# A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration

Brahim Betkaoui[1], Yu Wang [2], David B. Thomas[3], Wayne Luk[4]
[1,4] Department of Computing, Imperial College London
London, United Kingdom
[2] Department of Electronic Engineering
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University, Beijing, China
[3] Department of Electrical and Electronic Engineering, Imperial College London,
London, United Kingdom
[1,3,4]Email:{bb105,dt10,wl}@ic.ac.uk, [2]Email:yu-wang@mail.tsinghua.edu.cn

***Abstract*—In many application domains, data are represented using large graphs involving millions of vertices and billions of edges. Graph exploration algorithms, such as breadth-first search (BFS), are largely dominated by memory latency and are challenging to process efficiently. In this paper, we present a reconfigurable hardware methodology for efficient parallel processing of large-scale graph exploration problems. Our methodology is based on a reconfigurable hardware architecture which decouples computation and communication while keeping multiple memory requests in flight at any given time, taking advantage of the hardware capabilities of both FPGAs and the parallel memory subsystem. To validate our methodology, we provide a detailed design description of the Breadth-First Search algorithm on an FPGA-based high performance computing system. Using graph data based on the 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, and a throughput in excess of 2.5 billion traversed edges per second on RMAT graphs with 16 million vertices and over a billion edges. Using four Virtex-5 LX330 FPGAs based on 65nm technology and running at 75MHz, our BFS design achieves more than twice the speed of a 32-core Xeon X7560 based on 45nm technology and running at 2.26GHz.**

## I. Introduction

Many real-world problems, such as social networks and biological interactions, have been represented as large *graphs* or *networks* involving millions of vertices and billions of edges. For instance, in bioinformatics protein-protein interactions (PPIs) are commonly represented by graphs, where vertices represent proteins, and edges between proteins 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 graphs, due to a number of properties in graph problems [2]. Graph algorithms have data-driven computations dictated by the vertex and edge structure of the graph. Often, the data in graph problems are unstructured and highly irregular, requiring fine-grained random memory accesses. This often leads to poor spatial and temporal locality of memory accesses, and hence, suboptimal performance levels on conventional cache-based microprocessors. In addition, graph algorithms tend to explore the structure of the graph while performing a relatively small amount of computations, leading to execution times dominated by memory latency.

Previous work has shown that FPGA-based reconfigurable computing machines can achieve order of magnitude speedups compared to microprocessors for many important computing applications. However, one limitation of FPGAs that has prevented widespread usage is the requirement for regular or predictable memory access patterns 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 reconfigurable hardware methodology for efficient parallel graph exploration. Our approach is evaluated on a high-performance reconfigurable computing platform, using a case study to compare it with related work. This paper provides four key 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, taking advantage of the high bandwidth of a parallel memory subsystem.
- A detailed description of an efficient hardware solution to the well-known breadth-first search (BFS) problem as a case study, to provide a demonstration of our reconfigurable hardware methodology on a commercial FPGA-based high performance compute machine.

- An in-depth performance evaluation that considers different classes of graphs and analyses scalability, processing rate, and sensitivity to graph size.
- A performance comparison to related work in graph processing 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.

## II. Related work

The importance of efficient processing of large graph problems has been increasing as datasets quickly grow past the compute capacity of current HPC solutions. This has motivated a substantial amount of previous work that deals with the design and optimisation of graph exploration algorithm, in particular BFS designs, either for commodity processors [3], [4], [5], [6], [7], [8], or for dedicated hardware [9], [10], [11], [12].

Current state-of-the-art solutions for commodity processors are provided by Agarwal et al. for multi-core CPUs [6], and Hong et al. for CPU-GPU hybrid systems [4]. Using a 32-core CPU system, Agarwal et al.'s implementation [6] outperformed previous work on BFS implementations for large graph instances. Although, Hong et al's hybrid CPU-GPU implementation [4] provided comparable results to Agarwal et al's [6], the size of graph instances, including the average vertex degree, was limited by the GPU memory size. One way to allow for the GPU to process larger graph instances is to use multiple GPUs; however, it is not clear in [4] how the performance of multiple GPUs would scale in the presence of high costs of both GPU-GPU data transfer and global synchronisation across multiple GPUs.

