiments. For the graph data, we use two common classes of graphs: Uniformly Random graphs, and R-MAT graphs. Uniformly Random graphs are generated with n vertices each with degree d, where the d neighbours of a vertex are chosen randomly. The uniformly random graph data are generated using *GTgraph* [49], a synthetic graph generator suite. R-MAT (Scale-free) graphs [50] are characterised by their skewed degree distribution and fractal community. The scale-free graphs are generated using the Graph500 benchmark suite [5] based on the Recursive-Matrix (R-MAT) graph model. For the parameters of the R-MAT graph, we use the default values of the Graph500 benchmark (A=0.57, B=0.19, C=0.19).

To develop for Convey HC-1, we use the Convey Personality Development Kit (PDK) [51], which is a set of makefiles to support simulation and synthesis design flows. Convey provides a wrapper that allows the user to interface the FPGA design with both the host CPU and the memory controllers. Our graphlet counting hardware design is expressed in RTL using Verilog HDL, and is compiled using Xilinx ISE v13.1. Our FPGA-based graphlet counting design has 64 GPEs clocked at 120 MHz. Hardware resource utilisation is provided in Table 4.1 for both hybrid and non-hybrid graphlet counting algorithms.

Table 4.1: Device utilisation on a single Virtex5 LX330

Design	Slice registers	Slice LUTs	BRAM	DSP48
Non-hybrid design (64 GPEs)	65%	58%	62%	33%
Hybrid design (64 GPEs)	67%	68%	62%	33%

We compare the performance of our hardware graphlet counting desing with a software implementation. The source code of the graphlet counting algorithm was extracted from the GraphCrunch tool [55], an open source tool for biological network analysis developed at UC Irvine. We optimise the original code by removing unnecessary adjacency tests that were present in the original source code. We also use the Intel C compiler *icc* version 12.1.4 (gcc version 4.4.6 compatibility) to compile the code.

Table 4.2 provides the configuration details of the machines used in our experiments. The software implementation runs on a dual-socket Intel Xeon E5420 which has 4 cores (8 cores in total) clocked at 2.5 GHz. For the high performance reconfigurable computing system, we use the Convey HC-1 server [52] which has four user-programmable Virtex-5 FPGAs, and each FPGA is connected to a shared memory subsystem via a memory crossbar. Each memory controller is implemented on its own FPGA and is connected to two Convey-designed scatter-gather DIMM modules. Each AE has a 2.5 GB/s link to its corresponding memory controller, giving a theoretical aggregate peak memory bandwidth of 80 GB/s. However, the effective memory bandwidth of the AEs varies according to their memory access pattern.

Table 4.2: The specification of the machines used in our experiments

	8-core CPU	**Convey HC-1**
Core architecture	Intel Harpertown	Xilinx Virtex-5
Model No.	Xeon E5420	XC5VLX330
Fabrication process	45 nm	65 nm
Launch year	Late 2007	2006
Core frequency	2.5 GHz	120 MHz
Total Num. of CPU Cores	8	-
Total Num. of FPGA devices	-	4
Total Num. of threads/PEs	8	256
Main Memory	16 GB	16 GB

4.6 Experimental Results

In this section, we evaluate the performance of our reconfigurable hardware graphlet counter that we presented in Section 4.4. To begin we measure the performance of our hardware graphlet counter design on the Convey HC-1 machine (see Table 4.2), using R-MAT graph instances with different sizes. We then compare the performance results of our hardware graphlet counter to those of a software implementation running on a multi-core CPU (see Table 4.2).

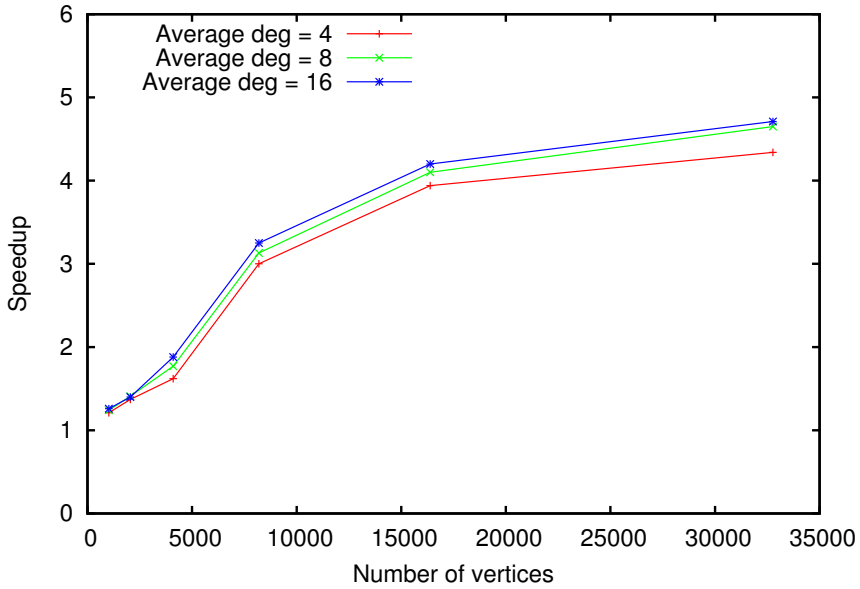


Figure 4.13: Performance speedup over the adjacency matrix approach: effects of the size of the input graph.

4.6.1 Impact of the hybrid adjacency tests

Based on the performance analysis presented in Section 4.2, we expect more than 15-25% reduction in execution time as reducing the number of random memory accesses to the adjacency-matrix will decrease the number of last level cache misses. Figures 4.13 and 4.14 shows the performance speedup of the hybrid adjacency testing approach over the adjacency-matrix approach. Figure 4.13 shows that a performance speedup of over 4 times is achieved for large graphs ($|V| > 20,000$). Small graphs ($|V| < 2500$ achieve much lower speedups (less than 1.5 times). This difference in speedup values is due to the impact of last level cache misses which increases as the graph size increases (ses Section 4.2) when the adjacency-matrix approach is used. Hence, our hybrid approach will have a bigger impact on large graphs by reducing the effect of last-level cache misses through the reduction of random memory accesses to the adjacency-matrix. Note that the size of the adjacency-list buffer have very little impact on the performance speedup values. This can be explained by the nature of power-law graphs where a large number of nodes have a low degree (less than 5).

4.6.2 Performance scalability

Figures 4.15 and 4.16 show the scalability of our graphlet counting hardware design for a single FPGA design and a multi-FPGA design respectively. For a single FPGA design the number of GPEs varies from 1 to 64,

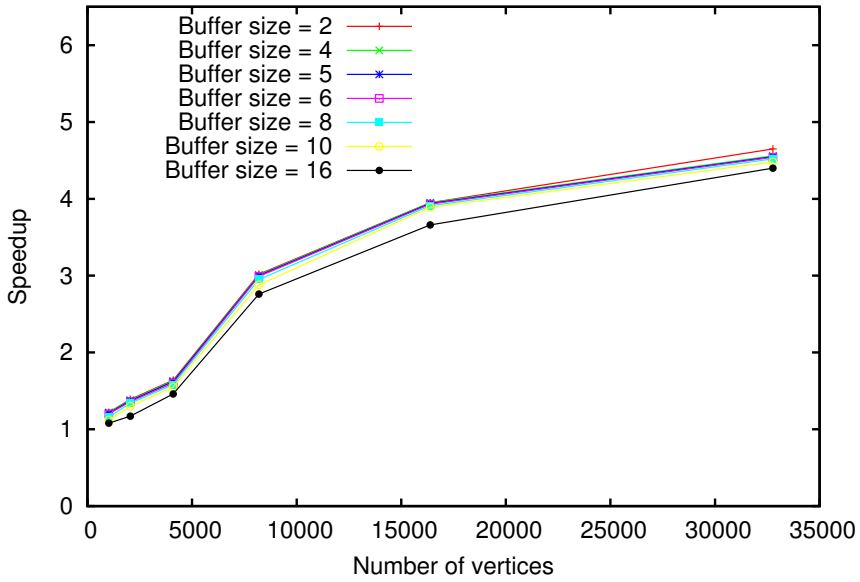


Figure 4.14: Performance speedup over the adjacency-matrix approach: effects of the size of the adjacency-list buffer.

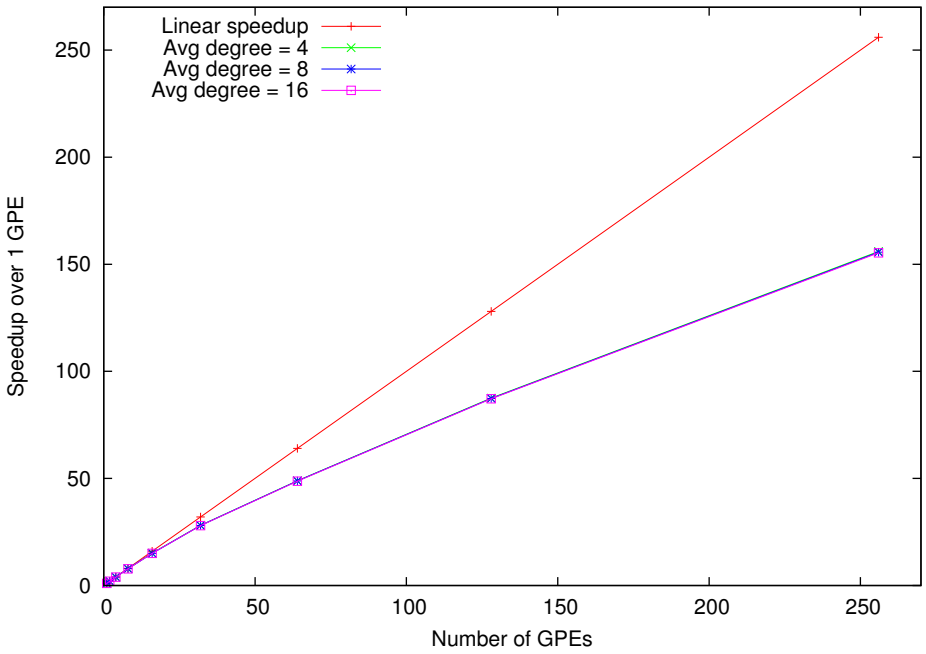


Figure 4.15: Performance scalability as parallel resources are increased from 1 GPE to 256 GPEs.

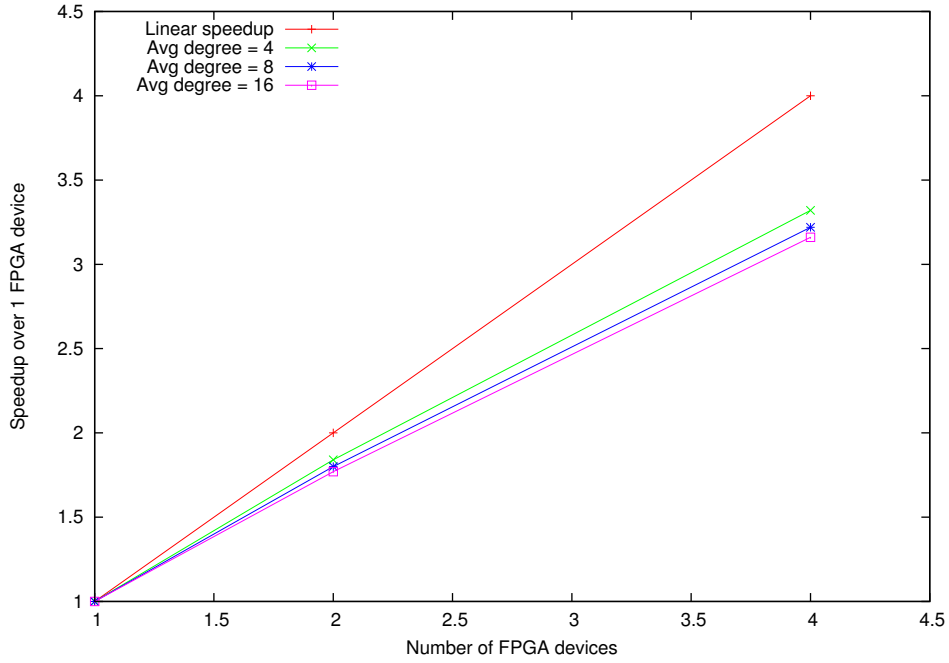


Figure 4.16: Performance scalability as parallel resources are increased from 1 FPGA device to 4 FPGA devices.

whereas for the multi-FPGA design the number of GPEs varies from 64 to 256 GPEs (1 FPGA to 4 FPGAs). The number of graph vertices is set to 32768 vertices with an average vertex degree of 4, 8, and 16. We define the efficiency as the ratio of speedup of g GPEs over 1 GPE, divided by the linear or ideal speedup, g. In our current design we are able to fit up to 64 GPEs per Virtex5 LX330 device, so we have 128 and 256 GPEs for 2 and 4 FPGA devices respectively. For a single FPGA device, we observe that our design achieves an efficiency of about 62.5% and about 75% for the single- and multi-FPGA designs respectively. This may be due to the increased memory bank contention leading to non-linear performance speedups as the number of GPEs is increased.

4.6.3 Performance comparison

Figures 4.17 and 4.18 compare the performance of our reconfigurable hardware graphlet counter to the performance of a multi-core CPU implementation. The details of the multi-core CPU and the FPGA-based HPC platforms are provided in the previous section (Table 4.2). Note that the CPU is based on 45nm technology, whereas the FPGAs we use in our work are based on 65nm technology. We use R-MAT graphs with variable size and average node degree. For our FPGA-based graphlet counting design, we use 4 FPGAs resulting in a

total of 256 GPEs. Both the hardware and software implementations use our proposed hybrid graphlet counting algorithm.

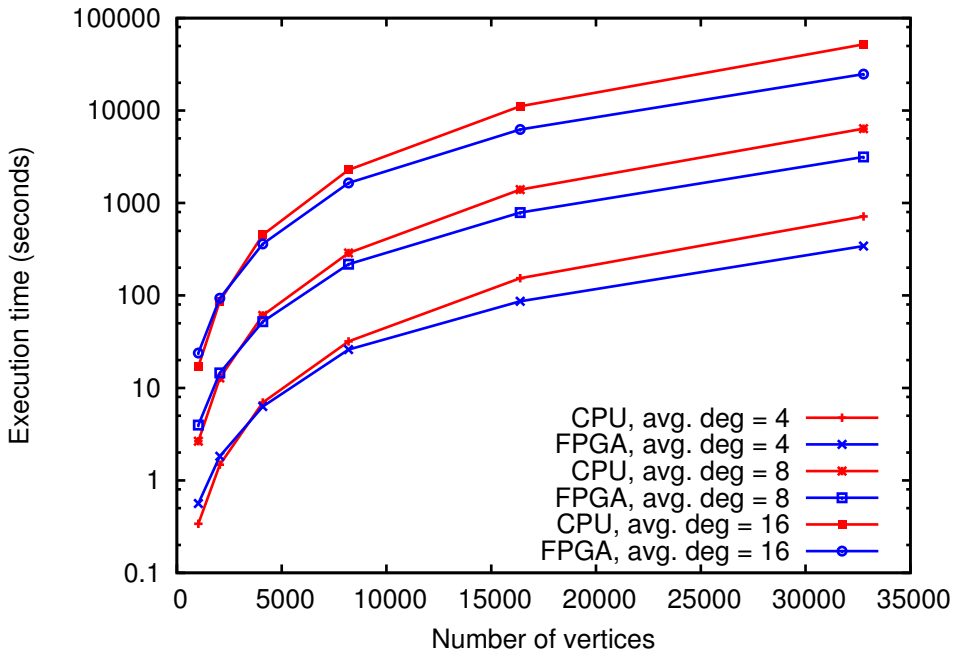


Figure 4.17: Performance comparison of multi-core CPU and HC-1: execution time.

A first observation that can be made from the performance results in Figure 4.17, is the long execution times of the graphlet counting algorithm for large graph problems. These long execution times, which can be as long as several hours (over 14 hours for an R-MAT graph with 32768 vertices and an average degree of 16), highlights the importance of accelerating such algorithms. This can be explained by the increasing number of last level cache misses in the CPU as the size of graph data increases, due to poor memory access locality characteristics that are exhibited by graph algorithms in general. For large graphs, the hardware implementation outperforms the software implementation by a factor of over 2 times. This may not seem like a great performance speedup, but it helps halving the execution time for large graphs from 14 hours to about 7 hours.

From these comparison results, we can say that our FPGA design outperforms the multi-core CPU implementations, as it is able to exploit higher parallelism through 256 application-specific GPEs compared to 8 threads for the quad-core CPU. In addition, as the graph size increases, the performance gap between HC-1 and the CPUs increases too. This is mainly due to the increased number of random memory accesses issued in the graphlet counting algorithm to perform adjacency tests distance of neighbouring. By issuing a large number of concurrent memory requests (up to 4x112x16=16384 concurrent requests), our FPGA design can cope better

with irregular memory accesses compared to the cache-based CPU.

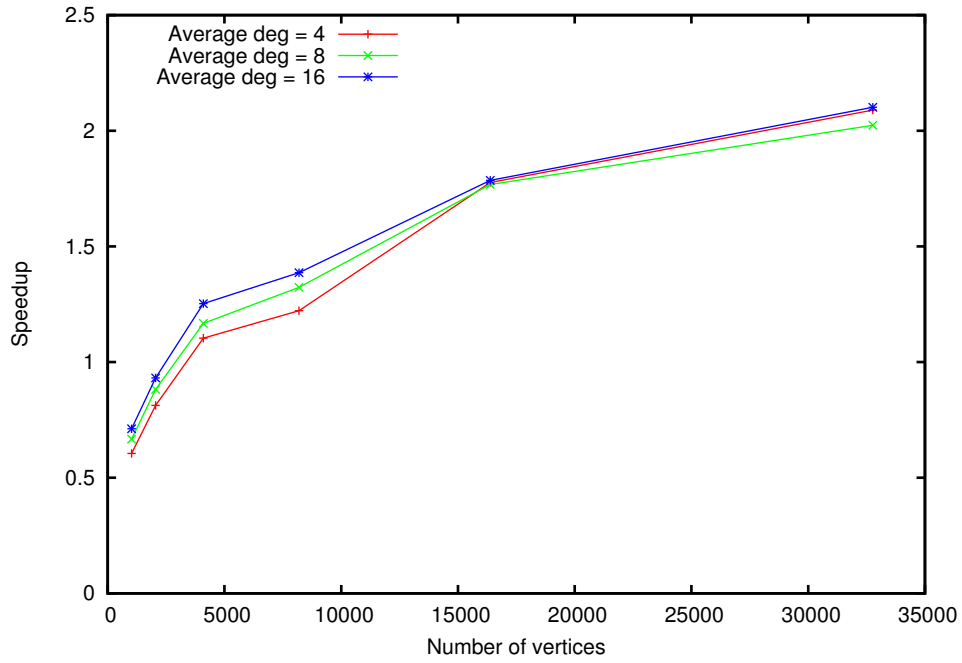


Figure 4.18: Performance speedup of HC-1 over multi-core CPU.

4.7 Summary

This chapter presented a reconfigurable hardware acceleration of the graphlet counting algorithm using our reconfigurable computing approach for graph traversal problems presented in Chapter 3. Unlike in the case of BFS where only the adjacency-list representation of graphs is used, efficient processing of the graphlet counting algorithm requires both the adjacency-list representation as well as the adjacency-matrix representation of the input graph data. We propose a new hybrid graphlet counting algorithm where adjacency tests are performed using both the adjacency-lists and the adjacency-matrix. Experimental results shows that our hybrid approach can improve the performance of the graphlet counting algorithm by a factor of 4 times. We also present a reconfigurable hardware graphlet counter that is up to 2 times faster than a quad-core CPU. In the next chapter, we show how on-chip memory can be used to accelerate graph traversal problems.

Chapter 5

Parallel FPGA-based All-Pairs Shortest-Paths for Sparse Networks

In this chapter, we present our FPGA-based accelerator for the *All-Pairs Shortest-Paths* (APSP) problem for an unweighted graph. This work has been motivated by a computationally intensive bio-informatics application that employs the APSP algorithm to analyse multi-subject voxel-based human brain networks [2]. Wang *et al.* [2] report that it takes up to 1 hour and 52 mintues to perform APSP on a human brain network based on BOLD signals (Blood Oxygen Level Dependent) using an AMD Phenom II X4 965 quad-core CPU system running at 3.4 GHz. As the size of the networks becomes larger, solving the APSP problem becomes intolerably time-consuming. Hence, a solution with a strong computational capability will greatly benefit *Brain Network Analysis*(BNA) research.

