A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study

Brahim Betkaoui, David B. Thomas, Wayne Luk, Natasa Przulj
Department of Computing
Imperial College London
London SW7 2AZ, United Kingdom
Email: {brahim.betkaoui, d.thomas1, w.luk, n.przulj}@imperial.ac.uk

***Abstract*—In many application domains, data are represented using large graphs involving millions of vertices and edges. Graph analysis algorithms, such as finding short paths and isomorphic subgraphs, are largely dominated by memory latency. Large cluster-based computing platforms can process graphs efficiently if the graph data can be partitioned, and on a smaller scale partitioning can be used to allocate graphs to low-latency on-chip RAMs in reconfigurable devices. However, there are many graph classes, such as scale-free social networks, which lack the locality to make partitioning graph data an efficient solution to the latency problem and are far too large to fit in on-chip RAMs and caches. In this paper, we present a framework for reconfigurable hardware acceleration of these large-scale graph problems that are difficult to partition and require high-latency off-chip memory storage. Our reconfigurable architecture tolerates off-chip memory latency by using a memory crossbar that connects many parallel identical processing elements to shared off-chip memory, without a traditional cached memory hierarchy. Quantitative comparison between the software and hardware performance of a graphlet counting case-study shows that our hardware implementation outperforms a quad-core software implementation by 10 times for large graphs. This speed-up includes all software and IO overhead required, and reduces execution time for this common bioinformatics algorithm from about 2 hours to just 12 minutes. These results demonstrate that our methodology for accelerating graph algorithms is a promising approach for efficient parallel graph processing.**

I. Introduction

Many real-world problems, such as various types of social networks and biological interactions, have been represented as large *graphs* or *networks* involving millions of vertices. In bioinformatics, for instance protein-protein interactions (PPIs) are commonly represented by graphs, where vertices represent proteins and edges represent physical interactions between the corresponding proteins [1]. As graph problems grow in size, efficient parallel graph processing becomes important as computational and memory requirements increase. Unfortunately, traditional software and hardware solutions that are used to parallelise mainstream parallel applications do not necessarily work well for large-scale graph problems. Graph problems have a number of properties that make them poorly matched to computational methods applied in mainstream parallel applications. In particular, the following properties of graph problems present significant challenges for efficient parallel processing of graph problems [2]:

- *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 computations is a challenging task due to lack of knowledge about the structure of the computations.
- *Unstructured problems*. Often, the data in graph problems are unstructured and highly irregular. This irregular structure of the graph data makes it difficult to partition the graph data to take advantage of small and fast on-chip memories, such as cache memories in cache-based microprocessors and on-chip RAMs in FPGAs.
- *Poor locality*. Data-driven computations coupled with irregular data structures results in low memory access locality. This often leads to suboptimal performance levels on conventional cache-based microprocessors, which rely on high spatial and temporal locality of memory accesses.
- *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 leads to execution times dominated by memory latency.

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 [3], [4], [5]. 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 the number of page misses in DRAM memories. Consequently, this low memory bandwidth incurs many pipeline stalls, resulting in little acceleration from the FPGA, and possibly even deceleration.

In this paper, we present a novel framework for reconfigurable hardware acceleration of graph algorithms. Our approach is evaluated on a high-performance reconfigurable computing platform, using a case study to compare it with a software implementation. Our key contributions are:

- A framework for reconfigurable hardware acceleration of large graph problems that require access to off-chip memory storage.
- A demonstration of our framework on a High-Performance Reconfigurable Computing (HPRC) system using a graphlet counting case-study from bioinformatics.
- A comparison of the software and hardware graphlet counting implementations, including all hardware overhead and IO costs, showing that up to 10 times performance speed-up can be achieved using our reconfigurable computing framework.

II. Framework for reconfigurable graph processors

As we have discussed earlier, the computational and memory access requirements of large-scale graph algorithms are significantly different from mainstream parallel applications, requiring new architectural solutions for efficient parallel graph processing. In this section we propose a framework for reconfigurable acceleration of large graph problems. Due to the wide variety of graph algorithms, we limit ourselves to a large class of common but demanding graph problems. In particular, we are interested in graphs problems with the following properties:

- *Large-scale sparse graphs* involving millions of vertices and edges. The graph data are too large to fit onto small and fast memories such as cache memory in commodity microprocessors and on-chip RAMs in FPGAs.
- *Static graphs*. The graphs are static at run-time, and hence, can be treated as read-only objects. There are many algorithms that takes graph datasets as input and output statistics about the graph. Examples include network alignment and comparison used in bio-informatics.
- *Highly Parallel*. Generally, graph problems have parallelism in abundance, where a task is performed independently on all vertices of the graph. The shortest path algorithm is an example, where for a given graph the shortest path problem for each vertex can be solved in parallel as an independent task.
- *Simple compute operations*. In addition to memory access requests, the compute operations performed by graph algorithms (e.g. integer addition, logical operations, etc.) are relatively simple compared to more complex operations such as floating-point computations.