Much previous work on using FPGAs to solve graph problems has used low-latency on-chip memory resources to store graph data [13], [9], [10]. 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. In our work, we present a reconfigurable computing approach to accelerate the processing of large-scale graph exploration problems that require high-latency off-chip storage.

Some recent publications have described successful parallelisation strategies of graph problems on reconfigurable hardware [11] and [12]. But, to the best of our knowledge, no previous FPGA work has tackled large-scale graph exploration algorithms which can compete with other high performance multi-core systems, in particular the recent work of Hong et al. [4] and Agarwal et al. [6].

## III. Background: The Parallel BFS Algorithm

Given a graph $G = (V, E)$ with a set $V$ of $n$ vertices and a set $E$ of $m$ directed edges, the BFS problem is to traverse the vertices of $G$ in breadth-first search order starting at 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, and its neighbours in BFS level 1, 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 exploration algorithms. For example, BFS can be used on a given graph to identify all of the connected components, to determine the graph diameter, or to perform a bipartiteness test [14]. In [15], BFS has been employed in brain network analysis of very sparse brain network data.

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 [4]. This approach, illustrated in Algorithm 1, is common in BFS implementations for high memory bandwidth machines such as GPUs [4], [5]. In algorithm 1, 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:** Level-synchronous read-based BFS

```
Input: $G(V, E)$, source vertex $v_s$
Output: Array distance[1..n] with distance[i]= minimum_distance($v_s$, $v_i$)
1  parallel foreach $v_i \in V$ do
2      $distance[i] \longleftarrow \infty$
3  $distance[s] \longleftarrow 0$, $bfs\_level \longleftarrow 0$
4  repeat
5      $done \longleftarrow$ true
6      parallel foreach $v_i \in V$ do
7          if distance[i] = bfs_level then
8              foreach $u_j$ adjacent to $v_i$ do
9                  if distance[j] = $\infty$ then
10                     $distance[j] \longleftarrow bfs\_level + 1$
11                     $done \longleftarrow$ false
12     $bfs\_level = bfs\_level + 1$
13 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 $O(n^2 + m)$ 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 [16]. Due to this property, the diameter of the graph is generally small, and hence, the number of BFS levels is much smaller than $n$. In addition, for such graphs, most vertices belong to one of a few critical BFS levels where the number of these vertices approaches $O(n)$. Since the execution time of the BFS algorithm is dominated by these critical levels, reading the whole $O(n)$-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 [4].

## IV. Reconfigurable Computing for Efficient Parallel Graph Exploration

As we have discussed in Section I, 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 exploration algorithms.

Algorithm 2 shows a general template for the graph 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 2:** Graph traversal algorithm template

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

### A. 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 using pipelining techniques, we aim to tolerate off-chip memory latency. 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**. Designing application-specific graph processing elements (GPEs) will result in efficient utilisation of hardware resources in contrast to a more general-purpose processing element. Given that operations performed in graph algorithms are simple compute operations (e.g. no floating-point operations) that map to relatively simple hardware implementations, high-level synthesis tools should be able to generate efficient implementations of an individual GPE, with spatial parallelism provided by replicating the GPE many times.

**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 2, line 2).

**3. Multiple concurrent memory requests**. Instead of using cache memories to hide memory latency, we adopt a *latency masking threads* technique [17]. The GPEs are directly connected to a shared off-chip memory system via a memory interconnect network 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 can be issued to parallel memory banks, leading to superior memory access performance. Having said that, we recognise that there are cases where an application-specific cache memory (scratch pad) can be used to improve the performance of the memory system.