In previous chapters, we have shown that by using our reconfigurable computing approach we can accelerate graph traversal algorithms. In both cases, BFS and graphlet counting, we do not use on-chip memory resources to store working data sets. In this chapter, we leverage the benefits of on-chip memory resources to accelerate the APSP problem for very sparse graphs. Our main contributions are:

- A reconfigurable hardware parallelisation methodology that we adopt to design a parallel and scalable FPGA-based solver for the APSP problem for sparse networks (Section 5.3).
- A detailed description of a reconfigurable hardware accelerator for the APSP problem, including a novel way to leverage the power of distributed on-chip memory resources to reduce off-chip memory accesses (Section 5.4).

- Design optimisations that yield different FPGA bitstreams which are selected at run-time based on the input graph data (Section 5.5).
- An analytical model to estimate the maximum amount of achievable parallelism for a given input graph size and a target FPGA device (Section 5.6).
- An in-depth performance evaluation that analyses performance scalability, and the effects of different design optimisations using real-word data (Section 5.8).
- A performance comparison with CPU and GPU implementations using human brain network data, showing that our FPGA design outperforms both the CPU and GPU implementations (Section 5.8).

The rest of this chapter is organised as follows. Section 5.1 provides background information about the APSP problem. In Section 5.3, we discuss the design consideration of an FPGA-based APSP, and provide an overview of our parallel FPGA design. Section 5.4 provides a detailed description of our FPGA design for APSP along with some design optimisations. Finally, we present results of our experiments in Section 5.8.

5.1 Background

5.1.1 All-Pairs Shortest Path

The All-Pairs Shortest-Paths (APSP) problem is defined as follows. Given a weighted, directed graph $G = (V, E)$ with a weight function, w: $E \rightarrow R$, that maps edges to real-valued weights, we wish to find, for every pair of vertices $u, v \in V$, a shortest (least weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. There are mainly two classes of APSP algorithms. One class is Johnson's algorithm [60], which is based on single-source shortest path algorithms such as Dijkstra algorithm [61] and Bellman-Ford algorithm [62]. When applied to unweighted graphs, Johnson's algorithm reduces to Breadth-First Search (BFS). Johnson's algorithm and BFS are efficient with sparse graphs but perform poorly with dense graphs. Another class is the Floyd-Warshall (FW) algorithm [63], which has $\mathcal{O}(|V|^3)$ time complexity and favours dense networks. In our work, we consider unweighted graphs and hence focus on using BFS to solve APSP. We refer to it in this paper as APSP-BFS (Algorithm 14). In many applications, such as human BNA [64], networks with different sparsity values are processed, so both algorithms can be useful for optimum performance results.

Algorithm 14: Sequential APSP-BFS algorithm

Input: Graph G(V,E)
Output: Array *distance*$[1..n][1..n]$ with *distance*$[u][v]$ holding the shortest path distance from u to v

```
1 foreach v ∈ V do
2     Invoke BFS_KERNEL(v)
```

5.1.2 Queue-based BFS

Algorithm 15 describes the standard sequential BFS algorithm. CQ (current queue) is used to hold the set of vertices that must be visited at the current BFS level. At the beginning of a BFS, CQ is initialised with v_s. As vertices are dequeued, their neighbours are examined. Unvisited neighbours are labelled with their distance (BFS level) and are enqueued for later processing in NQ (next queue). After reaching all nodes in a BFS level, CQ and NQ are swapped. This algorithm performs linear $\mathcal{O}(|V| + |E|)$ work since each vertex is labelled exactly once and each edge is traversed exactly once.

Algorithm 15: Simple sequential BFS

Input: Vertex set V, source vertex v_s
Output: Array *distance*$[1..n]$ with *distance*$[i]$ holding the minimum distance of v_i form v_s
Data: CQ: queue of vertices to be explored in current level;
NQ: queue of vertices to be explored in next level

```
1  CQ ← ∅
2  distance[] ← ∞
3  distance[s] ← 0
4  CQ ⟵ {v_s}
5  bfs_level ← 0
6  while (CQ != ∅) do
7      NQ ⟵ ∅
8      for all v_i ∈ CQ do
9          v ← Dequeue CQ
10         foreach u_j adjacent to v do
11             if distance[j] == ∞ then
12                 distance[j] ⟵ bfs_level + 1
13                 NQ ← Enqueue(u_j)
14     bfs_level ← bfs_level + 1
15     Swap(CQ, NQ)
```

5.2 Related Work

The importance of efficient processing of the APSP problem has led to a substantial amount of previous work that deals with the design and optimisation of APSP either for commodity processors [3, 10, 65], or for dedi-

cated hardware [12, 14, 13]. Matsumoto *et al.* [3] proposed a CPU-GPU hybrid system that is reported to have outperformed previous work on APSP for commodity processors.

Much previous work on using FPGAs to solve Shortest-Paths problems has used low-latency on-chip memory resources to store graph data [34, 13]. These solutions are not suitable for large graph problems that require high-latency off-chip storage. In our work, we present a reconfigurable hardware architecture to accelerate the APSP algorithm for graph problems that require high-latency off-chip storage.

To the best of our knowledge, no previous FPGA work has tackled the APSP problem for sparse graphs with unweighted edges. Bondhugula *et al.* [12] proposed a parallel FPGA design to accelerate the FW algorithm which is more suitable for dense networks, and is proven to be less efficient for very sparse networks. We consider our work complementary to Bondhugula *et al.*'s work.

5.3 Reconfigurable Hardware Parallelisation of APSP-BFS

Graph algorithms have a low computation to access ratio [6], where the algorithm is often traversing the vertices and edges of a graph while doing very little computation per vertex or edge. In other words, graph algorithms are in general memory-latency bound. Achieving good performance levels on FPGAs will require a design that exploits parallel on-chip memory resources to reduce off-chip memory accesses. Having said that, parallelism achieved for the APSP problem will likely be limited by the amount of on-chip memory resources. So choosing carefully how to use on-chip memory resources will prove key towards achieving high parallelism, and subsequently high performance. The sequential APSP-BFS algorithm (Algorithm 14 in Section 5.1) can be parallelised using one of the following three methods:

1. Globally-shared on-chip memory: The outer-loop in Algorithm 14 executes serially a parallel BFS kernel for each vertex in the graph. Only one BFS kernel is running at any time. So all the parallel resources are dedicated to executing one BFS kernel. This is illustrated in Figure 5.1, where all the GPEs have access to globally shared on-chip memory. One drawback of this method is that on-chip memory resources will be shared which will lead to high access contention. This access contention results in higher latencies to access shared on-chip memories, which defies the purpose of using low-latency on-chip memories in the first place. For example an on-chip *bitmap* implemented on FPGA BRAM will have a latency of one or two clock cycles. However, if there are 64 processing elements (PEs) trying to access it at the same time, then this access latency jumps from 1 or 2 clock cycles to more than 64 clock cycles. This will translate onto poor scalability as

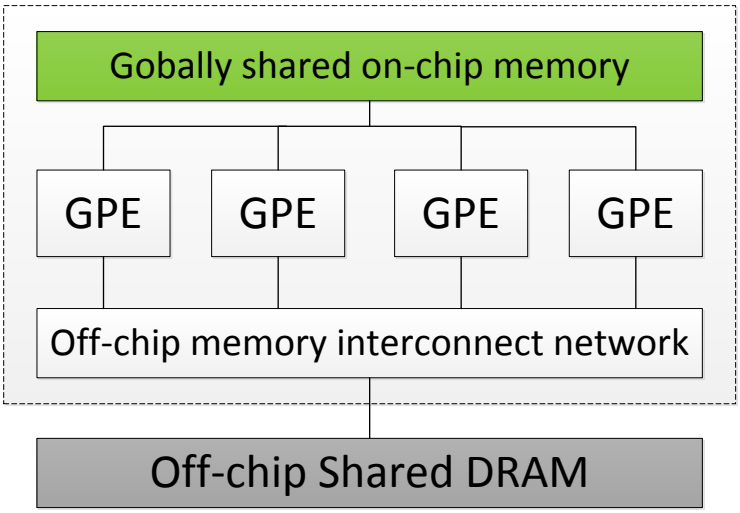


Figure 5.1: Reconfigurable hardware architecture template with globally shared on-chip memory

increasing the number of PEs will increase access contention overhead.

2. Private on-chip memories. The outer-loop in Algorithm 14 runs in parallel a number of sequential BFS kernels. This approach allows for private on-chip memory resources that are exclusive to a PE, as each PE is executing a different BFS kernel. This is illustrated in Figure 5.2. An advantage of this approach is performance scalability as the number of PEs is increased. However, one drawback of this method is that assigning exclusively on-chip memory resources to a specific PE may limit the number of PEs by the available on-chip memory resources, even if there are many unused hardware resources on an FPGA such as LUTs and DSP blocks.

3. Locally-shared on-chip memories: This is the general case where the outer-loop in Algorithm 14 executes in parallel a number of parallel BFS kernels. On-chip memories are locally shared as illustrated in Figure 5.3This method can be a compromise between the two previous methods by tolerating access contention up to a certain degree, while avoiding parallelism being limited by on-chip memory resources. If the number of PEs executing the same BFS kernel is kept small enough, then access contention overhead remains negligible.

Table 5.1 summarises the advantages and disadvantages of each parallelisation method. As explained above, using a globally-shared on-chip memory to parallelise APSP-BFS has the major drawback of high access contention to shared on-chip memory resources, and hence, it will not be considered in this work. Selecting either of the second method or the third method depends mainly on the size of the input graph data. Up to a given graph size, $|V|$, using private on-chip memories (method 2) will allow for high utilisation of on-chip memory resources as well as other hardware resources such as Slice LUTs and registers. As $|V|$ becomes relatively

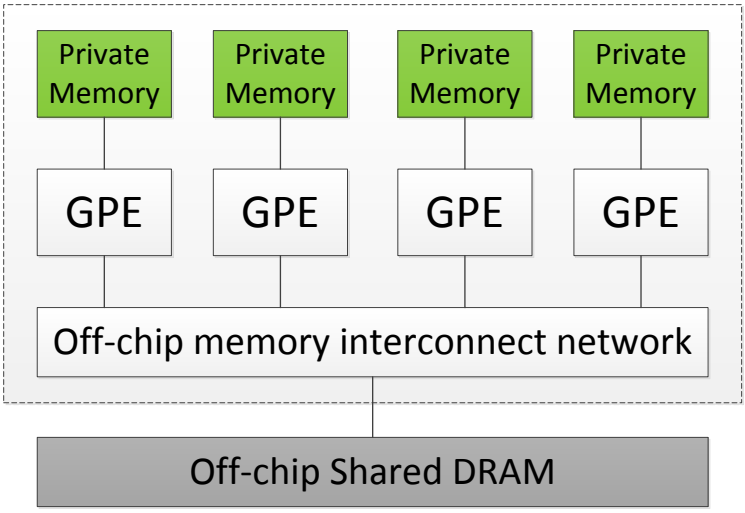


Figure 5.2: Reconfigurable hardware architecture template with private shared on-chip memories

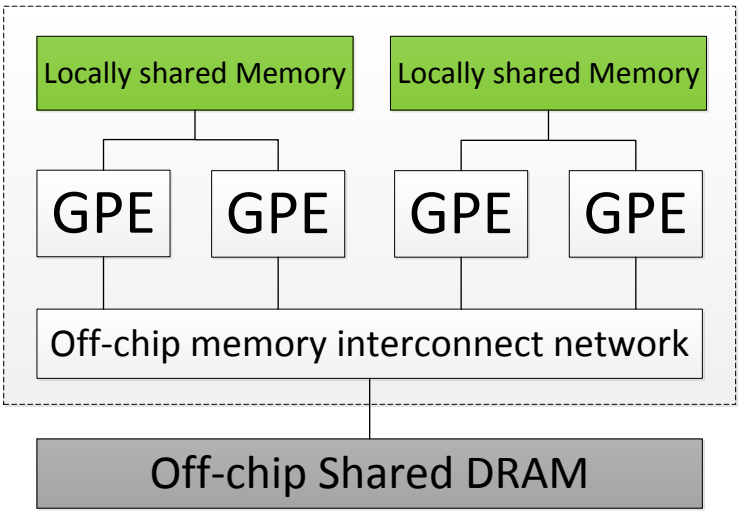


Figure 5.3: Reconfigurable hardware architecture template with locally shared on-chip memory

Table 5.1: Advantages and disadvantages of hardware parallelisation methods of APSP-BFS.

Parallelisation	**Advantages**	**Drawbacks**
Globally-shared on-chip memory	• can process very large graphs in terms of vertex count	• High access contention to shared on-chip memory resources • Poor performance scalability
Private on-chip memories	• No access contention overhead • Great performance scalability: near-linear	• Number of PE: limited by on-chip memory resources • can process relatively smaller graphs (in terms of vertex count)
Locally-shared on-chip memories	• Low access contention as long as the number of PEs sharing memory resources is kept small • can process large graphs	• Extra logic circuitry to arbiter access to shared memory resources

large, using private memories results in low utilisation of hardware resources other than on-chip memory resources. At this point, using the third method (locally-shared on-chip memories) is more efficient as it allows for on-chip memories to be locally shared by a set of exclusive PEs. We can say here that the second method is a special case of the third method where the number of PEs sharing the same memory is one. In this work, we adopt the second method for our FPGA-based APSP-BFS as the size of off-chip memory of our target platform limited the size of the input graphs (see Table 5.2 and Section 5.7). The evaluation of the third method is left for future work as more off-chip memory storage becomes available.

Using the reconfigurable hardware architecture template for parallel graph traversal presented in the previous chapter,the overall architecture of the FPGA-based APSP-BFS is shown in Figure 5.4. The template which resembles a scalable many-core style processor architecture, comprising a Control Processor (CP), multiple Graph Processing Elements (GPEs), and a memory interconnect network that is augmented with on-chip private memories. The CP manages the operation of the GPEs, including initialisation, task assignment, and synchronisation of the GPEs. Each GPE has a private on-chip memory accessible only to itself, while the memory interconnect network links the GPEs to an off-chip shared-memory subsystem. The GPEs are a collection of replicated and parallel processing elements that implements the serial BFS algorithm for APSP. Each GPE can independently execute a BFS on given graph node.

5.4 Parallel FPGA Design for APSP-BFS

In this section, we describe how we parallelise the APSP-BFS algorithm using our reconfigurable hardware architecture template presented in Section 5.3. In our approach, we choose to parallelise APSP-BFS algorithm using the sequential version of BFS (Algorithm 1). As for graph representation, we use the popular CSR

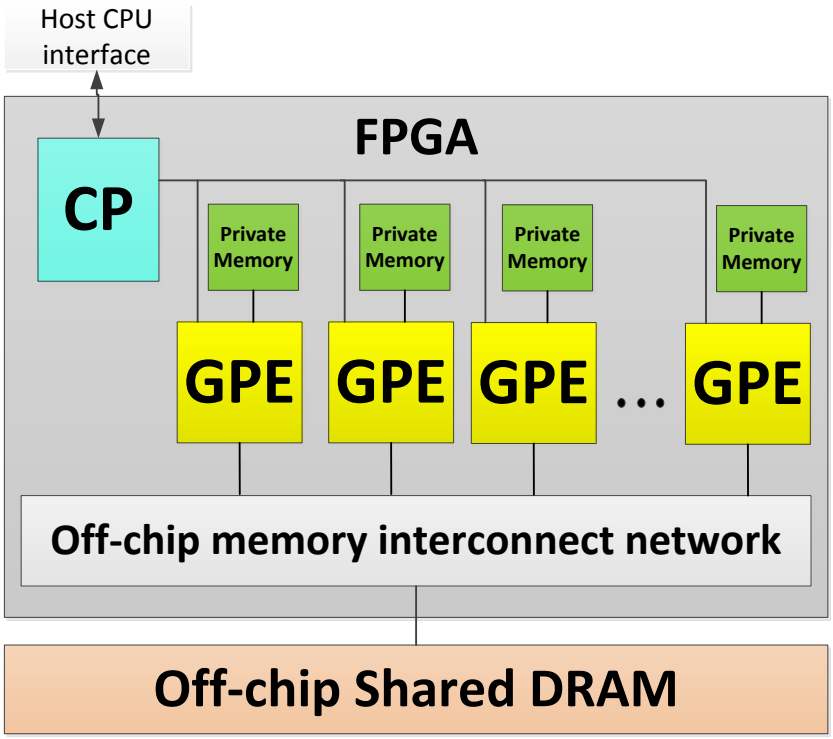


Figure 5.4: Reconfigurable hardware architecture template for parallel graph exploration algorithms

(Compressed Sparse Row) format which merges the adjacency lists of all vertices into a single $\mathcal{O}(|E|)$-sized array, with the beginning location of each vertex's adjacency list stored in a separate $|V|$-sized array. For each BFS, we require an $|V|$-sized array, the *distance* array, to store the BFS level of each vertex, and hence we require an $|V|^2$-sized array to store all the APSP-BFS results.

We start by breaking the APSP-BFS algorithm into two parts: one part running on the control processor (CP) (Algorithm 14, line 1), and the other part on the GPEs (Algorithm 1). The CP dispatches tasks to GPEs, by issuing source vertices for BFS execution. It starts by initialising the GPEs and waits for GPE requests which are queued up in a FIFO-based queue. Once all the source vertices have been dispatched the CP issues a termination signal to the host CPU to indicate that the APSP routine completed execution.

5.4.1 GPE design

Since the BFS problem is memory latency bound, we devise a GPE design approach based on three key design ideas: (i) efficient utilisation of BRAM resources on FPGAs to reduce off-chip memory accesses, (ii) prioritise reducing random accesses over regular accesses, and (iii) pipelining access to off-chip shared memory.

To reduce off-chip memory accesses, we design a Bitmap scheme to store the visitation status of all vertices in the graph, and emulate the queues used in the sequential BFS kernel. We use one bitmap, the *Status Bitmap*, for the visitation status of vertices, and two bitmaps for queue emulation, the *CQ_bitmap* and the *NQ_bitmap*. Each bitmap has $|V|$ bits, and is implemented as dual-port RAM that maps onto FPGA BRAMs. CQ_bitmap and NQ_bitmap are used as follows: if a vertex has been enqueued, then the corresponding bit in the bitmap is set to 1, otherwise this bit is set to 0. This means the maximum size of the CQ_bitmap and NQ_bitmap is $|V|$ bits, which can be stored entirely in on-chip memories. In contrast, using a standard FIFO queue requires $log_2(|V|)$ bits per vertex, and approaches $|V| \times log_2(|V|)$ bits for the worst case. This worst case scenario may lead the queue to spill to slow off-chip memory as fast on-chip memories are exhausted.

In addition, we use true-dual port RAM with two differently sized ports: a 1-bit port and a 64-bit port. The read/write 64-bit port is used to initialise the bitmaps as well as streaming the content of these bitmaps, in particular the CQ_bitmap. The 1-bit port is used for random accesses to these bitmap such as in the case of the $Status\_bitmap$ that stores the visitation status of vertices in a graph.

A key decision is to select data to store in on-chip memory and off-chip memory. Our main selection criterion is access patterns. We use on-chip memories to reduce random or irregular accesses. For example, in the BFS kernel (Algorithm 16), the visitation status of vertices requires random memory accesses (Algorithm 16, line 15). Instead of reading the $distance$ array from memory to determine the visitation status of a vertex, we store the visitation status of vertices in an on-chip bitmap, which can be accessed in one clock cycle, saving on up to tens of clock cycles.

Finally, pipelining memory accesses is achieved by enabling the GPE to sequentially issue multiple outstanding memory requests to a multi-bank memory subsystem, and use on-chip RAM resources to store data from memory for subsequent processing. Instead of issuing one memory request, and then waiting for response from memory, the GPE issues *multiple non-blocking memory requests* to take advantage of the capabilities of the multi-bank memory subsystem. Assuming that the requests are destined for different memory banks, the off-chip memory latency of a single memory request is amortised over multiple memory requests as they get serviced simultaneously.