Algorithm 1 shows a general template for the graph algorithms targeted by our framework. In terms of algorithm coding, this property translates into a loop that iterates through all the vertices in the graph. Each iteration can be performed as a separate kernel. The outer-loop (line #2) represents the coarse-grained parallelism required for our framework, while further fine-grained parallelism may be available within the graph kernel itself (line #2).

Algorithm 1 graph algorithm template

```
1: INPUT: a graph $G(V, E)$
2: for each vertex $v$ of $G$ in do
3:     {perform a graph kernel}
4: end for
5: OUTPUT: statistical data of $G(V, E)$
```

A. Hardware Architecture

Before we discuss the reconfigurable architecture, we first define the required characteristics of the target HPRC systems:

1) a high-bandwidth network between the FPGA devices and memory banks.
2) allow efficient word-level memory accesses.
3) multiple memory banks that can be accessed concurrently.

The hardware architecture used in our framework is illustrated in Figure 1. The architecture consist of three different components:

1) *Graph Processing Elements (GPEs)*. A collection of replicated and parallel processing elements that are application-specific. Each GPE can independently execute a graph kernel (see Algorithm 1, line #3).
2) *The memory interconnect network*. This links the GPEs to off-chip memory. In its basic form, it provides a point-to-point connection between each GPE and all memory banks through a memory crossbar. Through customisation, the memory interconnect can be optimised to improve the overall performance of the GPEs and memory bandwidth. For example, atomic counters can be implemented at the memory interconnect level instead of within the GPEs. Dynamic reordering of memory requests is another optimisation to improve the effective memory bandwidth.
3) *The run-time management unit*. This unit provides task assignment to the GPEs, as well as interfacing the FPGA-based coprocessor with the host processor. This unit can also provide dynamic load balancing, which is very important for a large class of graph problems such as scale-free networks.

Typically, mapping an algorithm onto a custom hardware accelerator requires extracting parallelism from 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, as we highlighted in section I, 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 using pipelining techniques, we aim to tolerate off-chip memory latency. In particular, we incorporate the following architectural design features into our solution to achieve efficient parallel processing of large graph problems:

- *High parallelism*. This is achieved by having a large number of GPEs operating 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.

- *Custom processing element*. Designing application-specific GPEs will result in efficient utilisation of hardware resources in contrast to general-purpose microprocessors. Given that operations performed in graph algorithms are simple compute operations that map to relatively simple hardware implementations, high-level synthesis tools should be able to generate efficient implementations of a GPEs while achieving high parallelism by replicating GPE cores.
- *Tolerating memory latency*. Instead of using cache memories to hide memory latency, we tolerate memory latency by connecting the GPEs to a shared memory system via a memory interconnect. Given a large number of parallel GPEs, multiple concurrent memory requests can be issued to parallel memory banks in the shared memory system, leading to superior memory access performance.
- *Decoupling access and execution units*. This will benefit the hardware synthesis process while improve the productivity of the framework user. For example, a GPE (the execution unit) can be generated using a high-level language or a domain-specific language, while the memory interconnect network (the access unit) can be obtained from a library of pre-compiled hand-crafted hardware components.