**4. Trading speed for area**. Since the execution times of graph exploration 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 50MHz 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. 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.

### B. Reconfigurable hardware architecture template

The overall architecture of the reconfigurable computing solution, as illustrated in Figure 1, resembles a scalable, many-core style processor architecture, comprising a ***Run-time Management Unit*** **(RMU)**, 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 2, line 3). 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 memory banks. The RMU act 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

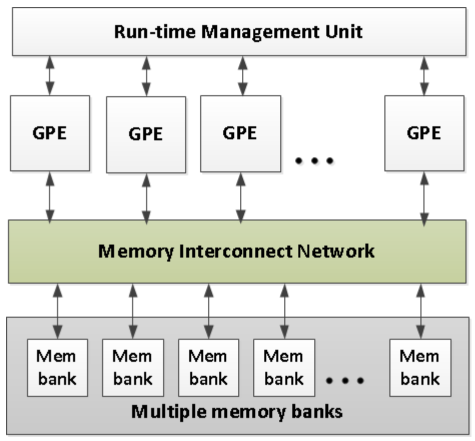

Fig. 1. Reconfigurable hardware architecture template for parallel graph exploration algorithms

of high performance reconfigurable systems with an FPGA-based coprocessor architecture. Optionally, each GPE can have a private local memory accessible only to itself, and/or have shared memories accessible to more than one GPE.

## V. Hardware design of parallel BFS

In this section, we describe how we parallelised the BFS algorithm using our reconfigurable hardware architecture template presented in Section IV-B. We chose to use a refined version of the Read-based BFS algorithm (Algorithm 1) which is proven to be suitable for: (1) platforms with high memory bandwidth, and (2) randomly-shaped real-world graphs governed by the small-world property [16] as discussed in Section III. As for graph representation, we used the popular CSR (Compressed Sparse Row) format which merges the adjacency lists of all vertices into a single $O(m)$-sized array, with the beginning location of each vertex's adjacency list stored in a separate $O(n)$-sized array. The BFS-level of each vertex is stored in a separate $O(n)$-sized array, the *distance* array.

We start by breaking the read-based BFS algorithm into two parts: one part running on the run-time management unit (Algorithm 3), and the other part on the GPEs (Algorithm 4).

### A. The RMU design

The RMU design (Algorithm 3) consists of three main steps:

**1. Initialisation of the GPEs**. The RMU 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 4) 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 send a termination signal to the RMU, indicating whether there had been any vertices marked for the next BFS level.

**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 3:** Read-based BFS algorithm running on the RMU

```
1  Partition set of vertices $V$ into disjoint sets $V_i$, with $|V_i| = \frac{|V|}{np}$
2  $bfs\_level \longleftarrow 0$
3  $distance[s] \longleftarrow 0$
4  **repeat**
5      $done \longleftarrow$ **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** $done$
```

### B. 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 (i) serialising execution and processing of data within the GPE, and (ii) parallelising access to off-chip memory. A serial implementation of the BFS kernel lead to an area-efficient design, and hence more GPEs can be instantiated on the FPGA. In addition, the inner loops in Algorithm 4 (lines 3, 4, 7, and 11) 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 2 presents a schematic overview of the GPE design for the BFS kernel (Algorithm 4). 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 the $v_i$ 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 4). For area-efficiency reasons, these registers are implemented using distributed RAM instead of Slice registers. Instead of issuing one memory request, and then waiting for response from memory, the GPE issues multiple non-blocking memory requests in an attempt to take advantage of the capabilities of the parallel

**Algorithm 4:** 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
foreach $v \in V_i$ do
    if (distance[i] == bfs_levels) then
        for (offset ← R[v]; offset < R[v+1]; offset += q) do
            foreach $i \in 0..q$ do
                $Nid[i] \leftarrow C[offset + i]$
            end
            foreach $i \in 0..q$ do
                $u \leftarrow Nid[i]$
                $bitmask[i] \leftarrow (distance[u] == \infty)$
            end
            foreach $i \in 0..q$ do
                $u \leftarrow Nid[i]$
                if $bitmask[i] == 1$ then
                    $distance[u] \leftarrow bfs\_level + 1$
                    $done \leftarrow 0$
                end
            end
        end
    end
end
```