Figure 5.5 presents a schematic overview of the GPE design for the BFS kernel (Algorithm 16) that incorporates the design choices discussed above. In Algorithm 16, data defined as **Input** and **Output** is stored in off-chip memory, whereas data defined as **Data** is stored in on-chip memories. A step-by-step description of the GPE design follows:

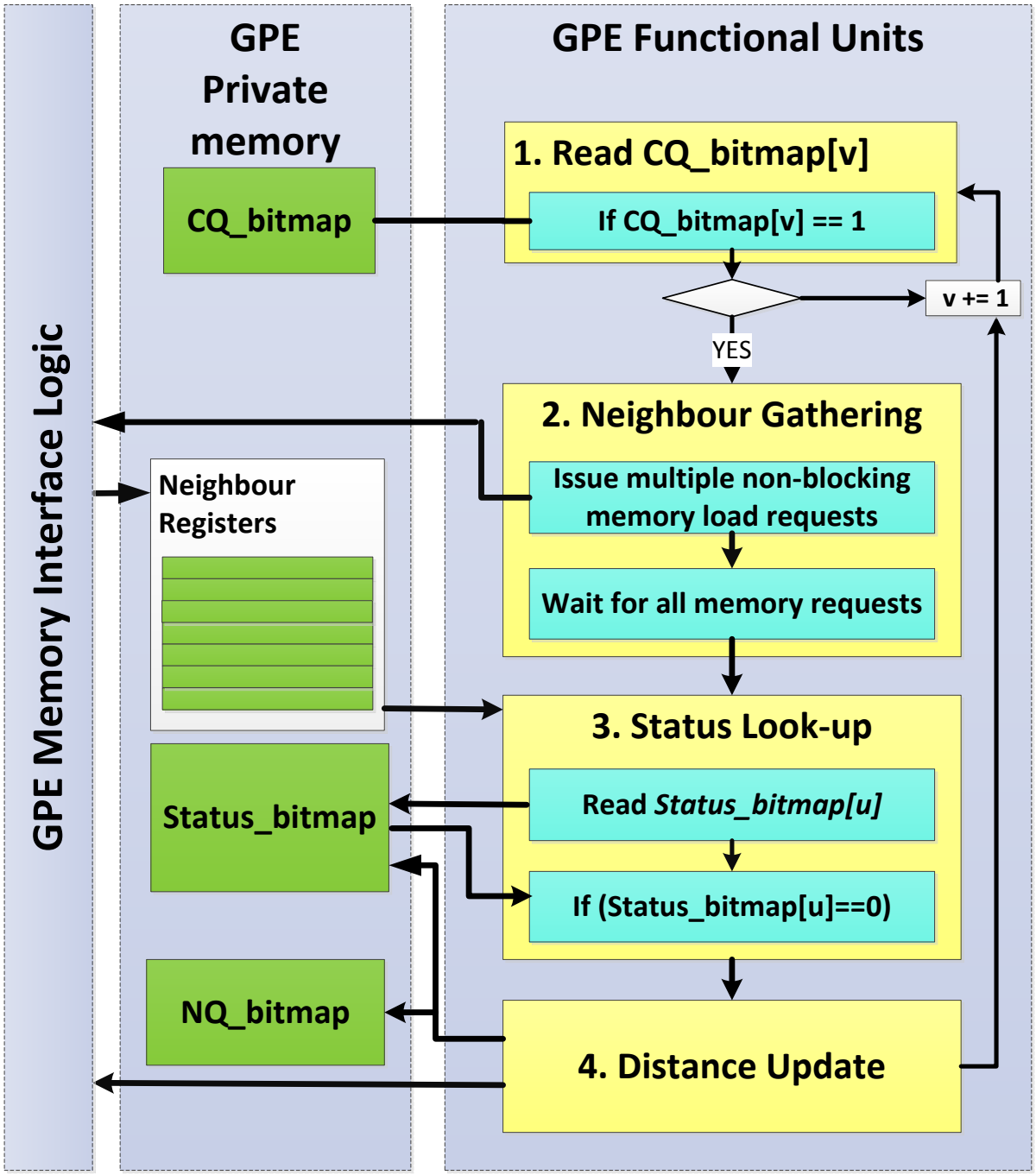


Figure 5.5: *Graph Processing Element (GPE)* design for APSP-BFS.

Algorithm 16: BFS kernel executed by each GPE

Input: R[1..n]:offsets of adjacency lists, C[1..m]: adjacency lists.
Output: Array *distance*[1..n] with *distance*[i] holding the minimum distance from v_s to v_i
Data: $NID[1..q]$: 16-bit GPE registers to store Neighbour IDs,
$Status\_bitmap[1..n]$: stores visitation status of vertices,
$CQ\_bitmap[1..n]$: stores marked vertices for current BFS level,
$NQ\_bitmap[1..n]$: stores marked vertices for next BFS level,
q: number of NID registers

```
1  bfs_level ⟵ 0
2  distance[s] ⟵ bfs_level
3  CQ_bitmap ← NQ_bitmap ← Status_bitmap ← 0
4  CQ_bitmap[s] ← Status_bitmap[s] ← 1
5  repeat
6      done ← 1
7      foreach v ∈ 1..n do
           // Step 1:  read CQ_bitmap[v]
8          if (CQ_bitmap[v]) then
9              CQ_bitmap[v] ← 0
10             for (offset ← R[v]; offset < R[v+1]; offset += q) do
                   // Step 2:  Neighbour gathering
11                 foreach i ∈ 1..q do
12                     NID[i] ← C[offset + i]
13                 foreach i ∈ 1..q do
14                     u ← NID[i]
                       // Step 3:  status look-up
15                     if (Status_bitmap[u] == 0) then
                           // 4.  distance update
16                         distance[u] ← bfs_level + 1
17                         Status_bitmap[u] ← 1
18                         NQ_bitmap[u] ← 1
19                         done ← 0
20     bfs_level = bfs_level + 1
21     Swap(CQ_bitmap, NQ_bitmap)
22 until (done);
```

1. **Read *CQ_bitmap***. This unit checks whether a v_i belongs to the current BFS level by reading *CQ_bitmap* (Line 8). If $CQ\_bitmap[v_i]$ is set, its neighbours are explored in the current iteration (steps 2-4). To optimise reading the CQ_bitmap, 64 values are read at the same time by using the 64-bit read port of the true dual port ram that implements the CQ_bitmap.

2. **Neighbour gathering**. The neighbours of v_i are retrieved from memory in q-sized batches using multiple non-blocking memory requests (Lines 11-12). The retrieved neighbouring vertices are stored in local registers (NID registers). For area-efficiency reasons, these registers are implemented using distributed RAM resources instead of Slice Registers. After each batch is retrieved from memory, steps 3 and 4 are

executed (Functional unit 3 and 4 in Figure 5.5) before the next batch of neighbours is read from memory.

3. **Status look-up and** . The visitation status of the gathered neighbours is checked (Line 15) by reading *Status_bitmap* using the 1-bit read port.

4. **Distance update**. Unvisited vertices will have their distance value updated to the current BFS level plus one (Line 16). $Status\_bitmap$ and NQ_bitmap are also updated accordingly (Lines 17 and 18).

5.5 Design optimisations

In this section, we present two optimisation techniques to improve the performance of our FPGA-based APSP-BFS design: graph data encoding, and hybrid BFS. The impact of these optimisations is evaluated in Section 5.8.1.

5.5.1 Adjacency lists encoding

This optimisation is platform-dependent: we consider the size of the native memory word of the target platform in bits, and encode our adjacency list such that for each memory load operation we get more than one neighbouring vertex from the adjacency list. So if the native memory word is k-bit wide, we can obtain $\lfloor k/w \rfloor$ neighbouring vertices per memory operation if we use $w = \lceil \log_2 |V| \rceil$ bits to represent neighbouring vertices. Figure 5.6 illustrate this idea with k=64 and w=16. An adjacency list must be padded to multiples of 64 bits if the number of neighbouring vertices is not a multiple of 4.

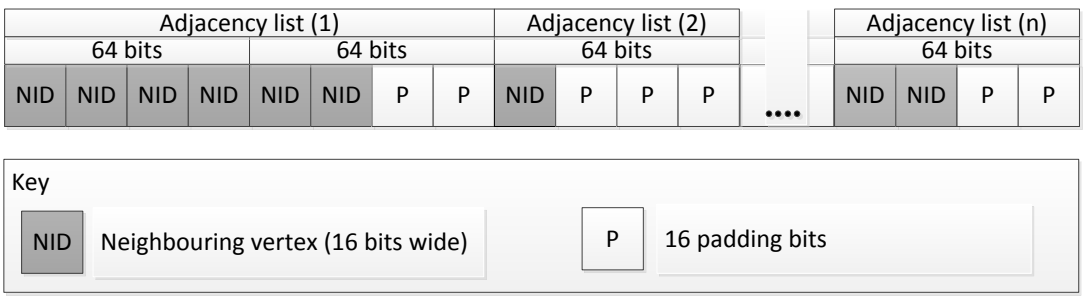


Figure 5.6: The encoding format of the adjacency list

5.5.2 Hybrid BFS kernel

Beamer *et al.* [48] presented a hybrid approach to the BFS algorithm that combines the conventional top-down approach from Algorithm 1 with a bottom-up approach. This hybrid approach takes advantage of the small-world property of real-world graphs [46] to significantly reduce the number of edges examined, and hence speed up the BFS kernel. Because of the small-world phenomenon the number of vertices in each BFS level grows very rapidly, leading to most edges being examined in one or two BFS levels, the *critical* BFS levels. Algorithm 17 describes the bottom-up algorithm that replaces Lines (7-19) in Algorithm 16 when processing critical BFS levels. In terms of hardware implementation, the GPE design can be modified so that it can execute the top-down approach for non-critical BFS levels, and the bottom-up approach for critical BFS level. Since the bottom-up approach (Algorithm **??**) is very similar to the top-down approach (Algorithm 16), much of the hardware circuitry that results from implementing the top-down algorithm can be re-used for the bottom-up algorithm.

Algorithm 17: Bottom-up algorithm for BFS kernel

```
1  foreach v ∈ 1..n do
2      if (Status_bitmap[v] == 0) then
3          for (offset ← R[v]; offset < R[v+1]; offset += q) do
4              foreach i ∈ 1..q do
5                  NID[i] ← C[offset + i];
6              foreach i ∈ 1..q do
7                  u ← NID[i];
8                  if CQ_bitmap[u] then
9                      distance[v] ← bfs_level + 1 ;
10                     Status_bitmap[v] ← 1;
11                     NQ_bitmap[v] ← 1;
12                     done ← 0;
13                     break;
```

The decision to switch from the top-down to bottom-up BFS is taken just before the start of a new BFS iteration (or level). This decision depends on the estimated number of edges that will be explored in the new BFS iteration that is about to be executed. This estimation is performed by summing the degrees of the neighbouring vertices explored in the previous iteration. If the sum is less than a certain threshold, than bottom-up approach is selected, otherwise the top-down approach is used. This threshold is determined heuristically, and for this work we set the threshold to about 30% of the total number of edges, $|E|$, in the graph. Figure 5.7 illustrates the finite state machine that implements this switching mechanism.

In addition, the switching mechanism from top-down approach to bottom-up approach requires knowledge about the sum of the degree of the vertices marked for the next iteration. So this means that when neighbouring

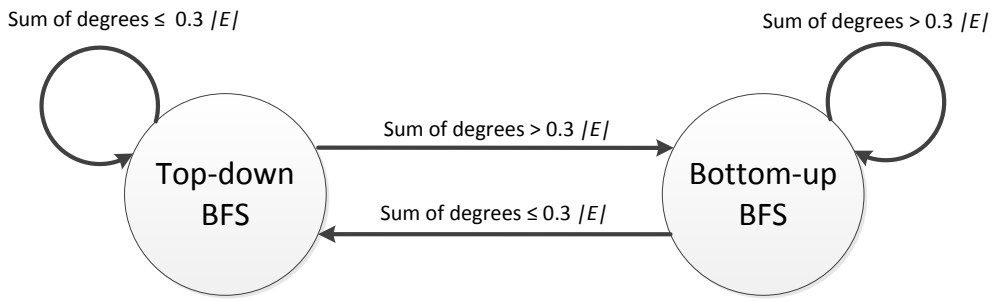


Figure 5.7: FSM diagram for switching mechanism between top-down BFS and bottom BFS

vertices are marked for the next BFS iteration, their degrees need to be obtained, leading to extra random memory accesses. In order to avoid these random memory accesses, the encoding of the adjacency list is modified as follows: 32 bits are used for each neighbouring vertex, with bits 0-15 used to store the vertex ID, while bits (16-31) store the degree of the vertex. This is illustrated in Figure 5.8.

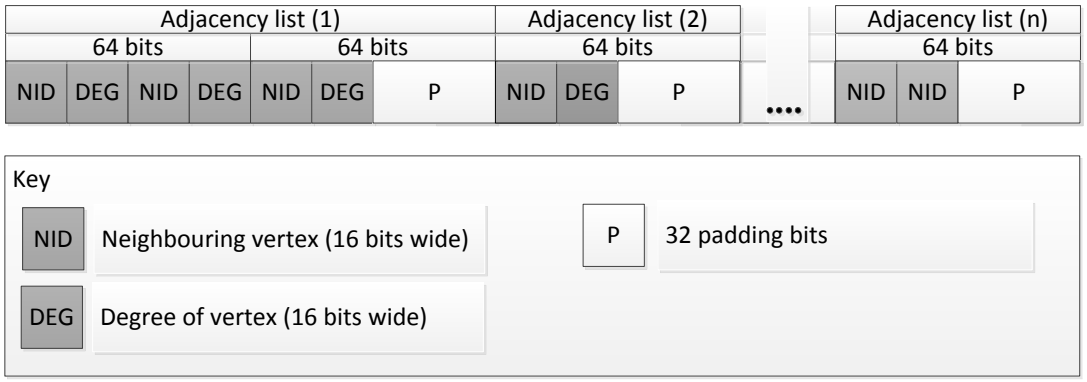


Figure 5.8: The encoding format of the adjacency list for Hybrid BFS

5.5.3 Run-time configuration selection

Since the hybrid BFS kernel requires 32 bits per neighbouring vertex (to store degree information), only one of the above optimisations can be present in an FPGA configuration. During BFS execution, the top-down algorithm (Algorithm 16) is used when the size of CQ is small, and the bottom-up algorithm is used when the size of NQ is large. In our work, we use human brain networks [64] with different diameters ranging from 7 to 32. Since we are executing BFS n times, we can afford to run BFS once on the host CPU to find out if there are any critical BFS levels. In the event of the existence of critical BFS levels, we opt to to use the hybrid algorithm. In our experiments, a BFS level is considered critical if the number of edges examined is above 30% of the total number of edges.

Table 5.2: On-chip and off-chip memory storage requirements for our FPGA-based APSP design.

$\|V\|$	On-chip memory (BRAM)		Off-chip memory (DRAM)
	1x GPE	32x GPE	
1024	3 kb	96 kb	4 MB
2048	6 kb	192 kb	16 MB
4096	12 kb	384 kb	64 MB
8192	24 kb	768 kb	256 MB
16384	48 kb	1536 kb	1024 MB
32768	96 kb	3072 kb	4096 MB
65536	192 kb	6144 kb	16384 MB

Table 5.3: Utilisation of slice registers and LUTs in our GPE implementation of the BFS kernel for the APSP-BFS algorithm.

$\|V\|$	Virtex-5 LX330		Virtex-6 LX760	
	Slice Regs	Slice LUTs	Slice Regs	Slice LUTs
1024	836	1013	828	982
2048	849	1032	841	990
4096	863	1060	855	1073
8192	875	1081	867	1086
16384	888	1103	880	1106
32768	901	1117	893	1119
65536	916	1183	908	1177

5.6 Estimating Amount of Achievable Parallelism

In our FPGA-based APSP-BFS design, the maximum number of GPEs depends on both the input graph data, and the target FPGA device. Each GPE requires three bitmaps: CQ_bitmap, NQ_bitmap, and $Status\_bitmaps$. These bitmaps are implemented using BRAM resources available in the FPGA device. Each bitmap requires $|V|$ bits of BRAM storage, where $|V|$ is the number of vertices in the input graph $G(V, E)$. Table 5.2 shows the required amounts of both on-chip memory resources (BRAM) as well as off-chip memory resource (DRAM). In addition to memory resources, each GPE requires a certain amount of Slice registers and Slice LUTs. Table 5.3 shows the resource utilisation of our GPE implementation of the BFS kernel used in the APSP-BFS algorithm.

In other words, the number of GPEs is limited by the amount of available BRAM resources and/or Slice LUTs resources. To determine the maximum number of GPEs for a given device, and an input graph $G(V, E)$, we develop the following analytical model.

Let $BRAM_{BITMAP}$ be the the number of BRAMs required for each bitmap. So

$$BRAM_{BITMAP} = \lceil \frac{3 \times |V|}{BRAM_{SIZE}} \rceil \tag{5.1}$$

where $BRAM_{SIZE}$ is the size of BRAM in bits.

Hence, the maximum number of GPEs based on BRAM resources only, Num_GPE_{BRAM}, is given by

$$Num\_GPE_{BRAM} = \frac{Num_{BRAM}}{BRAM_{BITMAP}} \tag{5.2}$$

where Num_{BRAM} is the number of BRAM resources available on the target FPGA device.

Using equation 5.1 with equation 5.2, Num_GPE_{BRAM} can expressed as follows

$$Num\_GPE_{BRAM} = \frac{Num_{BRAM}}{\lceil \frac{3\times|V|}{BRAM_{SIZE}} \rceil} \tag{5.3}$$

Let Num_{LUT} be the number of Slice LUTs available for GPE implementation on the target FPGA device, and let $LUT\_GPE(N)$ be the required amount of Slice LUTs for the GPE implementation for a given graph size $|V| = N$. So the maximum number of GPEs based on Slice LUT resources only, Num_GPE_LUT, is given by

$$Num\_GPE\_LUT = \frac{Num_{LUT}}{LUT\_GPE(|V|)} \tag{5.4}$$

Since the maximum number of GPEs, Num_GPE, depends on both BRAM and Slice LUTs, it can be expressed by combining equation 5.3 and equation 5.4 as follows

$$Num\_GPE = \min\left(Num\_GPE_{BRAM}, Num\_GPE\_LUT\right) \tag{5.5}$$

Or

$$Num\_GPE = \min\left(\frac{Num_{BRAM}}{\lceil \frac{3\times|V|}{BRAM_{SIZE}} \rceil}, \frac{Num_{LUT}}{LUT\_GPE(|V|)}\right) \tag{5.6}$$

From equation 5.6, we can say that the maximum number of GPEs that can be instantiated in a design depends on three factors:

- The number vertices, $|V|$, in the input graph G(V,E).
- The number and storage capacity of BRAM resources available in the target FPGA device.
- The number of Slice LUTs available in the target FPGA device.

Table 5.4: BRAM and Slice LUT resources on three FPGA devices from three different Virtex-6 sub-families.

Device name	Slice LUTs	BRAM (18Kb)
XC6VLX760	474,240	1,440
XC6VSX475T	297,600	2,128
XC6VHX565T	354,240	1,824

Using equations 5.3 and 5.4 with Tables 5.2 and table 5.3, we plot the maximum number of GPEs that can instantiated on a Virtex-6 LX760 device for different input graph sizes (i.e. number of vertices) as illustrated in Figure 5.9. The LX760 device has 474240 Slice LUTs and 1440 18Kb-BRAM resources. We can observe that for small input graph sizes ($|V|$) the maximum number of GPEs is determined by the amount of Slice LUTs available in the target device. As the input graph size increases, the maximum number of GPEs is dominated by the available amount of BRAM resources in the target device.

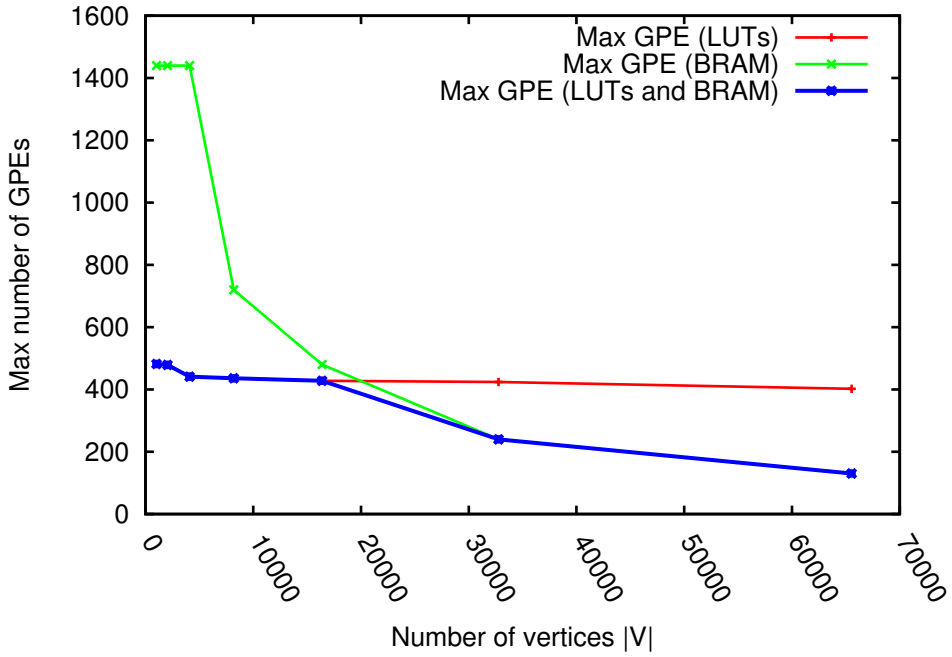


Figure 5.9: Maximum number of GPEs on a Virtex 6 LX760 device.