B. Design Flow

Figure 2 shows a high-level diagram of the design flow of the framework, from input specification through to hardware generation. A completely automated implementation of the proposed framework is currently under development. So for this paper, we provide the performance results of a manual execution of the design flow. Automating the flow used in

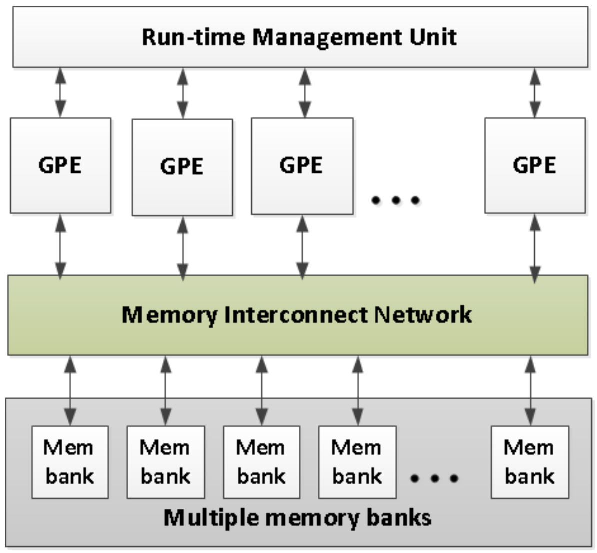

Fig. 1. System architecture for reconfigurable graph processing

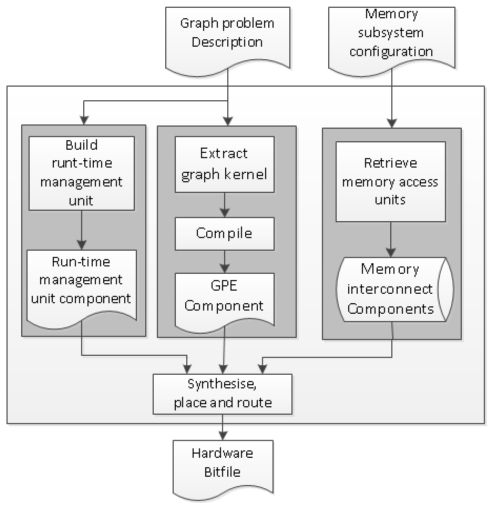
Graph problem Description
Memory subsystem configuration
Build runt-time management unit
Extract graph kernel
Retrieve memory access units
Compile
Run-time management unit component
GPE Component
Memory interconnect Components
Synthesise, place and route
Hardware Bitfile

Fig. 2. High-level design flow from input specification to hardware bitfile generation. The dotted box shows the design tools under development.

this paper should mainly be an engineering problem, requiring only existing tools and compiler technology; however, further important optimisations such as auto-tuning and providing graph algorithm-specific memory units will require additional research.

The input specification consists of the graph problem description, as well as information about the configuration of the platform to use. After reading the input specifications, the design flow can then proceed with three different processes in parallel: (1) extracting the graph kernel and synthesising it onto a GPE hardware component; (2) creating or selecting a pre-synthesised memory interconnect component that will connect the GPEs to the memory sub-system; (3) assembling the run-time management unit component that will schedule tasks for execution on the GPEs, as well as connect the GPEs with the platform interface. The three components are then combined and compiled to produce the final hardware bit-file and associated meta-data. At run-time the software running on a host processor can load the bit-file onto an FPGA-based coprocessor, and then execute the graph kernel on the FPGAs.

III. Case Study : Graphlet Counting Algorithm

Our proposed framework is evaluated in this section using a case-study that involves the acceleration of a graph algorithm, called the *graphlet counting algorithm*. For this paper, we report results on one graph algorithm only. Therefore, we do not claim this to be a thorough experimental study. Rather this paper serves as a case study for the applicability of our reconfigurable architecture to unstructured graphs, and as an introduction to a design framework for graph processors.

A. Algorithm Description

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 of a network [6]. Figure 3 shows 3-, 4-, and 5-node connected graplets. Enumerating all the graphlets in

Algorithm 2 The graphlet counting algorithm for k-node graphlets, with $k \in 3, 4, 5$

```
1:  for all nodes $a$ of $G$ do
2:      for all adjacent nodes $b$ of $a$ do
3:          for all adjacent nodes $c$ of $b$ do
4:              {*finding 3-node graphlets*}
5:              for all adjacent nodes $d$ of $c$ do
6:                  {*finding 4-node graphlets*}
7:                  for all adjacent nodes $e$ of $d$ do
8:                      {*finding 5-node graphlets*}
9:                  end for
10:             end for
11:         end for
12:     end for
13: end for
```

network has proved to be extremely useful in many network analysis algorithms such as GRAph ALigner (GRAAL) [7], and graphlet degree signatures (GDS) [8] but is also a computational bottleneck. The pseudo code of the graphlet counting algorithm is shown in the code snippet Algorithm 1.