Based on the size of the input graph, different FPGA devices can be used to achieve optimum amounts of parallelism (i.e. maximum number of GPEs) for the APSP-BFS algorithm. This is illustrated in Figure 5.10, where equation 5.6 is used to estimate the maximum number of GPEs instantiable on three FPGA devices from three different Virtex-6 sub-families. Table 5.4 summaries the available Slice LUTs and BRAM resources on these three devices.

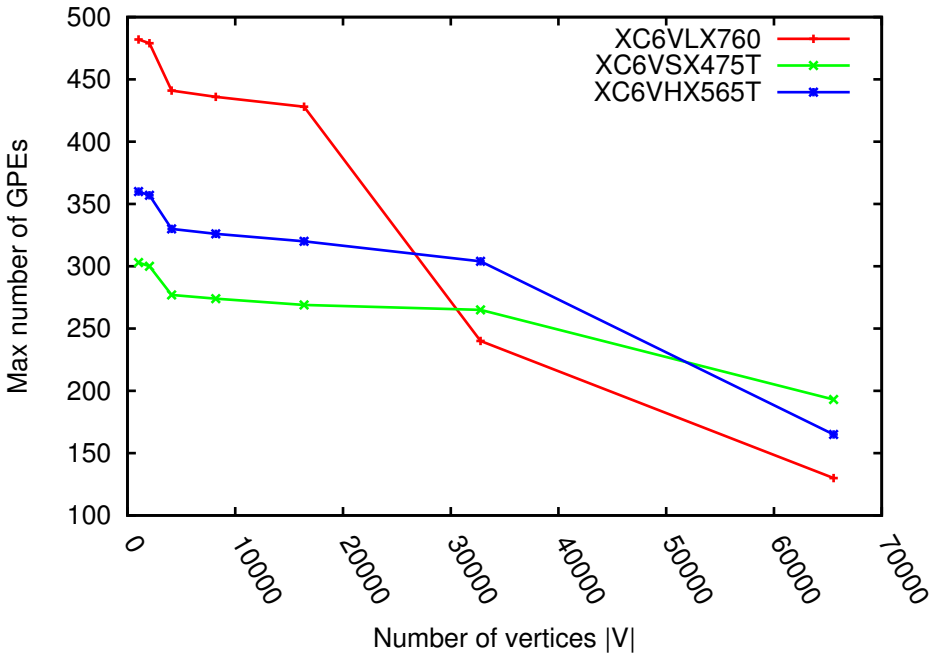


Figure 5.10: Maximum number of GPEs on three different Virtex-6 FPGA devices

5.7 Implementation and Measurements

This section provides details of our experiments. For the graph data, we use both synthetic R-MAT graphs and real-world human-brain network data. R-MAT graphs (Scale-free) graphs [50] are characterised by their skewed degree distribution and fractal community. These R-MAT graphs are generated using *GTgraph* [49], a synthetic graph generator suite. For the parameters of the R-MAT graph, we use the default values of the Graph500 benchmark (A=0.57, B=0.19, C=0.19).

In addition to synthetic graph data, we use real graph data extracted from a downloaded dataset from a fMRI data sharing project, *1000 Functional Connectomes Project* [64]. The dataset was downloaded from a fMRI data sharing project, namely 1000 Functional Connectomes Project [64]. The construction of the graph data from the fMRI data is described in detail in [2]. The first step in constructing the graph data is the calculation of Pearson's Correlation for 198 subjects, resulting in 198 correlation matrices. A single adjacency matrix with Boolean elements is then created by averaging and binarising these correlation matrices. Using different threashold values to binirise the correlation matrices results in adjacency matrices (or graphs) with different sparisity values (the average vertex degree). The resulting graphs have 38368 vertices with varying sparsity values. This is the same dataset that is used in [2].

For the high performance reconfigurable computing system, we use the Convey HC-1 server [52] which has four user-programmable Virtex-5 FPGAs, and each FPGA is connected to 16 GB of shared DRAM via a memory crossbar. Each memory controller is implemented on its own FPGA and is connected to two Convey-designed scatter-gather DIMM modules. Each AE has a 2.5 GB/s link to its corresponding memory controller, giving a theoretical aggregate peak memory bandwidth of 80 GB/s. However, the effective memory bandwidth of the AEs varies according to their memory access pattern.

To develop for Convey HC-1, we use the Convey Personality Development Kit (PDK) [51], which is a set of makefiles to support simulation and synthesis design flows. Convey provides a wrapper that allows the user to interface the FPGA design with both the host CPU and the memory controllers. Our FPGA-based APSP accelerator is expressed in RTL using Verilog HDL, and is compiled using Xilinx ISE v13.1. Our FPGA implementation has 32 GPEs (128 GPEs in total) that utilise about 69% of LUTs and 86% of BRAMs, with an operational clock frequency of 150MHz. We compare our performance results to CPU and GPU results reported in [2] and [3] respectively. Table 5.5 provides the configuration details of the machines used in our experiments and in the previous work [2] and [3].

Table 5.5: The specification of the machines used in our experiments and related work.

	Multi-core CPU [2]	**GPU** [3]	**Convey HC-1**
Core architecture	AMD Phenom II X4	AMD Cypress XT	Xilinx Virtex-5
Model No.	965	Radeon HD 5870	XC5VLX330
Fabrication process	45 nm	40 nm	65 nm
Launch year	2009	2010	2006
Core frequency	3.4 GHz	850 MHz	120 MHz
Total Num. of CPU cores	4	-	-
Total Num. of GPU stream cores	-	320	-
Total Num. of FPGA devices	-	-	4
Total Num. of threads/PEs	8	1600	256

5.8 Performance Evaluation

In this section, we validate the effectiveness of our reconfigurable computing methodology that we presented in Sections 5.3 and 5.4. To begin we examine the impact of design optimisations on the performance of our FPGA-based APSP-BFS design. We then measure the performance scalability of our APSP-BFS design as the number of GPEs is increased. Performance sensitivity to input graph scaling in terms of both vertex and edge counts is also examined. Finally, we conclude this section by comparing the performance of our FPGA-based APSP-BFS design to that of both multi-core CPU [2] and GPU [3].

5.8.1 Impact of design optimisations

Figure 5.11 and 5.12 show the effects of two design optimisations (see Section 5.5) over the baseline design for both synthetic R-MAT graphs and human brain network data . Using 16 bits to encode the adjacency lists improves the performance as the graph density (or average vertex degree) increases as the number of memory read operations to gather neighbouring vertices is almost halved in comparison with the baseline design that uses 32 bits to encode the adjacency list. For the hybrid algorithm, significant improvements are observed as the graph density increases, in particular when the average vertex degree is greater than 500 where a speedup of over 10x is achieved over the baseline design for R-MAT graphs. For low density graphs, the hybrid BFS design is slower than the baseline design due to the relatively large diameter of graphs, while the bottom-up approach algorithm (see Section 5.5.2 favours graphs with small diameter.

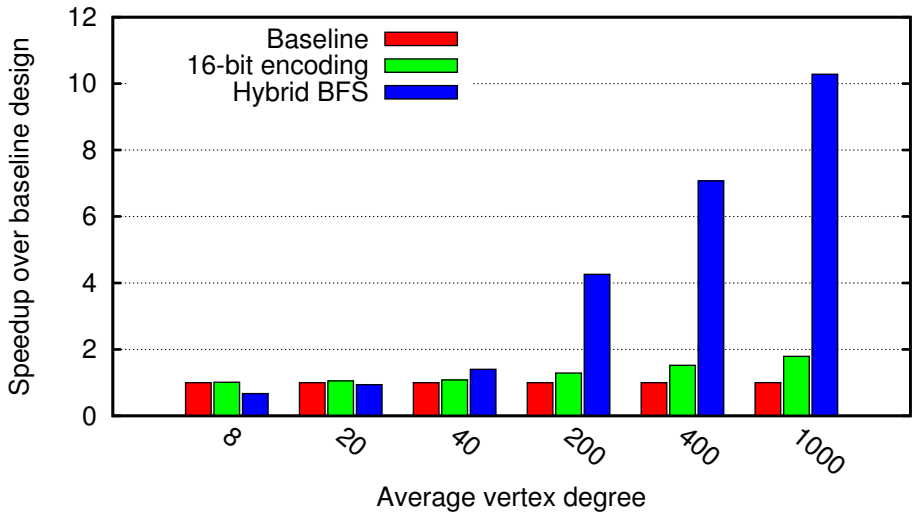


Figure 5.11: Optimisation effects on FPGA-accelerated APSP-BFS performance for R-MAT graphs with $|V| = 32768$ and variable edge count .

5.8.2 Performance scalability

Figure 5.13 and 5.14 show the scalability of our FPGA-based APSP-BFS design for R-MAT graphs and human brain network data respectively. The number of vertices for R-MAT graphs and human brain network data is

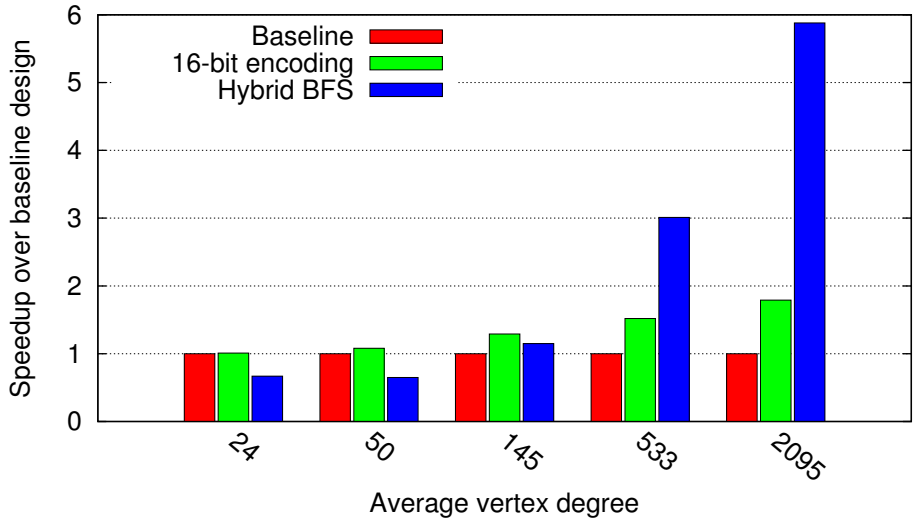


Figure 5.12: Optimisation effects on FPGA-accelerated APSP-BFS performance for human brain network data with $|V| = 38368$ and variable edge count .

set to 8192 and 38368 respectively, while the number of edges (or average vertex degree) varies. The number of GPEs varies from 1 to 128. We define the efficiency as the ratio of speedup of g GPEs over 1 GPE, divided by the linear or ideal speedup. In our current design we are able to fit up to 32 GPEs per Virtex5 LX330 device, so we use 2 and 4 FPGA devices for 64 GPEs and 128 GPEs respectively. For different average vertex degree values, we observe that our design not only scales well on one FPGA device giving over 96% of efficiency, but also over multiple FPGA devices as we are able to reach efficiency rates over 94% and over 92% for 2 and 4 FPGA devices respectively. This suggests that given a larger FPGA device, such as a Virtex-6 LX760, our design can achieve better performance results. This near-linear performance scalability is expected our FPGA design relies on private on-chip memories to parallelise the APSP-BFS algorithm, where each GPE has exclusive access to some on-chip memory resources.

5.8.3 Performance sensitivity to graph size scaling

Figure 5.15 shows the execution time of our FPGA-based APSP-BFS design using R-MAT graph instances with variable vertex and edge counts. The number of vertices is varied from 1024 to 32768 vertices, while the average vertex degree varies from 20 to 1000. We see that the execution time increases as the graph size

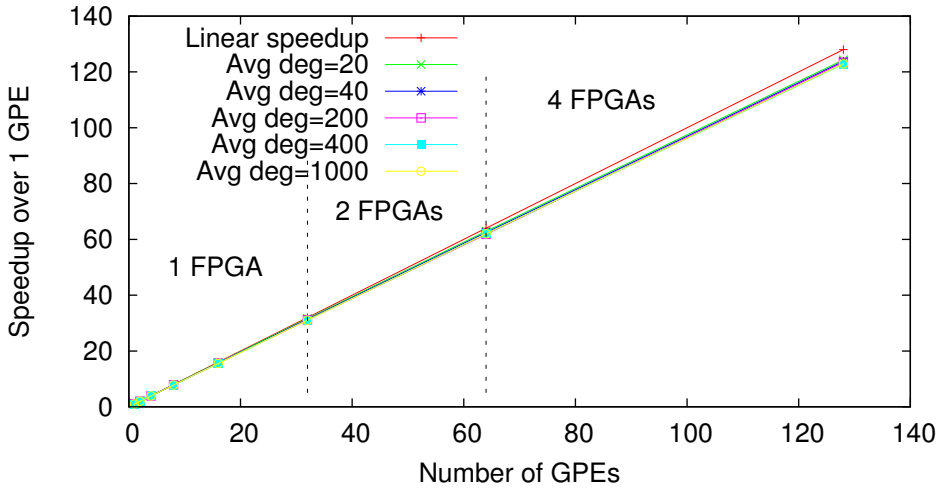


Figure 5.13: FPGA design performance: speedup over 1 GPEs for RMAT graphs.

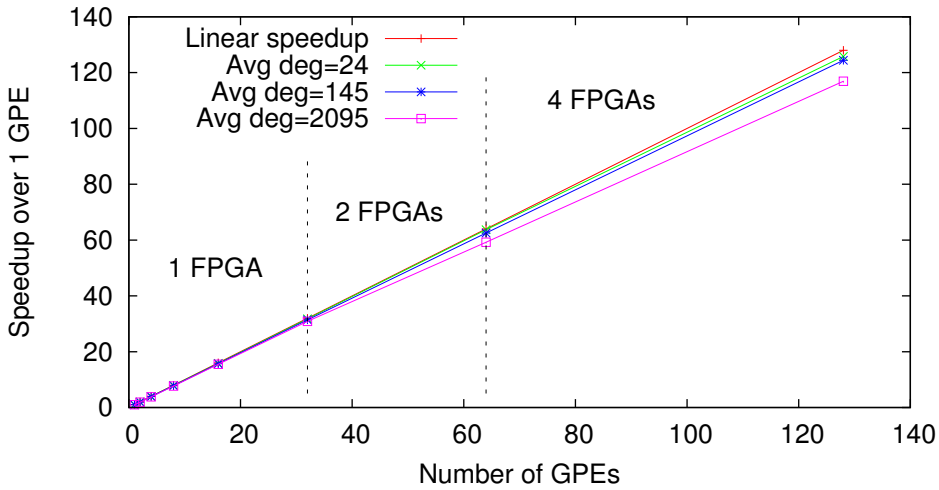


Figure 5.14: FPGA design performance: speedup over 1 GPEs for human bain network data.

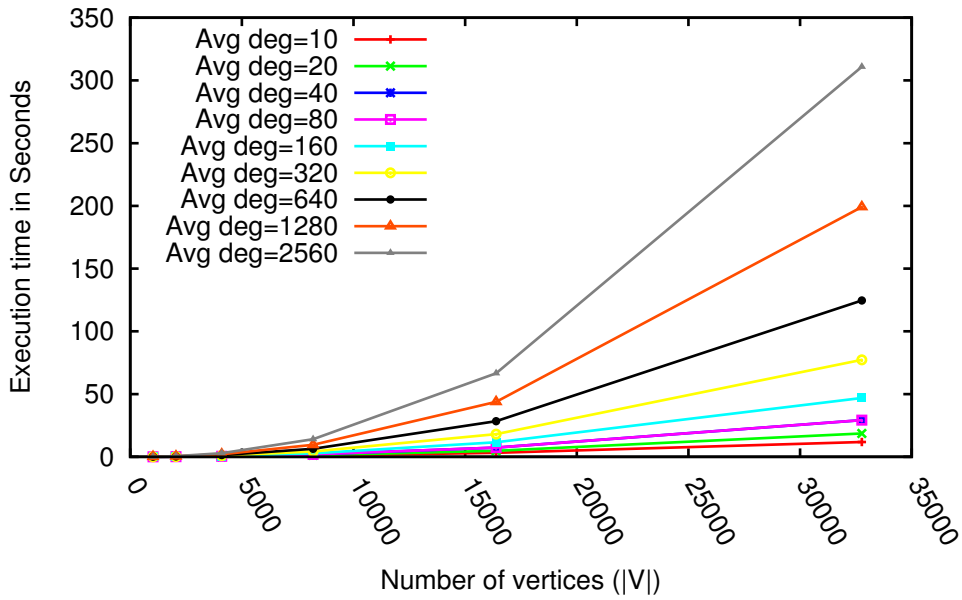


Figure 5.15: Performance sensitivity to graph size scaling for RMAT graphs.

increases in terms of vertex count and/or average vertex degree for R-MAT graphs. This increase of execution time is mostly sensitive to the number of vertex $|V|$. This can be explained by the fact that the APSP-BFS algorithm performs $\mathcal{O}(|V|+|E|)$ work for each vertex in the graph, leading to a total amount of work equal to $\mathcal{O}(|V|\times(|V|+|E|))$, or $\mathcal{O}(|V|^2+|V|\,.\,|E|)$. If d is the average vertex degree of the input graph $G(V,E)$, then $|E|=d.\,|V|$, and hence, the total amount of work for APSP-BFS can expressed as $mathcalO(|V|^2\times(1+d))$. In other words, the execution time of our FPGA-based APSP-BFS design is governed by this expression in which the execution time for given input graph is proportional to the square of the vertex count $|V|$.

5.8.4 Performance comparison to multi-core CPU and GPU

Table 5.6 compares the performance of our FPGA-based APSP design to that of a quad-core CPU [2], and a GPU [3]. Note that the GPU implementation executes the FW algorithm to solve the APSP problem, and hence the execution time stays the same when the average vertex degree of the graph varies. In contrast, the CPU and FPGA implementations run the APSP-BFS algorithm whose execution time depends on the both the graph's vertex count and average vertex degree. We present the result of two FPGA designs with different design optimisations: FPGA-1 which implements 16 bits encoding of adjacency list, and FPGA-2 which implements the hybrid BFS algorithm. Only one of the bitstreams of either FPGA-1 or FPGA-2 is loaded onto the FPGA during an APSP execution. As explained in Section 5.5.3, an FPGA configuration is selected based on the input

Table 5.6: Performance comparison with multi-core CPU [2] and GPU [3].

Average degree	AMD Quad-core CPU Phenom II X4 965 [2]		AMD Cypress GPU Radeon HD 5870 [3]		4xVirtex5 LX330			
					16-bit encoding		Hybrid-BFS	
	exec. time	speedup	exec. time	speedup	exec. time	speedup	exec. time	speedup
24	39s	1x	167s	0.2x	37s	1.1x	55s	0.7x
50	74s	1x	167s	0.4x	45s	1.6x	76s	0.9x
145	191s	1x	167s	1.1x	72s	2.7x	81s	2.5x
533	633s	1x	167s	3.8x	185s	3.4x	93s	6.7x
2095	2430s	1x	167s	14.5x	612s	3.9x	177s	13.7x

graph data at run-time.

The FPGA implementation of APSP outperforms both the quad-core CPU and the GPU for up to an average vertex degree of around 2000. As the graph grows in size, our FPGA implementation is 6-13 times faster than the CPU implementation, and 2-5 times faster than the GPU implementation for most average vertex degree values. Note that the GPU is faster than our FPGA design as the average vertex degree increases beyond 2000. This presents an opportunity to build a heterogeneous system comprising CPUs, GPUs, and FPGAs to accelerate the APSP problem for different average vertex degree values.

5.9 Summary

In this chapter, we presente a parallel and scalable FPGA design for the All-Pairs Shortest Paths problem for sparse graphs with unweighted edges. Our FPGA-based APSP solver is based on our reconfigurable computing approach that we presente in Chapter 3, augmented with an on-chip memory structure. Efficient utilisation of these on-chip memories allow us to greatly reduce off-chip memory accesses, and hence, achieve high performance results. Using a case study from bioinformatics, namely human brain connectomes, we have shown through experimental study that our FPGA design is able to outperform both multi-core CPUs and GPUs. Future work includes investigating ways to improve the performance by increasing the number of GPEs in the design by parallelising the BFS kernel as well as extending our design to support weighted edges.

Chapter 6

Comparing Performance of FPGAs and GPUs for High-Productivity Computing

6.1 Introduction

In previous chapters of this thesis (Chapter 3, 4, and 5), we present a reconfigurable computing approach to accelerate large-scale graph traversal algorithms. In our current approach, we use HDL-coded processing elements that are application-specific, which means that for each new algorithm, a new processing element needs to be hand-coded. While this led to great performance results, it required high design efforts as it involved hardware development using a hardware description language (i.e. Verilog HDL). In addition, programming FPGAs using a hardware description language requires software developers to learn a whole new set of skills and hardware design concepts.

One potential solution to the problem of programming FPGA-based accelerated systems is by providing a programming model similar to that of traditional software. In other words, problems are described using a more familiar high-level language, which allows software developers to quickly improve the performance of their applications using reconfigurable hardware. For example, in our reconfigurable hardware architecture for graph traversal algorithms (presented in Section 3.3.2), we can replace the HDL-coded graph processing elements with reconfigurable softcore processors such as the MicroBlaze [66] softcore processor. This will greatly reduce design efforts as softcores are programmed using a high-level language such as C or C++.

In this chapter, we evaluate the performance of FPGAs for high-productivity computing. For this purpose, we use a high-performance reconfigurable computing (HPRC) system where a commodity CPU instruction

set architecture is augmented with instructions which execute on specialised FPGA-based soft-cores, allowing the CPU and FPGA to co-operate closely while providing a programming model similar to that of traditional software. Our main contributions in this chapter are:

- A discussion of the landscape of high-productivity computing in the high-performance reconfigurable computing arena.
- Performance evaluation of benchmarks with different memory characteristics on two high-productivity platforms: an FPGA-based Hybrid-Core system, and a GPU-based system.
- A discussion of the advantages and disadvantages of using FPGAs and GPUs for high-productivity computing.

6.2 Related Work

Several researchers have explored ways to make FPGA programming easier by giving developers a more familiar C-style language instead of hardware description languages [67, 68]. However, these languages require a significant proportion of an existing application to be rewritten using a programming model specific to the accelerator, which is problematic in an environment where large amounts of legacy code must be maintained. There is previous work on comparing the performance of FPGAs and GPUs. Cope *et al.* [69] presented a systematic approach to the comparison of FPGAs and GPUs using throughput drivers. They assessed their approach using five case study algorithms with different arithmetic complexity, memory access requirements, and data dependence. Results of the study suggest that an FPGA fares better than a GPU for algorithms requiring large numbers of regular memory accesses, while the GPU outperforms the FPGA for algorithms with variable data reuse. In contrast to our work, this study does not take into account development efforts since the FPGA implementations are hand-crafted using VHDL.

In [70], it is reported that while FPGA and GPU implementations for real-time optical flow have similar performance, the FPGA implementation takes 12 times longer to develop. Another study shows that FPGA can be 15 times faster and 61 times more energy efficient than GPU for uniform random number generation [71]. A third study shows that for many-body simulation, a GPU implementation is 11 times faster than an FPGA implementation, but the FPGA is 15 times better than the GPU in terms of performance per Watt [72].

Che *et al.* [73] compared GPUs and FPGAs using three different applications -*Gaussian Elimination, Data Encryption Standard (DES), and Needleman-Wunsch*. Based on their comparative study, the authors propose a

best-fit mapping of applications to either platforms - GPU or FPGA. They consider GPUs to be a clear winner for applications with parallel computations and no inter-dependence in the data flow (Gaussian Elimination). For computations that involve significant detailed low-level hardware control operations which cannot be efficiently implemented in a high-level language, they identified FPGAs as a better fit than GPUs. They also report FPGAs fare better than GPUs for applications with a certain degree of complexity (e.g. Needlman-Wunsch application) and which can take advantage of data streaming and pipelining. The authors attempt to compare productivity using *Lines-of-Code* (LOC) as a metric to estimate programming effort. A serious limitation of this approach is that the cost of writing one line of source code in VHDL and CUDA C are likely to be significantly different. CUDA C is a high-level language and requires usually much less development time than VHDL to implement a parallel application.

Prior work on comparing FPGAs and GPUs for high-productivity computing used a set of non-standard benchmarks that target different process architectures [74] such as asynchronous pipeline and partially synchronous tree. Using the benchmarks in [74] results in analysis that considers processes at architectural level. However, these benchmarks do not cover applications with different memory access characteristics, and they do not address the performance of GPUs for non-streaming applications in comparison to a HPRC system.

Similarly, Fletcher *et al.* [75] used a many-core architectural template and a high-level imperative programming model to compare FPGAs and GPUs for high-productivity computing. In [75], the authors restricted their work to a single benchmark, namely the Bayesian computing problem, which represent a benchmark that has low requirements in both floating-point computations and off-chip memory bandwidth, two strong assets of modern GPGPUs. Even though the results obtained in this work showed that FPGAs outperformed GPUs for the Bayesian computing problem while reducing development efforts compared to HDL-crafted FPGA designs. It is not clear how this approach will perform for other problems, in particular applications with high requirements in terms of floating-point operations and/or off-chip memory bandwidth.

6.3 The Landscape of High-Productivity Computing in HPRC

Several researchers sought ways to make FPGA programming easier by proposing alternative methodologies to HDL-based design techniques. An assortment of different methodologies to achieve high-productivity computing using FPGAs with varying capabilities and application targets have been developed in both academia and industry. We distinguish between three different methodologies based on the programming model, and platform support and integration: (i) platform-independent HLS tools, (ii) platform-specific HLS tools, and

(iii) reconfigurable many-soft-core computing.

6.3.1 Platform-independent high-level synthesis tools

These tools aim to reduce design efforts compared to HDL-based design tools by using high-level languages to obtain design abstraction with better productivity. Most of these tools are based on a subset of synthesisable C/C++ or make use of a C-like language in an attempt to make such tools more accessible to software developers. Examples of major commercial C-based HLS tools include Xilinx's AutoESLAutoPilot[76], Cadence's C-to-Silicon Compiler [77], and Mentor's Catapult C [78].

Some of the advantages of platform-independent HLS tools are the reduced amount of design efforts as well as portability. However, there are many drawbacks that are holding back such tools from widespread adoption by HPC software developers. First of all, even though high-level languages are used by these HLS tools, the programming model requires some understanding of hardware design concept to achieve good performance levels. More importantly, while such tools may do a great job at synthesising datapaths, they often neglect important aspects such as interfaces to other hardware modules, and platform integration. Lack of interfacing support may become the main problem to achieve high-productivity, as it exposes the user of tool to low-level details such as memory bus interfaces and memory controllers. Also, poor platform integration will make in-system verification of hardware designs more time-consuming and hence less productive.

6.3.2 Platform-specific high-level synthesis tools

Platform-specific HLS tools offer similar features to those of platform-independent HLS tools. However, a major difference is the support of a specific platform offering better platform integration in terms of interfacing infrastructure. This means that a great deal of low-level details about interfacing with external components, such as off-chip DDR memory modules, are made transparent to the user, and hence improving overall developer productivity. In addition to platform integration, platform-specific HLS tools provide better simulation and debugging tools, in particular, for in-system verification of hardware designs leading to better time-to-solution compared to platform-dependent HLS tools.

Maxeler's MaxCompiler [79] is a commercial HLS tool that is platform-specific. MaxCompiler provides the user with compilers, simulators, and libraries to create an FPGA-based hardware accelerator that targets Maxeler platforms. Implementations created by MaxCompiler are based on streaming model of computation, which

may restrict the applicability of this HLS to application that can fit onto the streaming model, and streaming kernels, in general, provide only limited forms of control flow and no random access to global memory since such features break the regularity of data streams. In addition, the programming model may require software developers to step out of their comfort zone to achieve significant performance improvement. Finally, while being platform-specific, MaxCompiler can provide great platform integration but at the expense of portability, since code written for MaxCompiler cannot be ported to platforms other than Maxeler's machines.

6.3.3 Reconfigurable many-soft-core approach

This approach is based on implementing soft-cores on FPGAs to accelerate compute intensive portions of an applications. The overall architecture of a reconfigurable many-soft-core system is shown in Figure 6.1. It consists of three key components: a set of identical soft-cores, a memory interconnect network, and a management unit. Unlike embedded processor cores found commonly found in modern FPGAs, the soft cores can be either implemented as general-purpose RISC processor cores, or implemented as application-specific data-paths. For example, the width of datapaths can changed to reduce resource utilisation, and the depth (i.e. number of pipeline stages) of datapaths can be increased to improve the maximum operating clock frequency of these datapaths. Another customisation of the soft-cores can be the incorporation/omission of certain hard-wired operations such as Multiplier and Multiplier-&-Accumulate operations. Precision of arithmetic operations can also be adjusted to improve area efficiency while meeting application-specific requirements (e.g 32-bit floating-point operations versus 64-bit floating-point operations).

The memory interconnect network is used to connect the soft-cores to both-on-chip and off-chip shared memory. Through customisation, the interconnect network can be optimised for different memory access patterns. For example, in an interleaved memory system, different interleaving schemes can be selected for different memory access patterns. Finally, the management unit schedules tasks for execution on the custom cores. In tightly-coupled host-coprocessor system where the reconfigurable many-core system is used as an FPGA-based coprocessor, the management unit can be used to interface the soft-cores with the host processor.

A typical development model of a many-soft-core machine is shown in Figure 6.2. The applications are coded in high-level language such as C/C++, or FORTRAN. The compilation of the source code is partitioned onto two separate tracks: 1) the compilation of the portion of source code that runs on general-purpose processor (the host processor), and 2) the compilation of the source code that executes on a the FPGA-based many-soft-core machine (the coprocessor). The latter compilation process may involve hardware synthesis of custom datapaths

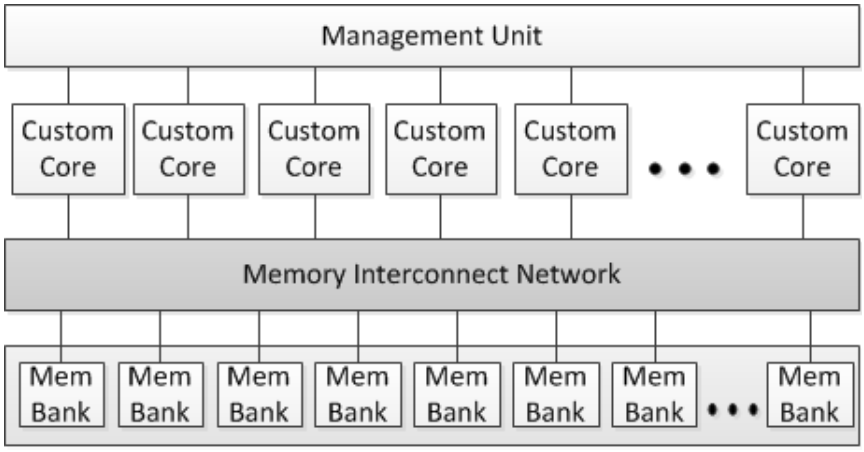


Figure 6.1: Reconfigurable many-soft-core architecture.

or soft-cores that are specific to the target application.

6.4 Characterising Productivity and Locality

Creating a meaningful productivity measure which can be applied to programming models and architectures would require the aggregation of results from many individual projects, and is outside the scope of this paper. Instead we restrict our productivity study to the idea that if the same programmer effort is applied to two systems, then the system providing the highest performance solution is the most productive. This stems from the following equation that was developed in [80]:

$$Productivity = \frac{Relative\ Speedup}{Relative\ Effort} \tag{6.1}$$

and hence productivity will be dictated by the relative speedup achieved by a platform:

$$\frac{Productivity(GPU)}{Productivity(FPGA)} = \frac{Relative\ Speedup(GPU)}{Relative\ Speedup(FPGA)} \tag{6.2}$$

So in our work, we restrict our productivity comparison to using a common programming model based on familiar high-level programming languages and development tools. The benchmarks are developed using the C language plus platform specific extensions, using comparable development efforts in terms of design time and programmer training.

In contrast to previous work [74], we select a number of established benchmarks based on the HPC Challenge (HPCC) Benchmark Suite [81] and the Berkeley technical report on Parallel Computing Research [82] to

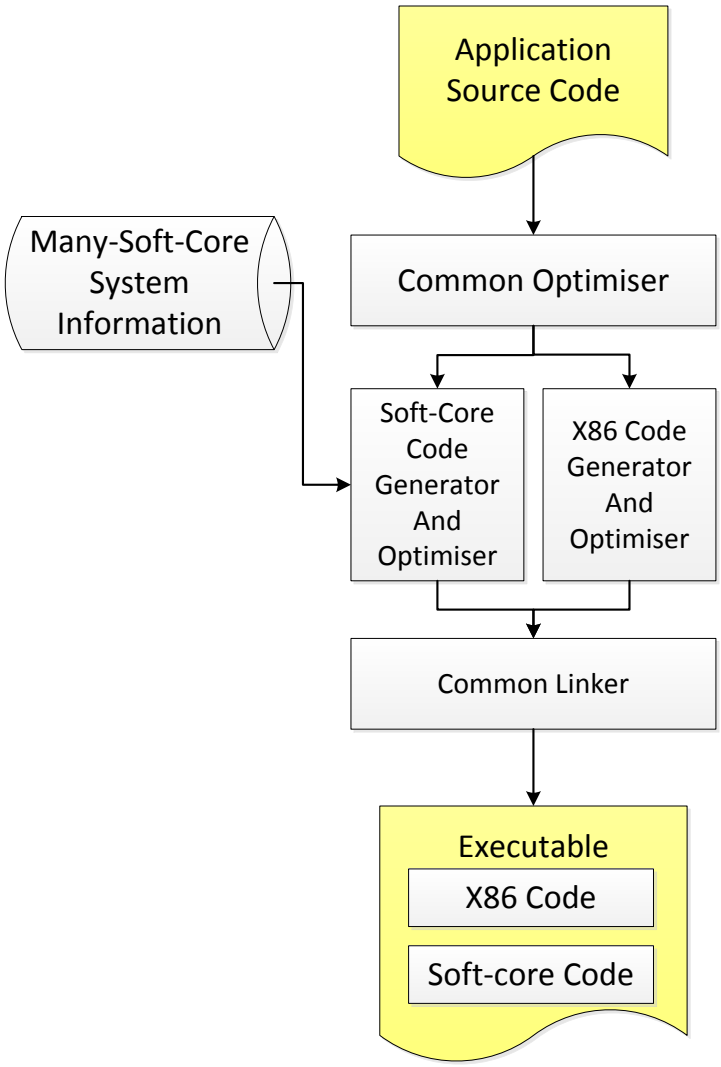


Figure 6.2: Development Model of a Reconfigurable Many-Soft-Core Machine

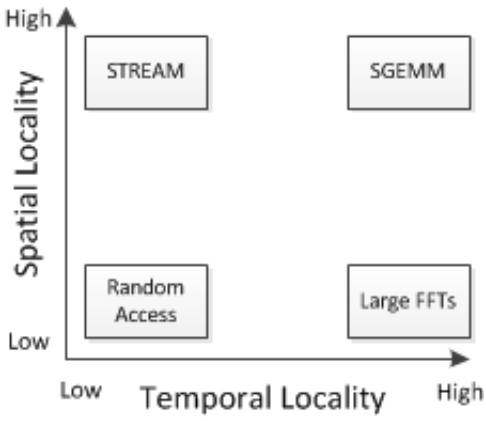


Figure 6.3: Benchmark applications as a function of memory access characteristics

Table 6.1: Limits to performance of benchmarks

Benchmark Program	Main performance limiting factor
STREAM	Memory Bandwidth limited
Dense matrix multiplication	Computationally limited
Fast Fourier Transform	Memory latency limited
RandomAccess	Memory latency limited
Monte-Carlo Methods	Parallelism limited

measure the relative speedup for each platform. Five benchmark programs are used:

- The STREAM benchmark.
- Dense matrix multiplication.
- Fast Fourier Transform (FFT).
- RandomAccess benchmark.
- Monte-Carlo methods for pricing Asian options.

The authors in [82] identify the main performance limiting factor for these benchmarks, shown in Table 6.1. Some of these benchmarks are part of the HPCC benchmarks, which aim to explore spatial and temporal locality by looking at streaming and random memory access type benchmarks, as illustrated in Figure 6.3. Where possible these benchmark programs are implemented using vendor libraries, which are optimised for each platform.

6.4.1 STREAM

The STREAM benchmark [81] is a simple synthetic benchmark that is used to measure sustained memory bandwidth to main memory using the following four long vector operations:

$$\begin{aligned} \texttt{COPY}: \; & c \leftarrow a \\ \texttt{SCALE}: \; & b \leftarrow \alpha c \\ \texttt{ADD}: \; & c \leftarrow a + b \\ \texttt{TRIAD}: \; & a \leftarrow b + \alpha c \end{aligned}$$

where $a, b, c \in \mathbf{R}^m; \alpha \in \mathbf{R}$

The STREAM benchmark is designed in such a way that data re-use cannot be achieved. A general rule of this benchmark is that each vector must have about 4 times the size of all the last-level caches used in the run. In our work, we used arrays of 32 Million floating-point elements (4 bytes for each element), which require over 300MB of memory. Each vector kernel is timed separately and the memory bandwidth is estimated by dividing the total number of bytes read and written, by the time it takes to complete the corresponding operation.

6.4.2 Dense matrix multiplication

Dense floating-point matrix-matrix multiplication is a vital kernel in many scientific applications. It is one of the most important kernels in the LINPACK benchmark, as it is the building block for many higher-level linear algebra kernels. The importance of this benchmark has led HPC system vendors to both optimize their hardware and provide optimised libraries for this benchmark. In our work, we use vendor-provided libraries for the matrix multiplication benchmark, in order to achieve optimum or near optimal performance results for each platform.

The SGEMM routine in the BLAS library performs single precision matrix-matrix multiplication, defined as follows:

$$c \leftarrow \beta C + \alpha AB$$

where $A, B, C \in \mathbf{R}^{n \times n}; \alpha, \beta \in \mathbf{R}^n$

6.4.3 Fast Fourier transform

The Discrete Fourier Transform (DFT) plays an important role for a variety of areas, such as biomedical engineering, mechanical analysis, analysis of stock market data, geophysical analysis, and radar communications [83]. The Fast Fourier Transform [84] is an efficient algorithm for calculating the DFT and its inverse. In particular, the FFT requires $O(N)$ memory accesses versus only $O(N \log N)$ floating-point operations, requiring not only high computation throughput but also high memory bandwidth [85]. In addition, unlike SGEMM, FFT requires non-unit stride memory access, and hence exhibits low spacial locality [86].

6.4.4 RandomAccess

The RandomAccess benchmark measures the performance of a system for random memory accesses. The benchmark calculates the GUPS (Giga UPdates per Second) by identifying the number of memory locations that can be randomly updated in one second, divided by 1 billion. The term ”randomly” means that there is little relationship between one address to be updated and the next, except that they occur in the space of 1/2 the total system memory. An update is a read-modify-write operation on a table of 64-bit words. An address is generated, the value at that address read from memory, modified by an integer operation (add, and, or, xor) with a literal value, and the new value is written back to memory. Listing 6.1 shows the pseudo-code of the scalar implementation of RandomAccess using the 64-bit Galois Linear Feedback Shift Register (LFSR) random number generator [87]. In the case of parallel implementation of the RandomAccess benchmark, a small (less than 1%) percentage of missed updates is permitted.

Listing 6.1: Scalar implementation of RandomAccess using Galois LFSR

```
ran = 1;
for (i=0; i < 4*TableSize; ++i) {
   ran = (ran<<1) ^ (((int64_t)ran <0)? 7:0);
   table[ran & (TableSize-1)]^= ran;
}
```

6.4.5 Monte-Carlo methods for Asian options

Monte Carlo methods [88] are a class of algorithms that use psuedo-random numbers to perform simulations, allowing the approximate solution of problems which have no tractable closed-form solution. In this work, we

examine financial Monte-Carlo simulations for pricing Asian options. Asian options are a form of derivative which provides a payoff related to the arithmetic average of the price of an underlying asset during the option life-time:

$$P_{call} = max(\frac{1}{n+1}\sum_{i=0}^{n} S(t_i) - K, 0) \tag{6.3}$$

where P_{call} is the payoff of the Asian call option, $S(t_i)$ is the asset price at time t_i, and K is the strike price.

Unlike European options, which can be priced using the Black-Scholes equation [89], there is no closed-form solution for pricing Asian options. Instead, Monte-Carlo methods are used to calculate the payoff of Asian options. Using Monte-Carlo methods leads to accurate results at the expense of a large number of simulations, due to the slow convergence of this method [90].