The basic operations of this algorithm can be described as follows. For each 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 3) 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 $G2$); otherwise, we have a path (graphlet $G1$). To enumerate 4-node graphlets, all adjacent nodes d to node c are listed (line #11). Similarly, using the adjacency tests, we can deduce the different 4-node and 5-node graphlets in the subsequent inner for loops.

In terms of computational complexity, counting all graphlets in a graph G has a time complexity of $O(|V|^5)$. 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.

B. Graph data representation

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 our work, graphs are represented in a compact adjacency list form, and the adjacency lists of all vertices are packed into a single array as illustrated in Figure 4. 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 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 $i-1$. In addition to adjacency lists, adjacency matrices are often used to perform adjacency tests between the vertices of a graph. We use an adjacency matrix that requires $|V|^2$ bits of memory storage, which limits the size of the graph that we can process, while wasting a significant amount of bits since only a small portion of these bits is needed in sparse graphs. In the future, we will adopt more efficient storage schemes of the adjacency matrix such as using a hardware-accelerated hash-table.

C. Design Considerations

In section I, we outlined the main challenges in parallel graph processing. 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: typically only two or three iterations will occur, but occasionally 10 or 100 iterations will be needed. 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.

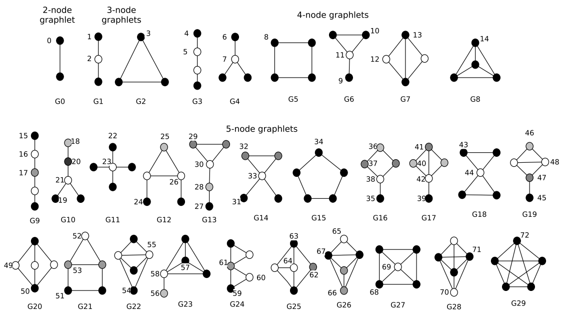

Fig. 3. 3-, 4- and 5-node connected graphlets

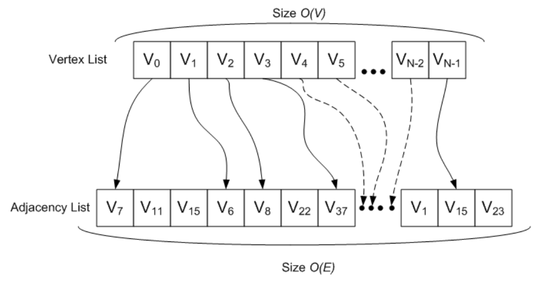

Fig. 4. Graph representation with vertex list pointing to adjacent vertex list

- *Synchronisation issues*: In the graphlet counting algorithm, we require 72 counters per vertex to enumerate all the 3-, 4-, 5-node graphlets (see Figure 3) for a given graph. These counters must be stored in off-chip memory for large graphs. A counter may be incremented by two or more processing elements simultaneously requiring a synchronisation mechanism. 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 2), and a serial implementation of all the inner loops (lines #2,3,5, and 7 in Algorithm 2). 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 memory 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, an atomic increment module can be implemented within the switching interconnect network to prevent read-after-write (RAW) hazards that may be caused by two GPEs trying to update the same counter. Using the atomic increment operations to update counters will not only prevent RAW hazards, but it will also improve the performance too, as an atomic increment operation will avoid GPE stalls while waiting for counter data to be read from memory. In addition moving the incrementation unit out of GPEs to the memory interconnect network allows GPEs to share this unit, and hence improve the overall device utilisation.

D. 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 component, and the run-time management unit. We explained in Section II-B how these three components are generated by the framework. A completely automated implementation of the proposed framework is under development, and so we generate hardware through manual execution of the framework. When designing the three components, we avoid any sophisticated optimisations to allow for the characterisation of the performance that can be expected from the final automated version of the framework.

1) Graph Processing Elements: . The graph processing element implements the graphlet counting kernel. It consists of a control unit, address function units, address and data registers, scalar functional units, and ROM tables to store hashing tables. The control unit includes a single finite-state machine (FSM) that implements serially all the inner loops in the graphlet counting algorithm. This FSM can be generated automatically by a high-level synthesis tool such as Handel-C.

The address function units include multipliers and adders with custom bit-width to compute memory addresses. In the case of the graphlet counters (72 counter per vertex), we combine logical shifts with addition instead of multiplication (since 72 = 64 + 8 or $72 = 2^6 + 2^3$). The scalar functional units include adders and logical comparison operators that are customised for data bit-width, and adders with constant operands such as 1-unit increment operations performed on graphlet counters. Again, all these optimisations use existing static analysis techniques that can be captured by a high-level synthesis tool.