We choose Monte Carlo methods as one of the benchmarks for two reasons. First, they are widely used in many applications including finance, physics, and biology. Second, they support highly parallel execution [82] with low memory bandwidth requirements, which would map well onto FPGAs and GPUs.

6.5 High-Performance Reconfigurable Computing System

6.5.1 HC-1 system architecture

Convey's Hybrid-Core technology is an example of a commercially-available HPRC system that achieves a compromise between productivity and performance. In the Hybrid-Core approach, the commodity instruction set, Intel's x86-64, is augmented with application-specific instruction sets (referred to as personalities), which are implemented on an FPGA-based coprocessor to accelerate HPC applications. A key feature of this architecture is the support of multiple instruction sets in a single address space [91]. The Convey Hybrid-Core server, HC-1, has access to two pools of physical memory: the host memory pool with up to 128GB of physical memory located on the x86 motherboard, and the coprocessor memory pool with up to 128GB of memory, located on the coprocessor board. An overview of the architecture of the HC-1 is shown in Figure 6.4.

The coprocessor contains three main components: the Application Engine Hub (AEH), the Memory Controllers (MCs), and the Application Engines (AEs). The AEH represents the central hub for the coprocessor. It implements the interface to the host processor via a Front-Side Bus (FSB) port, an instruction fetch/decode unit, and a scalar processor to execute scalar instructions. All extended instructions are passed to the application engines. The coprocessor has eight memory controllers that support a total of 16 DDR2 memory channels offering an

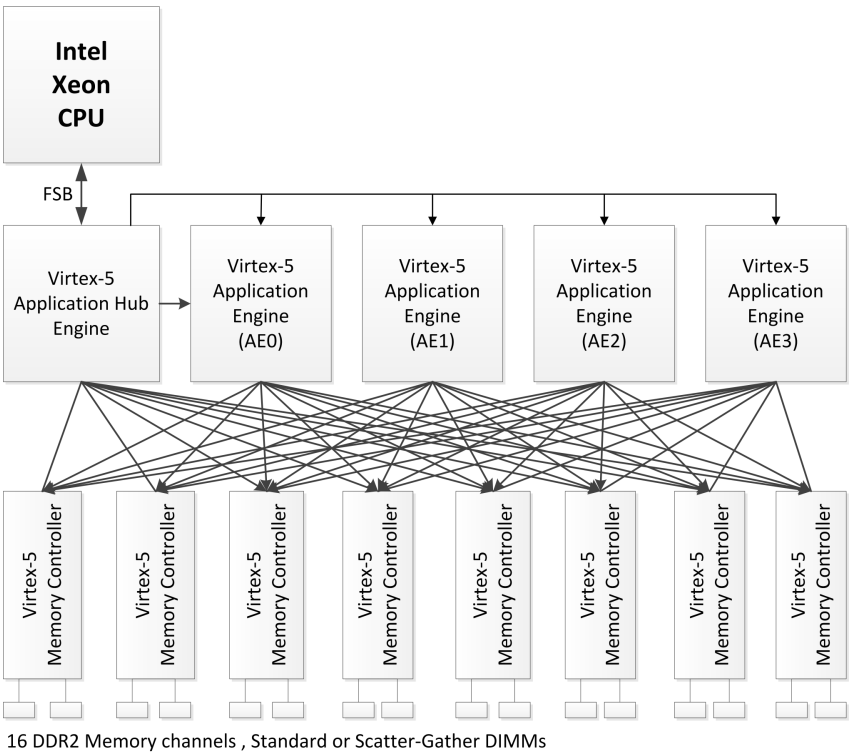


Figure 6.4: Convey HC-1 architecture.

aggregate of over 80GB/s of bandwidth to ECC protected memory. Finally, the AEs implement the extended instructions used to accelerate applications. Each Application Engine is connected to all the eight memory controllers via a network of point-to-point links to provide high sustained bandwidth for different memory access patterns.

6.5.2 HC-1 development model

The HC-1 programming model is shown in Figure 6.5. A number of key features distinguish the Convey programming model from other coprocessor programming models [92]. Applications are coded in standard C, C++, or Fortran. Performance can then be improved by adding directives and pragmas to guide the compiler. In an existing application, a routine or section of code may be compiled to run on the host processor, the coprocessor, or both. A unified compiler is used to generate both x86 and coprocessor instructions, which are integrated into a single executable and can execute on both standard x86 nodes and accelerated Hybrid-Core nodes. The coprocessor's extended instruction set is defined by the personality specified when compiling the application.

Porting an application to take advantage of the Convey coprocessor capabilities can be achieved using one or more of the following approaches:

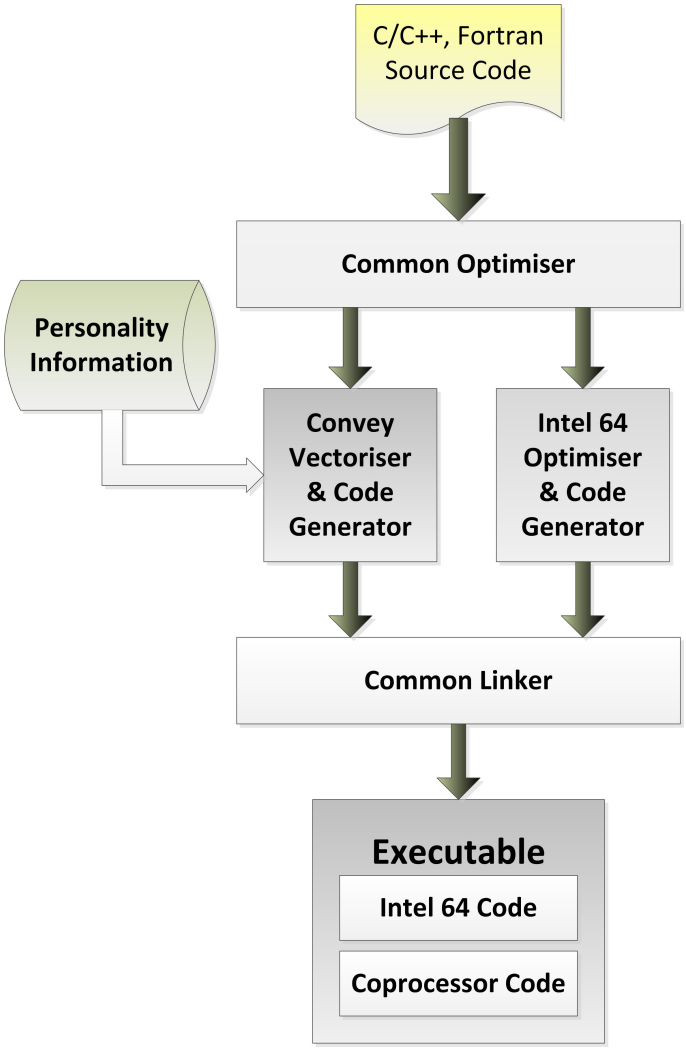


Figure 6.5: Programming model of the Convey Hybrid-Core

- Use the Convey Mathematical Libraries (CML) [93], which provide a set of functions optimised for the accelerator.

- Compile one or more routines with Convey's compiler. This can be performed by using the Convey auto-vectorization tool to automatically select and vectorise DO/FOR loops for execution on the coprocessor. Directives and pragmas can also be manually inserted in the source code, to explicitly indicate which part of a routine should execute on the coprocessor.

- Hand-code routines in assembly language, using both standard instructions and personality specific accelerator instructions. These routines can then be called from C, C++, or Fortran code.

- Develop a custom personality using Convey's Personality Development Kit (PDK).

6.5.3 Convey personalities

A personality groups together a set of instructions designed for a specific application or class of applications, such as finance analytics, bio-informatics, or signal processing. Personalities are stored as pre-compiled FPGA bit files, and the server dynamically reconfigures between bit-files at run-time to provide applications with the required personality. Convey provides a number of personalities optimized for certain types of algorithms such as floating-point vector personalities, and a financial analytics personality. In our work, we use three Convey personalities: the single-precision vector personality, the double-precision vector personality, and the financial analytics personality.

6.6 Productive GPU-based Computing

6.6.1 GPU architecture

A CUDA GPU [24] has a massively-parallel many-core architecture that supports numerous fine-grained threads, and has fast shared on-chip memories that allow data exchange between threads. In this work, we use nVidia's Tesla C1060 GPU which has 240 streaming processors running at 1.3GHz. It has a theoretical peak single-precision floating-point performance of 933 GFlops. It also has access to 1GB of GDDR3 memory at 800MHz, offering up to 102GB/sec of memory bandwidth [94].

6.6.2 CUDA development model

The NVIDA CUDA SDK environment provides software developers with a set of high-level languages such as C and Fortran. These programming languages are extended to support the key abstractions of CUDA such as hierarchical thread groups, shared memories, and barrier synchronization. CUDA C extends C by allowing the programmer to implement kernels as C functions that are executed in parallel by CUDA threads. These threads can be grouped in thread blocks where they share the processing and memory resources of the same processor core. The blocks are organized into a one, two, or three dimensional grid of thread blocks, with the number of thread blocks chosen according to the size of the processed data and the processing resources. The CUDA programming model assumes that the CUDA threads execute on a physically separate device, operating as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

6.7 Development and Optimisation of the benchmarks

6.7.1 Convey HC-1

The implementation of the STREAM benchmark on the HC-1 is straight forward, as it only consists of vector operations. For the SGEMM and FFT benchmarks, we use the Convey Mathematical Libraries (CML) [93]. The CML library provides a collection of linear algebra and signal processing routines such as BLAS and FFT. In addition, Convey provides a single-precision *floating-point vector personality* which implements the CML libraries, as well as vector operations for both integer and floating-point data types. This personality is based on a load-store architecture with modern latency-hiding features. Figure 6.6 shows a diagram of one of the 32 function pipes that are available in the single precision personality. The double precision version has only two Fused Multiply-Add (FMA) units and one adder unit. We can calculate the peak floating-point performance of the HC-1 for the single and double precision vector personalities as follows:

$$FLOPS_{(peak)} = Number\,of\,FP\,units\ \times Clock\,Frequency$$

The single precision personality has 32×4 fused multiply-add units, which means it can perform 32x4x2 floating-point operations per clock cycle. The single precision vector personality is clocked at 300 MHz, and so the floating-point peak performance of the HC-1, when the vector personality is loaded onto the coprocessor, is

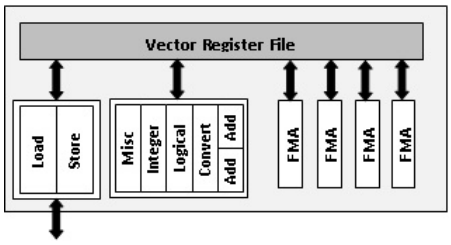


Figure 6.6: A soft-core from Convey's Single-precision vector personality.

about 76.8 GFlops single precision.

The Asian option pricing benchmark contains two parts: pseudo-random number generation, and Monte-Carlo based simulation paths. The financial analytics personality provides custom hardware for generating random numbers using the Mersenne-Twister algorithm. The second part of the benchmark contains nested loops that can be vectorised for efficient execution on the HC-1 coprocessor.

The RandomAccess benchmark is implemented using Convey's Personality Development Kit (PDK), as the current Convey personalities (soft-cores) do not support random memory access patterns at the time of writing this thesis. The Convey PDK is a set of makefiles to support simulation and synthesis design flows. Convey provides a wrapper that allows the developer to interface the FPGA design with both the host CPU and the off-chip memory. The hardware design is expressed in RTL using Verilog HDL and was compiled using ISE v13.1, and runs at 150 MHz. Although this implementation appears like a digression from the programming model described in section 6.5.2, it can be easily implemented in a reconfigurable many-soft-core system such as the MARC-II system [95]. In addition, we avoid any heavy optimisations, and all the design optimisations present in our implementation use existing static analysis techniques that can be captured by a high-level synthesis tool.

6.7.2 NVIDIA Tesla C1060

The STREAM benchmark is implemented as 4 CUDA kernels, with each kernel being executed by a large number of threads to achieve optimum performance. The number of threads is set so that the first vector kernel yields the same performance as the CUBLAS:sscopy routine. For dense matrix multiplication we use the CUBLAS library [96], provided by nVidia for linear algebra kernels. In our application, the host code creates the input matrices in the GPU memory space, fill them with data, call the the CUBLAS:SGEMM routine, and finally upload the results from the GPU memory back to the host memory. Given the low double-precision floating-point peak performance (78 GFLOPS) of the Tesla C1060 compared to its single precision peak performance (933 GFLOPS), we only consider the single precision version (SGEMM).For double-precision comparison, it

would be more appropriate to use the new Tesla C20x series, which has a double-precision floating-point peak performance of 515 GFlops, but these cards are still unavailable at the time of writing.

The FFT benchmark is implemented using CUFFT [97], the NVIDIA CUDA Fast Fourier Transform (FFT) library. The CUFFT library provided a simple interface for computing parallel FFTs on an NVIDIA GPU, which allows us to leverage the floating-point power and parallelism of the Tesla C1060 GPU without having to develop a custom, GPU-based FFT implementation.

The RandomAccess benchmark is implemented as a single CUDA kernel. This kernel reads the values of a table stored in global memory, and then updates these values. We use 512 threads to execute this CUDA kernel.

The Asian option pricing application is implemented using two kernels: a pseudo-random number generator (PRNG), and a path simulator. The PRNG is implemented using nVidia's Mersenne Twister RNG. For the path simulator kernel, each thread simulates a price movement path and then the results from each thread are summed in a hierarchical fashion starting with threads from the same block using shared memory. The partial results are then stored in the global memory, and final aggregation takes place for the valuation of the option price.

6.8 Performance Comparison

In the following, we present performance results of the GPU and FPGA implementations of the aforementioned benchmarks. For our performance evaluation, we use an Intel quad-core Xeon E5420 clocked at 2.5 GHz with 12 MB of L2 cache and 16 GB of DDR2-DRAM. For the GPU, we use the Tesla C1060 which has 240 streaming processors running at 1.3GHz with 1 GB GDDR3 of global memory. Finally, for the FPGA-based system, we use Convey Hybrid-Core (HC-1) platform described earlier in section 6.5.

6.8.1 STREAM

Tables 6.2 and 6.3 show the results for the STREAM benchmark on the HC-1 server and the Tesla C1060 GPU respectively. The size of the vectors used in our work is 32 million elements. From these results, we can see that the sequential memory bandwidth sustained by the GPU is about two times larger than that of the HC-1. These figures, when combined with the peak floating-point performance of GPUs and the HC-1, suggest that the GPU is likely to outperform the HC-1 for streaming programs with intensive computation and bandwidth requirements.

Table 6.2: STREAM benchmark results for HC-1

Function	Average Rate (GB/s)	Min Rate (GB/s)	Max Rate (GB/s)
COPY	35.82	35.77	35.88
SCALE	35.10	35.06	35.16
ADD	42.09	42.06	42.12
TRIAD	44.75	44.61	44.86

Table 6.3: STREAM benchmark results for Telsa C1060

Function	Average Rate (GB/s)	Min Rate (GB/s)	Max Rate (GB/s)
COPY	73.84	73.24	74.34
SCALE	74.00	73.52	74.34
ADD	71.14	70.87	71.46
TRIAD	71.23	71.05	71.43

6.8.2 Dense matrix multiplication

In our work, we use the Intel MKL [98] for software matrix multiplication. The matrix-matrix multiplication benchmark results are shown in Figure 6.7. The GPU is a clear winner with a floating-point performance of about 370 GFLOPS for large square matrices. This is expected, since the performance of the matrix multiplication benchmark is computation bound, and the Tesla C1060 has over 10 times more floating-point peak performance than the HC-1. In addition, the HC-1 implementation offers no significant speed-up over a multi-threaded MKL implementation running on an Intel Xeon E5420 Quad-Core CPU. The GPU is about 5 times faster than both the CPU and the Convey Coprocessor. This speed-up decreases to about 2.5 to 4.2 times if we include data transfer from the main memory to the GPU memory, while the HC-1 coprocessor can be slower than the CPU when data transfers from the host processor memory to the coprocessor memory are taken into account.

6.8.3 Fast Fourier transform

In this work, we use the popular FFTW [99] for the CPU implementation, as FFTW supports multi-threading and is reported to be more efficient than its Intel MKL counterpart. Figure 6.8 shows the performance of a one-dimensional in-place single-precision complex-to-complex FFT on the three platforms. The HC-1 outperforms the GPU (CUFFT) by up to a factor of three for large FFT sizes, and is about 16 times faster than the single-threaded FFTW, or about 4 times faster than the multi-threaded version.

The HC-1 implementation of the FFT achieves over 65 GFlops, thanks to its coprocessor memory subsystem that is optimized for non-unit stride and random memory accesses [91]. The Tesla C1060 uses GDDR

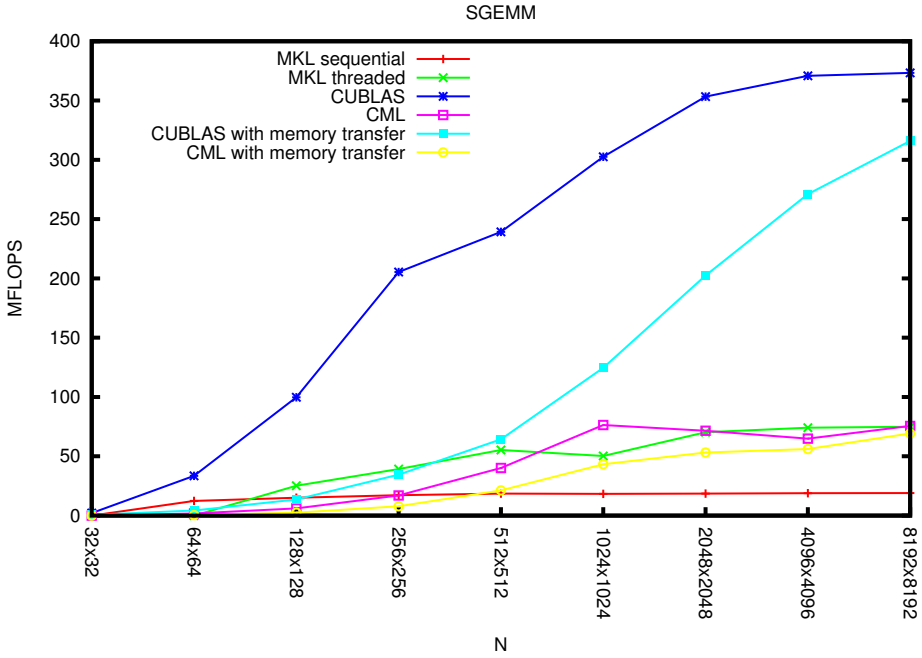


Figure 6.7: SGEMM - floating-point performance.

memories which are optimised for sequential memory access operations and stream programming for graphics applications. This leads to a performance penalty if GPU applications, such as FFT, involve non-sequential memory accesses [85].

To further investigate the effect of strided memory access on the performance of the HC-1 and Tesla C1060 GPU, we conduct the following experiment. We measure the effective bandwidth that can be achieved with strided memory access, and how it compares to a baseline sequential memory access bandwidth. We used the BLAS routine *blas:sscopy* available to each platform. This routine copies a real vector into another real vector. The increment between two consecutive elements in each vector can be specified, i.e. the stride parameter. The results of this experiment for vectors of 32 million elements are shown in Figure 6.9. As the stride parameter gets larger, the data transfer rate of the GPU becomes slower than that of HC-1.

However, the floating-point performance of the HC-1 drops significantly if data transfer times between host memory and coprocessor memory are taken into account. Fortunately, there are various applications that execute many FFT operations between memory transfers from CPU memory to coprocessor memory. An example of such an application is the protein-protein docking simulator ZDock [100], which uses a scoring scheme to determine the best docking positions. At the heart of ZDock is an FFT used for computing convolutions, and once the host processor has sent the initial input data, the coprocessor can perform multiple FFTs without significant data transfer.

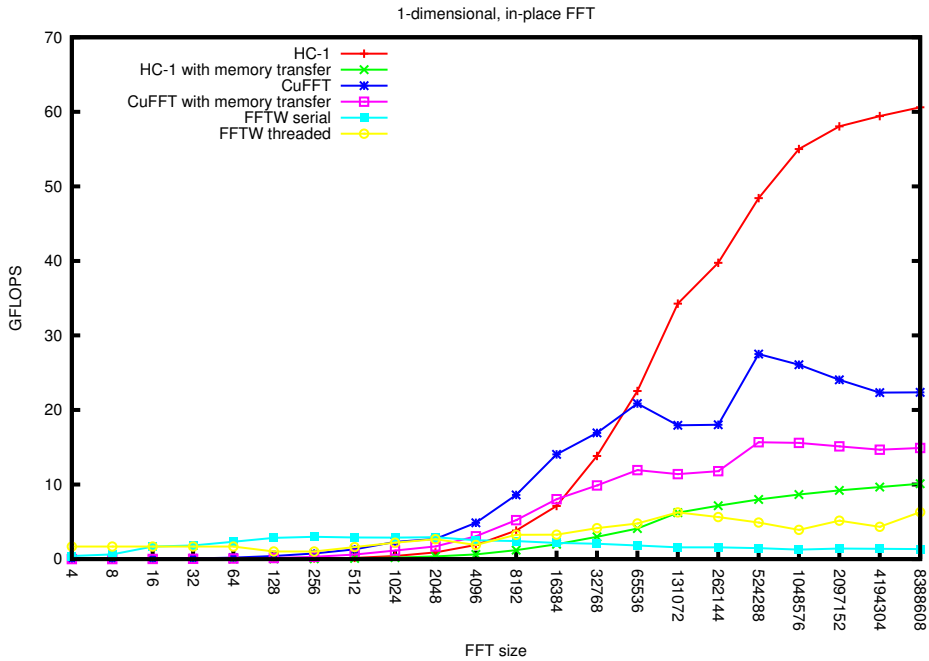


Figure 6.8: 1-dimensional in-place single-precision complex-to-complex FFT.

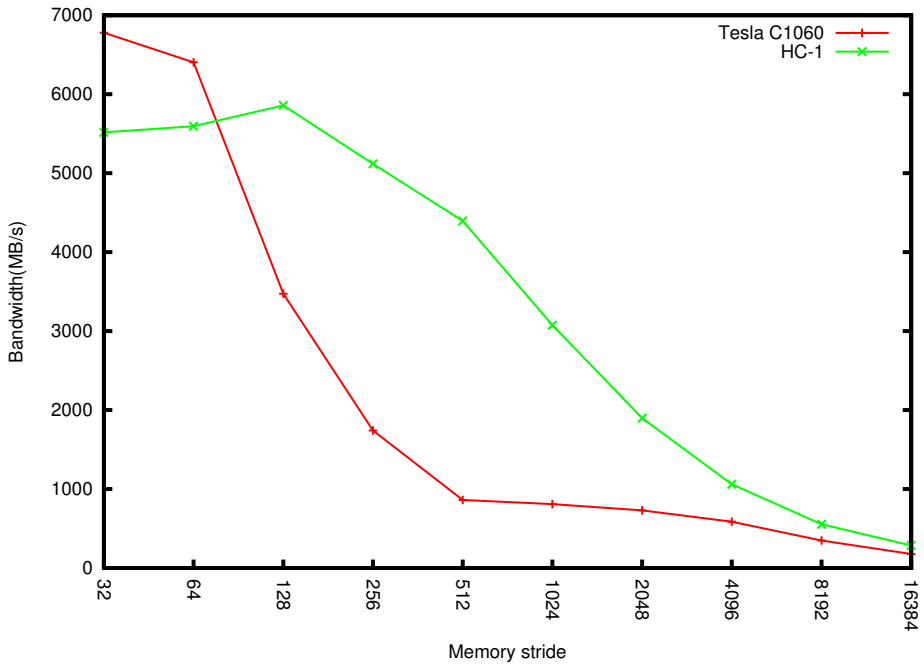


Figure 6.9: Memory bandwidth for stride memory access.

6.8.4 RandomAccess

Table 6.4 shows the performance of the RandomAccess benchmark on CPU, GPU, and HC-1. For the CPU implementation, we use the reference code provided by HPCC, and execute the benchmark on single CPU code. HC-1 outperforms both CPU and GPU by a factor of 40 and 6.7 respectively. This is due to the memory subsystem of Convey HC-1 coprocessor which is optimised for both sequential and random accesses.

6.8.5 Monte-Carlo methods: Asian option pricing

Table 6.5 shows the performance of an HC-1 implementation of an Asian option pricing application, compared to the performance of CPU and GPU implementations. For the Asian option pricing benchmark, we select the same parameters as in [90], i.e. one million simulations over a time period of 356 steps. We use an Intel Xeon 5138 dual-core processor running at 2.13GHz with 8GB of memory for the CPU implementation. We also include performance results of an optimised FPGA implementation [90] coded in a hardware description language (HDL) targeting a Xilinx xc5lx330t FPGA clocked at 200MHz.

These results show that using the HC-1 coprocessor yields a performance improvement of up to 18 times over an optimised multi-threaded software implementation. The performance of the HC-1 is about the same as a single precision GPU implementation, and is twice as fast as the double precision version. The major reason for this performance improvement is the vectorisation of the FOR loops, which form the bottleneck in the option pricing benchmark. Moreover, the random number generator is implemented in the HC-1 as a custom hardware library, whereas the CUDA GPU must use an instruction based approach. Note that currently the finance analytics personality for HC-1 does not support single-precision floating-point operations, so it cannot be compared directly with a hand-crafted single-FPGA version in single-precision floating-point arithmetic [90] also shown in Table V; however, there is no doubt that the hand-crafted version involves much more development effort.

Table 6.4: Performance results for the RandomAccess benchmark

Implementation	CPU single-core	Tesla C1060	Convey HC-1
Main Memory size	16 GB	1 GB	16 GB
GUPS	0.02	0.12	0.8
Speedup	1x	6x	40x

Table 6.5: Performance results for Asian option pricing.

Implementation	Execution time		Speed-up	
	Single	Double	Single	Double
CPU single-threaded	8,333 ms	14,727ms	0.53x	0.57x
CPU multi-threaded	4,446 ms	8,378 ms	1x	1x
Convey HC-1	-	471 ms	-	17.8x
Tesla C1060 [90]	440 ms	1,078 ms	10x	7.7x
HDL-coded FPGA [90]	115 ms	-	38.6x	-

6.9 Summary

This chapter provides a comparison of FPGAs and GPUs for high-productivity computing. Using a number of established benchmarks and a common development model, we evaluate the performance of a commercially-available high-performance reconfigurable computing (HPRC) system, the Convey HC-1 server. We also compare its performance to those of a multi-core CPU as well as a GPU-based HPC system. The selected benchmarks for our work have different memory access patterns ranging from high locality memory characteristics to low locality characteristics. The HC-1 and the GPU outperform the CPU for all of our benchmark programs. From our results, we can summarise the following about some of the pros and cons of the HPRCS and the GPU:

- GPUs often perform faster than FPGAs for streaming applications. GPUs usually enjoy a higher floating-point performance and memory bandwidth than FPGA-based HPC systems. The streaming performance of the HC-1 is often limited by the floating-point computational performance achieved by FPGAs.
- The HC-1 demonstrates superior performance for highly parallel applications requiring low memory bandwidth, as illustrated by the Monte Carlo benchmark for pricing Asian options.
- The HC-1 employs a heavily-banked parallel memory subsystem optimised for non-sequential memory accesses, which makes them faster than GPUs for applications such as Fast Fourier Transform. GPUs, on the other hand, use GDDR memories which are optimised for sequential memory access operations, incurring a higher performance penalty for non-contiguous memory block transfers.

Chapter 7

Conclusion

7.1 Summary of Thesis Achievements

This thesis presents a reconfigurable computing approach to efficient processing of large-scale graph traversal problems. The key points of the main chapters in this thesis are summarised as follows.

7.1.1 High Performance Reconfigurable Computing for Efficient Parallel Graph Traversal

Chapter 3 presents a novel reconfigurable hardware approach for efficient processing of large-scale graph traversal problems, which exploits the capabilities of both FPGAs and multi-bank memory subsystems. Our approach is based on a reconfigurable hardware architecture that decouples computation and communication while keeping multiple memory requests in flight at any given time, and so takes advantages of the high bandwidth of a multi-bank memory subsystem. To validate our methodology, we provide a detailed description of an FPGA-based accelerator of the well-known BFS problem using an actual high performance reconfigurable computing system. Using graph data based on power-law graphs found in real-world problems, we achieve performance results that are superior to those of high performance multi-core CPU systems from the recent literature. Our design delivered a processing rate in excess of 6.3 billion traversed edges per second on R-MAT graphs with 16 million vertices and over a billion edges. Using four Virtex-5 LX330 FPGAs based on 65nm technology and running at 100 MHz, our FPGA–based BFS design achieved more than 3.5 times the speed of a 32-core Xeon X7560 based on 45nm technology and running at 2.26GHz.

Our reconfigurable computing approach to efficient processing of large-scale graph traversal problems relies

on both hardware customisation and a high performance multi-bank memory subsystem to achieve optimum results. In our work, we use a high-performance reconfigurable computing system that has a multi-bank memory subsystem (1024 memory banks). Such a sophisticated memory subsystem may not be available on other FPGA-based HPC systems. While our reconfigurable hardware architecture assumes an arbitrary number of memory banks and/or memory controllers, it remains to be seen how performance is affected when our hardware design is ported to another FPGA-based platform with a higher or lower number of memory banks than that of the Convey HC-1 system.

In our performance comparison to multi-core CPUs and GPUs, our FPGA-based HPC platform is a little bit outdated as its FPGA devices (Virtex-5 LX330) are based on 65nm technology compared to the CPUs and GPUs which are based on 45nm. In addition, our FPGA-based system has a memory system built from DDR2-SDRAM modules, whereas the CPU–based systems have access to DDR3-SDRAM technology. This means that our FPGA-based acceleration can go faster if we are to use the most recent technologies such as the Convey HC-2 system which has Virtex-6 FPGAs (compared to Virtex-5 in HC-1), and DDR3 memory technology (compared to DDR2 memories in HC-1). While going from HC-1 to HC-2 leads in general to a speedup of 2-3x (according to the manufacturer), it is not clear how much performance speedup can be obtained for parallel graph traversal algorithms if we are to port our current design from HC-1 to HC-2. From a silicon area perspective, we can say that by using Virtex-6 FPGAs we can more than double the number of processing elements without a drop in clock frequency leading to about a 2 times speedup, assuming we can keep all the processing elements busy.

7.1.2 Reconfigurable hardware acceleration of graphlet counting

Chapter 4 further evaluates our reconfigurable hardware methodology with a case study from bioinformatics, namely *graphlet counting*. Unlike in the case of BFS where only the adjacency-list representation of graphs is used, efficient processing of the graphlet counting algorithm requires both the adjacency-list representation as well as the adjacency-matrix representation of the input graph data. The adjacency matrix is used in the graphlet counting algorithm to perform adjacency tests and requires random memory accesses.

We present an actual hardware implementation of the graphlet counting algorithm that is efficient and scalable. A performance comparison with a multi-core CPU implementation shows that our FPGA-based graphlet counting design is more than 2 times faster than a multi-core CPU implementation for large graph instances. This speedup includes all software and I/O overhead requirements. These results demonstrate that our method-

ology for accelerating large-scale graph traversal problems is a promising approach for efficient parallel graph traversal.

We also propose an improved version of the graphlet counting algorithm that is more efficient than the conventional graphlet counting algorithm in terms of both performance and memory storage requirements. The proposed algorithm allows for larger input graphs to be processed in comparison with the conventional version that relies on the adjacency-matrix representation for optimum performance results. Instead of using the standard adjacency-matrix representation, we design a different data structure that only stores a small part of the adjacency-matrix. To be more precise, we only store rows in the adjacency-matrix that represent large-degree vertices. For low-degree vertices, we modify the graphlet counting algorithm so that adjacency-lists are used to perform adjacency tests.

One of the limitations of this work is the lack of support for atomic operations that are needed to update certain counters in the graphlet counting algorithm. The graphlet counting algorithm enumerates two types of graphlets: global counters of 3-,4-, and 5-node graphlets (29 counters in total), and local counters where each node in the graph has 72 counters ($|V| \times 72$ counters in total). Unlike the global counters, local counters require atomic operations to avoid read-after-write data hazards that may occur due to a counter being updated by two processing elements simultaneously. As a result, our current implementation of the graphlet counting algorithm only supports global counters since our current reconfigurable hardware architecture for graph traversal problems does not support atomic operations at the moment. Convey HC-1, the FPGA-based HPC system used in our work, has a sophisticated memory subsystem with memory controllers implemented outside the user-accessible FPGA devices. These memory controllers re-order memory operations to achieve optimum memory bandwidths, which makes the implementation of atomic operations by the user a very challenging task. It would be of interest to find out the achievable memory bandwidth, and hence the performance of the graphlet counting algorithm, when atomic operations are implemented.

7.1.3 Parallel FPGA-based All-Pairs Shortest-Paths for sparse networks

Chapter 5 presents a highly parallel and scalable reconfigurable hardware design for the All-Pairs Shortest-Paths (APSP) problem for sparse networks. This work builds on our reconfigurable hardware approach presented in Chapter 3, where the reconfigurable architecture is augmented with an on-chip memory capabilities for efficient processing of the APSP problem in directed graphs with unweighted edges. We propose in this chapter a parallelisation methodology that leverage the benefits of FPGAs for the BFS algorithm when it is used to solve

the APSP problem. We also present a detailed description of an implementation of our FPGA-based APSP solver on an actual high performance reconfigurable computing system. Using both synthetic graph instances, as well as human brain network data, we are able to achieve performance results superior to those of multi-core CPU and GPU implementations, while attaining linear scaling over the number of processors introduced. Our FPGA-based APSP design is about 10 times faster than a quad-core CPU implementation and 2-5 times faster than an AMD Cypress GPU implementation. Note that our FPGA-based system is built using Virtex-5 FPGAs which are based on 65 nm technology, whereas the GPU used in our work is based on 40 nm technology.

Since we use a private on-chip memory scheme where each GPE has exclusive access to a block of on-chip memory, we achieve linear performance scalability as the number of GPEs increases. However, the size of these private memories depends of the problem size (i.e. the number of vertices in the input graph), and hence, the number of private memories is limited by the size of the input graph. This in turn will limit the number of GPEs or achievable parallelism for our FPGA-based APSP design. One way to solve this problem is to locally share on-chip memories. For example, each two GPEs can have exclusive access to the same on-chip memory block. While this may allow for the instantiation of a larger number of GPEs, and hence, greater parallelism, it doesn't come without a performance penalty. Sharing on-chip memories will lead to access contention, which will increase the access latency. In addition, our current design use a global clock rated at 150MHz for both GPEs and on-chip memories. A better approach is to use multiple clock domains: a slow clock frequency (150MHz) for the GPEs and a fast clock frequency (300 MHz) for the on-chip memories. This will eliminate the access contention overhead and keep the access latency to on-chip memories to one clock cycle from the GPE's point of view.

7.1.4 Comparing performance of FPGAs and GPUs for high-productivity computing

Chapter 6 evaluates the performance of FPGAs for high-productivity computing, where a commodity CPU instruction set architecture is augmented with instructions which execute on a specialised FPGA co-processor, allowing the CPU and FPGA to co-operate closely while providing a programming model similar to that of traditional software. To compare the GPU and FPGA approaches, we use a set of established benchmarks with different memory access characteristics, and compare their performance on an FPGA-based high-performance computing system with a GPU-based system. Our results show that GPUs excel at streaming applications with regular memory accesses, in comparison to the FPGA-based system. However, the FPGA-based system outperforms the GPU in applications with poor memory locality characteristics and random or irregular memory access patterns.

A key idea in this work is the utilisation of benchmarks with different memory characteristics. This has allowed us to expose a weakness in GPU-based systems with regards to applications with low memory access characteristics. GPUs have originally been designed for graphics applications which fit onto a streaming model, and hence the memory subsystem of GPUs was designed to meet those streaming requirements. However, not all applications fit nicely onto the streaming model due to several factors such as the random and irregular memory access pattern requirements. FPGA-based systems, on the other hand, can have a more flexible memory subsystem that can be tailored for application-specific memory access characteristics.

A weakness of the benchmarks used in our study is that each benchmark is dominated by a certain memory access pattern; i.e. memory access locality characteristics. A benchmark that combines different memory locality characteristics may provide more insights into which platform may be best in general. For example, a compute cluster may run various applications with different memory access characteristics, so it would be useful to determine which compute platform is best for such a scenario.

7.2 Future Work

The section describes possible future extensions to the work presented in this thesis.

7.2.1 Reconfigurable computing for efficient parallel graph traversal

- *Soft core processors*. In our current approach, we use HDL-coded processing elements that are application-specific, which means that for each new algorithm, a new processing element needs to be hand-coded. While this leads to optimised performance results, it requires high design efforts as it involves hardware development using a hardware description language. A more productive approach would be to replace the HDL-code processing elements with soft-core processors such as MicroBlaze [66]. This will greatly reduce design efforts as soft cores are programmed using high-level languages. We expect the soft core approach to be less efficient than the current approach which involves application-specific processing elements. Therefore, it would be useful to investigate the trade-off between performance and design efforts.

- *Customising the memory interconnect network*. The current memory interconnect network connects the GPEs to off-chip memory without performing any advanced techniques to improve the overall performance of the system. Through customisation, the memory interconnect can be optimised to support memory operations other than simple memory read and write operations. For example, implementing atomic increment operations in the interconnect network can lead to better resource sharing as all GPEs can use the same increment operator. Dynamic reordering of memory requests to avoid page misses in DRAMs can be another optimisation to improve the effective memory bandwidth. Our reconfigurable hardware architecture allows aggressive re-ordering of memory requests, due to the large number of GPEs generating memory requests.

7.2.2 Reconfigurable hardware acceleration of graphlet counting

The performance and scalability of our reconfigurable hardware architecture are already satisfactory compared to general-purpose processors. Having said that, we are investigating the following techniques to improve the performance and efficiency of our reconfigurable architecture:

- *Multi-threaded GPEs*. We have shown that the performance of our hardware solution scales almost linearly with the number of GPEs, indicating that memory bandwidth is not saturated. The next step

is to increase the number of concurrent memory requests, or the number of parallel requesters. This can be achieved by increasing the number of GPEs, but this number will be limited by the size of the reconfigurable device. A better approach is to enable support of multiple concurrent threads within a single GPE, which offers better device resource utilisation while reducing the number of stall cycles in GPEs caused by long waits for memory fetches.

- *Dynamic workload balancing*. Scheduling of the tasks assigned to GPEs can be further optimised to improve the overall performance. Currently we use a static scheduler that may result in unbalanced workloads for the GPEs. A dynamic approach should in principle improve the workload balance amongst the GPEs. This will benefit the performance of our FPGA design for scale-free or power-law graphs which are more prevalent in real-world graphs than uniformly random graphs.

- *GPU implementation*. In our work, we compare our reconfigurable hardware graphlet counter to a multi-core CPU implementation. Another promising candidate for the acceleration of the graphlet counting algorithm is many-core GPUs. It would be useful to include GPUs in our performance comparison given that the data size in graphlet counting problems should fit onto a GPU's global memory, and that GPUs have shown promising results for other graph algorithms [8], [9].

7.2.3 Parallel and scalable FPGA-based APSP solver

- *Locally shared-memories*. In our current FPGA design for APSP-BFS, we use private on-chip memories (each GPE has exclusive access to some on-chip memory resources). As we explained in Section 5.3, using locally-shared memories (a set of on-chip memory resources being shared by more than one GPE) may become a necessity if we are to process larger input graphs, since locally-shared memories will allow us to have a greater number of GPEs than private memories for large graph instances.

- *Support for weighted edges*. Our current FPGA-based APSP design only supports unweighted edges as it implements the BFS algorithm. We can extend our design to support weighted edges, and hence, implement Johnson's algorithm [60] which is similar to BFS except that edges are weighted. This will also involve floating-point or fixed-point operations as weights are usually represented using real numbers. Depending on the application, the floating-point or fixed-point operators can be customised in hardware to match the precision and/or range of real numbers required by the application running the APSP algorithm.

- *GPU implementation of APSP-BFS*. Currently, in our comparison of FPGAs against GPUs for the APSP

problem, we use the BFS algorithm for the FPGA implementation, whereas we use the Floyd-Warshall (FW) algorithm for the GPU. While both algorithms solve the APSP problem, we explained in Section 5.1.1 how APSP-FW favours dense networks as opposed to APSP-BFS which favours sparse networks. Hence, a GPU implementation of APSP-BFS would certainly improve the quality of our comparison between FPGAs and GPUs for the APSP problem.