The bit-width of both data and address registers is customised for the data that will be handled in these registers. For example, if all memory accesses are 8-byte long, than we can ignore the 3 least significant bits from the address registers and increment the address register by 1 instead of 8. Finally, the graphlet counting algorithm uses two hash tables in the inner-most loop to find 5-node graphlets; these hash tables are stored locally in the GPE using ROM memories.

2) 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.

3) Run-time Management Unit: .This unit manages the execution of the GPEs at run-time by using a static scheduler,

TABLE I
DEVICE UTILISATION ON A SINGLE VIRTEX5 LX330

Number of GPEs	Slices	BRAM
1 stand-alone GPE	2,919/207,360 (1%)	1/288 (1%)
16 GPE per AE	86,008/207,360 (41%)	60/288 (20%)
32 GPE per AE	124,822/207,360 (60%)	68/288 (23%)

that assigns an equal number of nodes for each GPE to process. This involves some simple control logic that can be easily generated by a high-level synthesis tool.

IV. PERFORMANCE EVALUATION

A. Software Implementation

We compare the performance of the hardware graphlet counting architecture with a software implementation. The source code of the graphlet counting algorithm was extracted from the GraphCrunch tool [9], an open source tool for biological network analysis developed at UC Irvine. We optimised the original code and used GCC compiler 4.1.2 with -O3 option flag to compile the code. The resulting executable is tested on an Intel Xeon E5420 CPU which has four cores, each core running at 2.5 GHz with access to 12MB of L2 cache and 16GB of DDR2-SDRAM.

B. Hardware Implementation

For the high performance reconfigurable computing system, we use the Convey HC-1 server [10] which has Virtex-5 FPGAs. The coprocessor board is housed in a 1U chassis that is fused to the top of another 1U chassis containing the host motherboard. The coprocessor board has four user-programmable Virtex-5 LX330s, which Convey calls application engines (AEs). Each AE is connected to eight independent memory controllers through a full 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), 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. The wrapper requires a fixed amount of resource overhead: 22% of BRAMS and about 10% of slices for each FPGA device.

Our hardware design is expressed in RTL using Verilog HDL and was compiled using ISE v13.1. The design runs at 150 MHz and resource utilisation of both stand-alone single GPE and in-system GPEs is shown in Table I.

C. Performance results

For our experimental work, we generate synthetic graphs using the LEDA library [11]. We restrict our experimental graph data to undirected and unlabelled sparse graphs based on the Erdos-Renyi random graph model. We vary graph size, $|V|$, and the average degree d. Table II compares the performance of the software and hardware implementations in terms of execution time. We use three configurations for the hardware implementation: 16, 64, and 128 GPEs. For the software implementation, we measure the performance of the quad-core CPU using both single and multiple CPU cores.

A first observation that can be made from the performance results, is the long execution times of the graphlet counting algorithm for large graph problems. 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. These long execution times, which can be as long as several hours, highlights the importance of accelerating such algorithms. Fortunately, the hardware implementation outperforms the software implementation for large graphs. As the size of the graph and edge density increases the speed-up of the hardware implementation over the software implementation increases too, reaching a performance speed-up of about 10 times for graphs with more than 100,000 vertices. In a computing grid, this means that 10 CPU servers can be replaced with one Convey HC-1 server to perform the same computing task, resulting in significant savings on both power consumption and rack space.

In terms of scalability of parallel resources, the HC-1 is more scalable that CPU as shown in Figure 5. HC-1's performance seems to scale almost linearly from 16 GPEs to 64 GPEs with an efficiency greater than 95%, which suggests that the memory subsystem has not been saturated, and hence adding more GPEs will likely increase the performance of the HC-1 implementation. As the number of GPEs is doubled from 64 to 128, the efficiency of HC-1 starts dropping to around 90% as the effects of memory bank contention become more significant for 128 GPEs. For the CPU, adding more CPU cores doesn't scale-up accordingly for large graphs as it achieves an efficiency of less than 80%. This non-linear performance scale-up is caused by the cache protocol overhead, and possibly an increased rate of last level cache misses.

V. PLANNED IMPROVEMENTS

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 as the number of GPEs is increased, suggesting we have yet to saturate all available memory bandwidth. 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

TABLE II
EXPERIMENTAL RESULTS, GRAPHLET COUNTING KERNEL ON CPU VS. CONVEY HC-1

number of vertices	number of edges	average degree	size in memory	CPU execution time		HC-1 speed-up over 4-core CPU		
				1 core	4 cores	16 GPE	64 GPE	128 GPE
10,000	40,000	∼8	18.17 MB	8.65s	2.57s	0.5x	1.2x	1.05x
10,000	50,000	∼10	18.32 MB.	20.95s	6.23s	0.5x	1.6x	2.5x
50,000	200,000	∼8	329.3 MB	1m2s	19.09s	0.8x	2.8x	4.7x
50,000	250,000	∼10	330 MB	2m31s	46.1s	0.8x	3.2x	5.5x
100,000	400,000	∼8	1.225 GB	1m36s	48.1s	1.0x	3.8x	6.7x
100,000	500,000	∼10	1.226 GB	6m8s	1m53s	1.1x	4.0x	7.4x
300,000	1.5 million	∼10	10.665 GB	17m23s	7m32s	1.4x	5.6x	10.4x
10,000	100,000	∼20	19.1 MB	5m15s	1m33s	0.6x	2.3x	4.2x
50,000	500,000	∼20	333.88 MB	36m39s	11m20s	0.9x	3.4x	6.4x
100,000	1 million	∼20	1.234 GB	1h30m26s	27m50s	1.1x	4.2x	8.0x
300,000	3 million	∼20	10.687 GB	6h1m46s	1h52m46s	1.5x	5.7x	10.8x

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.
- *Specialised cache memories*. Although in our current hardware architecture, there is no memory hierarchy due to poor memory locality characteristics of graph problems in general, we plan in the future to explore specialised caches to exploit locality that may be present in some algorithms. This can be effective where one portion of the dataset used by the graph algorithm will benefit from cache memories, whereas the other part of the dataset is not suitable for caching. For example it is often useful to cache node degrees and labels, but not information about edges.
- *Customising the memory interconnect*. 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 implement more operations than memory reads and writes, such as atomic increment operations, leading 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 framework allows such re-ordering to be particularly aggressive, due to the large number of GPEs generating requests - potentially individual GPE memory requests can be stalled for thousands of cycles, as long as it allows other GPEs to make better use of open memory pages.
- *Reducing memory footprint*. In our current design, the size of graphs is limited by the size of memory space required to store the adjacency matrix, which requires $|V|^2$ bits of memory storage. Using a hashing table can reduce greatly the memory footprint of the adjacency matrix data, given that most of real world graphs are rather sparse and rarely dense. This hashing can then be built into the memory interconnect, without increasing the computational load on the GPEs.

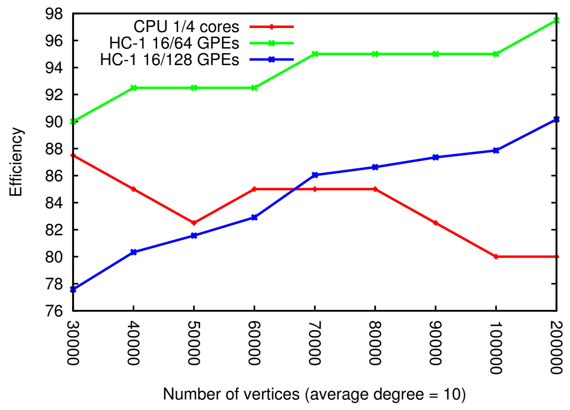

Fig. 5. Performance scalability as parallel resources are increased from 1 core to 4 cores for the CPU, and from 16 GPEs to 64 and 128 GPEs for HC-1. Efficiency is measured as the speed-up over a single core divided by the number of cores employed, so 100% efficiency is perfect scalability

VI. RELATED WORK

The importance of efficient processing of large graph problems has been increasing as datasets quickly grow past the capacity of current HPC systems. The authors in [2] surveys existing hardware architecture of HPC systems that are currently used to process graph problems. Distributed Memory computer clusters are a popular choice. Made mainly of commodity parts, distributed memory computers can offer good performance using inexpensive components [12]; However, they require the graph data to be partitioned which, as we discussed earlier, is not always a trivial task in unstructured graphs [13]. Such graphs are quite common in informatics applications, where the number of local vertices can be much smaller than the set of adjacent vertices.