7.2.4 Comparing performance of FPGAs and GPUs for high-productivity computing

- *Soft general-purpose processors*. In our evaluation of the performance of FPGAs for high-productivity computing, we use domain-specific soft-cores (i.e. Convey's single-precision vector personality and the finance analytics personality). While these domain-specific soft-cores allow us to program the FPGA using a development model similar to that of CPUs and modern many-core GPUs, we are restricted in terms of the target applications. For example, developing applications with random memory access patterns such as the RandomAccess benchmark was not possible as these domain-specific soft-cores did not support random memory accesses. An alternative approach is be to build a many-soft-core system using off-the-shelf general purpose soft-core processors such as MicroBlaze [66]. This will allow us to benefit from the flexibility of such soft-core processors, while providing us with a rich and mature development platform.

7.3 Final Conclusions

This chapter concludes the thesis with a summary of the main contributions and a discussion of directions for future work. The goal of the thesis is to investigate the possible acceleration of graph traversal algorithms for large-scale graph problems using FPGA-based reconfigurable computing. Our research has made promising progress in achieving this goal, and we hope that work on this approach will continue in the future, particularly in the directions of the future work mentioned in this chapter.

Bibliography

[1] "The Graph 500 List, June 2012," http://www.graph500.org/results_june_2012, 2012.

[2] Y. Wang, M. Xu, L. Ren, X. Zhang, D. Wu, Y. He, N. Xu, and H. Yang, "A heterogeneous accelerator platform for multi-subject voxel-based brain network analysis," in *ICCAD*, 2011, pp. 339–344.

[3] K. Matsumoto, N. Nakasato, and S. Sedukhin, "Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system," in *HPCC*, 2011, pp. 145–152.

[4] O. Kuchaiev, A. Stevanović, W. Hayes, and N. Pržulj, "Graphcrunch 2: Software tool for network modeling, alignment and clustering," *BMC bioinformatics*, vol. 12, no. 1, p. 24, 2011.

[5] "The Graph 500 List," http://www.graph500.org/index.html, 2010.

[6] A. Lumsdaine *et al.*, "Challenges in parallel graph processing." *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[7] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA." in *HiPC*, vol. 4873. Springer, 2007, pp. 197–208.

[8] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU." in *PACT*, 2011, pp. 78–88.

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012.

[10] A. Buluc, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU." *Parallel Computing*, vol. 36, no. 5-6, pp. 241–253, 2010.

[11] M. DeLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. K. Jr., and A. DeHon, "GraphStep: a system architecture for sparse-graph algorithms," in *FCCM*, 2006, pp. 143–151.

[12] U. Bondhugula *et al.*, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," in *IPDPS*, April 2006, pp. 1–10.

[13] K. Sridharan, T. Priya, and P. Kumar, "Hardware architecture for finding shortest paths," in *TENCON*, 2009.

[14] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on FPGA for breadth-first search," in *FPT*, 2010.

[15] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69 –76, Feb. 2008.

[16] R. Diestal, *Graph Theory*, Fourth ed., ser. Graduate Texts in Mathematics. Springer-Verlag, Heidelberg, July 2010, vol. 173.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[18] N. Przulj, D. G. Corneil, and I. Jurisica, "Efficient estimation of graphlet frequency distributions in protein-protein interaction networks," *Bioinformatics*, vol. 22, no. 8, pp. 974–980, 2006.

[19] D. Knoke and S. Yang, *Social Network Analysis*. SAGE Publications, 2008, vol. 154.

[20] N. Przulj, "Graph theory analysis of protein-protein interactions," in Knowledge Discovery in Proteomics, I. Jurisica and D. Wigle (eds.), CRC Press, 2005.

[21] N. Przulj and T. Milenkovic, "Computational methods for analyzing and modeling biological networks," in Biological Data Mining, J. Chen and S. Lonardi, CRC Press, to appear.

[22] J. loup Guillaume and M. Latapy, "The web graph: an overview," in *AlgoTel*, 2002.

[23] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in *IPDPS*, May 2009, pp. 1 –9.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro-Institute of Electrical and Electronics Engineers*, vol. 28, no. 2, pp. 39–55, 2008.

[25] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[26] C. Chang, K. Kuusilinna, B. C. Richards, and R. W. Brodersen, "Implementation of BEE: a real-time large-scale hardware emulation engine," in *FPGA*, 2003, pp. 91–99.

[27] R. Baxter, S. Booth, M. Bull, G. Cawood, K. D'Mellow, X. Guo, M. Parsons, J. Perry, A. Simpson, and A. Trew, "High-performance reconfigurable computing - the view from Edinburgh," in *AHS*. IEEE Computer Society, 2007, pp. 373–279.

[28] A. Yoo, E. Chow, K. Henderson, W. Mclendon, B. Hendrickson, and U. C. urek, "A scalable distributed parallel breadth-first search algorithm on Bluegene/l," in *SC*, 2005, p. 25.

[29] D. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the Cell/BE processor," *Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381 –1395, Oct. 2008.

[30] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors." in *SC*, 2010, pp. 1–11.

[31] ı. V. ÇAtalyüRek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Comput.*, vol. 38, no. 10-11, pp. 576–594, Oct 2012.

[32] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency." in *IPDPS*, 2012, pp. 378–389.

[33] A. Dandalis, A. Mei, and V. Prasanna, "Domain specific mapping for solving graph problems on reconfigurable devices," *Parallel and Distributed Processing*, pp. 652–660, 1999.

[34] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient multi-context graph processors," in *FPL*, vol. 2438, 2002, pp. 915–924.

[35] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and ST-connectivity on the Cray MTA-2," in *ICPP*, Aug. 2006, pp. 523 –530.

[36] K. Madduri, D. Bader, J. Berry, and J. Crobak, "Parallel shortest path algorithms for solving large-scale instances," *9th DIMACS Implementation Challenge-Shortest Paths*, 2006.

[37] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," *International Parallel and Distributed Processing Symposium*, pp. 1–8, 2009.

[38] 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 *SIGMOD '10*, 2010, pp. 135–146.

[39] A. Chan, F. Dehne, and R. Taylor, "CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters and shared memory machines," in *International Journal of High Performance Computing Applications*, vol. 19, no. 1. Springer, 2005, pp. 81–97.

[40] D. Gregor and A. Lumsdaine, "The parallel BGL: a generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[41] U. Meyer and N. Zeh, "I/O-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths," *ACM Trans. Algorithms*, vol. 8, no. 3, pp. 22:1–22:28, jul 2012.

[42] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[43] *The Boost Graph Library: user guide and reference manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[44] J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures," *SPIE Photonics East: RTRPDC*, pp. 225–236, 1996.

[45] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.

[46] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, vol. 393, no. 6684, pp. 440–442.

[47] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on FPGAs," in *IAAA*, 2011, pp. 31–34.

[48] S. Beamer, K. Asanovi, and D. A. Patterson, "Searching for a parent instead of fighting over children: a fast breadth-first search implementation for Graph500," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, Nov 2011.

[49] D. A. Bader and K. Madduri, "GTgraph: A suite of synthetic random graph generators," 2006. [Online]. Available: http://www.cse.psu.edu/ madduri/software/GTgraph/index.html

[50] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM International Conference on Data Mining*, 2004, p. 541.

[51] *Convey Personality Development Kit Reference Manual*, Convey Computer™, November 2011.

[52] J. D. Bakos, "High-performance heterogeneous computing with Convey HC-1," *Computing in Science and Engineering*, vol. 12, no. 6, pp. 80–87, 2010.

[53] Y. Xia and V. K. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in *PDCS*, vol. 668, no. 077, Nov 2009, p. 91.

[54] Convey Computer™, "Convey coprocessor memory management," White paper, Convey Computer, 2012.

[55] T. Milenkovic, J. Lai, and N. Przulj, "GraphCrunch: a tool for large network analyses," *BMC Bioinformatics*, 2008.

[56] N. Przulj, D. A. Wigle, and I. Jurisica, "Functional topology in a network of protein interactions," *Bioinformatics*, vol. 20, no. 3, pp. 340–348, 2004.

[57] O. Kuchaiev, T. Milenkovic, V. Memisevic, W. Hayes, and N. Przulj, "Topological network alignment uncovers biological function and phylogeny," *Journal of The Royal Society Interface*, vol. 7, no. 50, p. 1341, Oct 2009.

[58] T. Milenkovic and N. Przulj, "Uncovering biological network function via graphlet degree signatures," Department of Computer Science, University of California, Irvine, CA 92697-3435, U.S.A, Tech. Rep. Technical Report No. 08-01, Feb 2008.

[59] David Levinthal, "Cycle accounting analysis on Intel Core 2 processors," Intel Corp., Tech. Rep., 2006.

[60] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *JACM*, vol. 24, no. 1, pp. 1–13, 1977.

[61] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 296–271, 1959.

[62] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[63] R. W. Floyd, "Algorithm 97: Shortest path," *CACM*, vol. 5, no. 6, pp. 345–, 1962.

[64] "NITRC: the source of neuroimaging tools and resources," http://www.nitrc.org/projects/fcon_1000.

[65] K. Matsumoto and S. G. Sedukhin, "A solution of the all-pairs shortest paths problem on the Cell Broadband Engine processor." *IEICE Transactions*, vol. 92-D, no. 6, pp. 1225–1231, 2009.

[66] Xilinx, "Microblaze Soft Processor Core." http://www.xilinx.com/tools/microblaze.htm.

[67] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "ImpulseC vs. VHDL for accelerating tomographic reconstruction," in *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 171–174.

[68] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. J. Eggers, "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures," in *FPL*. IEEE, 2008, pp. 173–178.

[69] B. Cope, P. Y. K. Cheung, W. Luk, and L. W. Howes, "Performance comparison of graphics processors to reconfigurable logic: a case study," *IEEE Trans. Computers*, vol. 59, no. 4, pp. 433–448, 2010.

[70] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: a comparison study," in *16th International Symposium on Field-Programmable Custom Computing Machines. FCCM'08*, 2008, pp. 173–182.

[71] D. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *Proceeding of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2009, pp. 63–72.

[72] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A comparative study on ASICs, FPGAs, GPUs and general purpose processors in the O(N^2) gravitational N-body simulation," *NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 447–452, 2009.

[73] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *SASP*. IEEE, 2008, pp. 101–107.

[74] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung, "GPU versus FPGA for high productivity computing," in *FPL*, 2010, pp. 119–124.

[75] C. W. Fletcher, I. A. Lebedev, N. B. Asadi, D. R. Burke, and J. Wawrzynek, "Bridging the GPGPU-FPGA efficiency gap," in *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 119–122.

[76] Xilinx AutoESL High-Level Synthesis Tool, http://www.xilinx.com/tools/autoesl.htm.

[77] Cadence TLM-driven design and verification methodology, http://www.cadence.com/products/fv/tlm.

[78] T. Bollaert, *Catapult synthesis: a practical introduction to interactive C synthesis*. Heidelberg: Springer, 2008.

[79] Maxeler Technologies , http://www.maxeler.com.

[80] M. Zelkowitz, V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura, "Measuring productivity on high performance computers," in *Proceedings of the International Symposium on Software Metrics*, 2005, pp. 19–22.

[81] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC challenge benchmark suite," Lawrence Berkeley National Laboratory, Tech. Rep., 2005.

[82] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[83] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, and Y. Kaneda, "16.4-TFLOPS direct numerical simulation of turbulence by a Fourier spectral method on the earth simulator," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–17.

[84] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[85] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," *SC Conference*, pp. 1–11, 2008.

[86] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*. IEEE Press, 2008, pp. 1–11.

[87] T. G. Lewis and W. H. Payne, "Generalized feedback shift register pseudorandom number algorithm." *J. ACM*, vol. 20, no. 3, pp. 456–468, 1973.

[88] N. Meteopolis and S. Ulam, "The Monte-Carlo method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

[89] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–54, May-June 1973.

[90] A. H. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk, "Efficient reconfigurable design for pricing asian options," *SIGARCH Comput. Archit. News*, vol. 38, no. 4, pp. 14–20, Jan. 2011.

[91] Convey Computer™, *Convey Reference Manual*, 2010.

[92] *Convey Programmers Guide*, Convey Computer™, 2010.

[93] Convey Computer™, "Convey Mathematical Libraries User's Guide," 2010.

[94] NVIDIA Corporation, "Compute Unified Device Architecture," http://www.developer.nvidia.com/object/cuda.html.

[95] Lebedev, I. and Shaoyi Cheng and Doupnik, A. and Martin, J. and Fletcher, C. and Burke, D. and Mingjie Lin and Wawrzynek, J., "MARC: A many-core approach to reconfigurable computing," in *Reconfigurable Computing and FPGAs (ReConFig)*, 2010, pp. 7–12.

[96] *CUDA CUBLAS Library*, NVIDIA Corp., 2007.

[97] *CUDA CUFFT Library*, NVIDIA Corp., 2007.

[98] Intel Corp., "Intel Math Kernel Library," http://software.intel.com/en-us/intel-mkl/.

[99] FFTW, "Fastest Fourier Transform in the West – Homepage," http://www.fftw.org/, 2003.

[100] R. Chen, L. Li, and Z. Weng, "ZDOCK: an initial-stage protein-docking algorithm," *Proteins: Structure, Function, and Bioinformatics*, vol. 52, no. 1, pp. 80–87, 2003.

Appendix A

Graphlet Counting Source Code

```
/*
 * Implementation of the Graphlet counting algorithm
 * Source code extracted from the GraphCrunch tool
 *
 * Function: Counts graphlets and graphlet degrees (called node classes here)
 * Basic algorithm: Brute force enumeration of 3-5 node connected subgraphs
 * With some overlap that needs to be factored out later.
 *
 * Process is:
 *   Pick node A, pick a node B adjacent to A, pick a node C adjacent to B, pick a
 *   node D adjacent to C, pick a node E adjacent to D. And each node can only
 *   appear once in the subgraph.
 *
 *   Examine the edges between them to determine which graphlet the
 *   subgraph corresponds to. Classify each node in the graphlet and add
 *   it to the count for that graphlet type.
 *
 *  Inputs:
        v_size: size of the graph (|V|)
        adj_offset[]: contains offset of nodes in the adjacency list
        adj_list[]: compact adjacency lists of all nodes of the input graph
        adj_mat[][]: the adjacency matrix of the input graph.

    Outputs:
        gcount[]: 3-,4-, and 5-node graphlet counters
 */
```

```
27
28 #include <stdlib.h>
29 #include <stdio.h>
30 #include <string.h>
31 #include <math.h>
32
33 /* 3-, 4-, and 5-node graphlet */
34 enum {P3, C3, P4, CLAW, C4, FLOW, DIAM, K4, P5, X10, X11, X12, X13,
35 X14, C5, X16, X17, X18, X19, X20, X21, X22, X23, X24, X25, X26,
36 X27, X28, K5_A};
37
38 /* hash table */
39 const char gtable [][8] =
40 {{-1,-1,10,-1,-1,8, -1,-1},
41  {-1,11,-1,-1,15,14,12,-1},
42  {17,19,-1,16,18,20,-1,-1},
43  {-1,-1,23,24,-1,21,-1,-1},
44  {-1,-1,26,25,-1,-1,-1,-1},
45  {-1,-1,-1,-1,-1,-1,27,-1},
46  {28,-1,-1,-1,-1,-1,-1,-1}
47 };
48
49 void *gcounter (int v_size, int* adj_offset, int* adj_list, int** adj_mat, int* gcount) {
50     int off_b, off_c, off_d, off_e;
51     int endoff_b, endoff_c, endoff_d, endoff_e;
52     int a, b, c, d, e, x;
53
54     for (a=0; a<v_size; a++){
55
56         off_b = adj_offset[a];
57         endoff_b = adj_offset[a+1];
58
59         for (off_b; off_b < endoff_b; off_b++){
60             b = adj_list[off_b]; /* read neighbour b of a from adjacency-list */
61
62             off_c = adj_offset[b]
63             endoff_c = adj_offset[b+1];
64
65             for (off_c ; off_c < endoff_c; off_c++) {
```

```
66                  /* read neighbour c of b from adjacency-list */
67                 c = adj_list[off_c];
68                 if (c == a) continue ; /* ignore self-loops */
69
70                 int deg3_a =1;
71                 x = adj_matrix[a][c]; /* adjacency test (a,c)*/
72                 deg3_a += x;
73
74                 // classify 3-node graphlets
75                 if(deg3_a == 1)
76                     gcount[P3]++; /* path */
77                 else
78                     gcount[C3]++; /* triangle */
79
80                 off_d = adj_offset[c];
81                 endoff_d = adj_offset[c+1];
82
83                 for (off_d; off_d < endoff_d; off_d++) {
84                     /* read neighbour d of c from adjacency-list */
85                     d = adj_list[off_d];
86                     if(d == a || d == b) continue;
87
88                     int deg4_a = deg3_a;
89                     int deg4_c = deg3_a+1;
90                     int deg4_b = 2;
91                     int deg4_d = 1;
92
93                     x = adj_mat[a][d]; /* adjacency test (a,d)*/
94                     deg4_d += x; deg4_a += x;
95
96                     x = adj_mat[b][d]; /* adjacency test (b,d)*/
97                     deg4_d += x; deg4_b += x;
98
99                     int num_edges = deg4_a + deg4_b + deg4_c + deg4_d;
100
101                    // classify 4-node graphlets
102                    if(num_edges == 6){
103                        gcount[P4]++; // P4
104                    }
```

```
105                 else if(num_edges == 10)
106                 {
107                     gcount[DIAM]++; /* Diamond */
108                 }
109                 else if(num_edges == 12)
110                 {
111                     gcount[K4]++; /* K4 */
112                 }
113                 else if(num_edges == 8) // C4 or Flower
114                 {
115                     if(deg4_b == 3 || deg4_c == 3)
116                         gcount[FLOW]++; /* Flower */
117                     else
118                         gcount[C4]++; /* C4 */
119                 }
120
121                 /* classify 5-node graphlets */
122                 off_e = adj_offset[d]
123                 endoff_e = adj_offset[d+1];
124
125                 for (off_e; off_e < endoff_e; off_e++) {
126                     /* read neighbour e of d from adjacency-list */
127                     e = adj_list[off_e];
128                     if(e == a || e == b || e == c || e == d) continue;
129
130                     int deg5_a = deg4_a, deg5_b = deg4_b, deg5_c = deg4_c,
131                     deg5_d = deg4_d + 1, deg5_e = 1;
132
133                     x = adj_mat[a][e]; /* adjacency test (a,e)*/
134                     deg5_e += x; deg5_a += x;
135
136                     x = adj_mat[b][e]; /* adjacency test (b,e)*/
137                     deg5_e += x; deg5_b += x;
138
139                     x = adj_mat[c][e]; /* adjacency test (c,e)*/
140                     deg5_e += x; deg5_c += x;
141
142                     // add degrees of node and neighbors to find each ndeg
143                     int ndeg_a = deg5_a + deg5_b;
```

```
144                         ndeg_a += (deg3_a == 2) ? deg5_c : 0;
145                         ndeg_a += (deg4_a == deg3_a + 1) ? deg5_d : 0;
146                         ndeg_a += (deg5_a == deg4_a + 1) ? deg5_e : 0;
147
148                         int ndeg_b = deg5_b + deg5_a + deg5_c;
149                         ndeg_b += (deg4_b == 3) ? deg5_d : 0;
150                         ndeg_b += (deg5_b == deg4_b + 1) ? deg5_e: 0;
151
152                         int ndeg_c = deg5_c + deg5_b + deg5_d;
153                         ndeg_c += (deg3_a == 2) ? deg5_a : 0;
154                         ndeg_c += (deg5_c == deg4_c + 1) ? deg5_e: 0;
155
156                         int ndeg_d = deg5_d + deg5_e + deg5_c;
157                         ndeg_d += (deg4_a == deg3_a + 1) ? deg5_a : 0;
158                         ndeg_d += (deg4_b == 3) ? deg5_b : 0;
159
160                         int ndeg_e = deg5_e + deg5_d;
161                         ndeg_e += (deg5_a == deg4_a + 1) ? deg5_a : 0;
162                         ndeg_e += (deg5_b == deg4_b + 1) ? deg5_b: 0;
163                         ndeg_e += (deg5_c == deg4_c + 1) ? deg5_c: 0;
164
165                         int hash = (ndeg_a % 4 + ndeg_b % 4 + ndeg_c % 4 + ndeg_d % 4 +
                                ndeg_e % 4);
166                         int deg_total = deg5_a + deg5_b + deg5_c + deg5_d + deg5_e;
167                         int gtype = gtable[deg_total/2 - 4][hash/2];
168
169                         // not caught by gtable
170                         if(deg_total == 14 && hash == 6)
171                             gtype = (ndeg_a > 12 || ndeg_a == 5) ? 22 : 24;
172                         assert(gtype > 7 && gtype < 29);
173
174                         gcount[gtype]++;
175                     }
176                 }
177             }
178         }
179     }
180 }
```