Another class of parallel machines used for parallel graph processing are shared-memory computers. This class of parallel machines, where the global memory address space is accessible to all processors, includes cache-coherent parallel computers and massively multi-threaded machines (MMT). Cache-coherent parallel computers can process graphs faster than distributed memory systems, but have a memory hierarchy involving multiple cache memories, and so require cache coherence protocols for correct operation. Cache coherence protocols adds overhead which can degrade performance and introduce scalability issues. Moreover, the poor locality of memory accesses in graph problems renders memory cache ineffective, while the performance penalty due to cache coherence protocols cannot be avoided. Another sub-class of cache coherent parallel computers are MMT machines, such as the Cray MTA-2 [14]. MMT machines tolerate memory latency by providing hardware support for many concurrent threads that are able to issue multiple outstanding memory requests. A major drawback of MMT machines is that processors are custom, and not commodity, which means they are relatively more expensive and have slower clock rates than commodity processors.

Much previous work related to using FPGAs to solve graph problems has focused on using on-chip memory resources [3], [4] and [5]. However, many real world graphs are too large to fit into on-chip memory of FPGAs, requiring the use of off-chip memories such DRAM. Due to the significant difference in access times between on-chip memories and off-chip memories, many efficient on-chip FPGA solutions are not suitable for high-latency off-chip storage. In our work, we present a reconfigurable computing approach to accelerate the processing of large graph problems that require high-latency off-chip storage.

While there have been many applications in computational biology which have been successfully accelerated on FPGA-based compute engines [15], [16], [17], [18], there has been no previous work to which we can compare our results on the hardware acceleration of the graphlet counting algorithm.

VII. Conclusions and future work

In this paper we described a novel framework for reconfigurable hardware acceleration of graph algorithms. Using a case study, we have shown through experimental study that our approach is able to provide over an order of magnitude of performance speed-up over a software implementation running on a quad-core CPU, reducing execution time from several hours to just few minutes, while achieving better scalability of parallel resources. Overall, we have demonstrated that using our framework we are able to overcome the cost of memory latency, a dominant factor in the run-time of graph algorithms.

At the time of writing this paper, most of the steps in the design flow of the framework are performed manually. Our goal is to provide a proof concept that our approach can provide significant performance improvements. While this has been a short term goal, we ultimately aim to provide a fully-automated framework for hardware acceleration of graph algorithms. This includes exploring high-level synthesis tools to generate hardware. We will also explore more algorithms that will enable us to get further insight onto how to improve our framework. Currently, we only support unlabelled and unweighted graphs ; in the future we will extend our framework to support labelled and weighed graphs. We also plan on extending our testing graph data to include graph models other than the Erdos-Renyi model such as geometric and scale-free graph models.

Acknowledgment

The support of Imperial College London Research Excellence Award, the FP7 REFELCT (Rendering FPGAs for Multi-Core Embedded Computing) Project, the UK Engineering and Physical Sciences Research Council, HiPEAC, Convey Computer Corporation, and Xilinx, Inc. is gratefully acknowledged.

References

[1] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang *et al.*, "Topological structure analysis of the protein-protein interaction network in budding yeast," *Nucleic Acids Research*, vol. 31, no. 9, p. 2443, 2003.

[2] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[3] J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures," *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development & Computing*, pp. 225–236, 1996.

[4] 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.

[5] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient multi-context graph processors," *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pp. 915–924, 2002.

[6] 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.

[7] 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.

[8] 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.

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

[10] J. Bakos, "High-performance heterogeneous computing with the convey hc-1," *Computing in Science Engineering*, vol. 12, no. 6, pp. 80 –87, nov.-dec. 2010.

[11] S. Naher and K. Mehlhorn, "The LEDA platform of combinatorial and geometric computing," *Communications of the ACM*, 1995.

[12] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," 2005.

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

[14] W. Anderson, P. Briggs, C. Hellberg, D. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg, "Early experience with scientific programs on the cray mta-2," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003, p. 46.

[15] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *J. VLSI Signal Process. Syst.*, 2007.

[16] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," *Bioinformatics*, vol. 21, no. 16, pp. 3431–3432, 2005.

[17] D. Thomas, W. Luk, and M. Stumpf, "Reconfigurable hardware acceleration of canonical graph labelling," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 302–313, 2007.

[18] T. Mintz and J. Bakos, "A cluster-on-a-chip architecture for high-throughput phylogeny search," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2010.