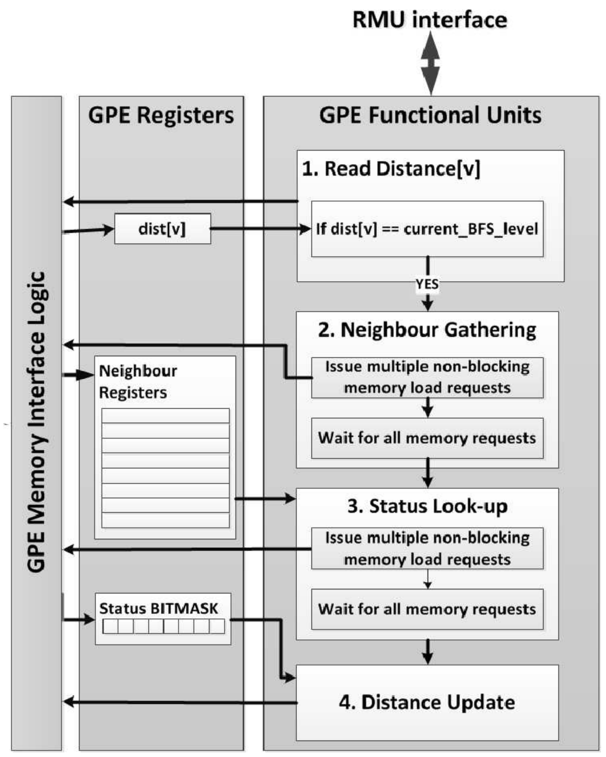
RMU interface
GPE Memory Interface Logic
GPE Registers
GPE Functional Units
1. Read Distance[v]
dist[v]
If dist[v] == current_BFS_level
YES
2. Neighbour Gathering
Issue multiple non-blocking memory load requests
Wait for all memory requests
Neighbour Registers
3. Status Look-up
Issue multiple non-blocking memory load requests
Wait for all memory requests
Status BITMASK
4. Distance Update

Fig. 2. *Graph Processing Element (GPE)* design for the BFS kernel (Algorithm 4)

memory subsystem. Assuming that the requests are destined for different memory banks, the high off-chip memory latency of single memory request is amortised over multiple memory requests as they get serviced simultaneously. The maximum number of non-blocking memory requests is bounded by $q$, the number of the *Neighbour* 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 $q$ (lines 4-5). 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 7-9). 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 11-16). Vertices marked in previous step will have their distance value updated to the current BFS level plus one (lines 13-14). By updating a vertex's distance, the vertex is also marked for the next BFS iteration.

## VI. Methodology

This section provides details of our experiments. For the graph data, we used two different classes of graphs: uniformly random graphs, and scale-free graphs (R-MAT). 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 *GT-graph* [18], a synthetic graph generator suite. R-MAT graphs are characterised by their skewed degree distribution and fractal community. The scale-free graphs are generated using the Graph500 benchmark suite [19] based on the Recursive-Matrix (R-MAT) graph model. For the parameters of the R-MAT graph, we used 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 $O(n)$. As in previous related work [4], [6], our BFS performance is reported as the number traversed edges per second, which is computed by dividing the actual number of directed edges over the BFS execution time.

Table I provides the configuration details of the machines used in our experiments and in the previous work [4] and [6]. For the high performance reconfigurable computing system, we use the Convey HC-1 server [20] which has four user-programmable Virtex-5 FPGAs, and each FPGA is connected to a shared memory subsystem via a memory crossbar.

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

TABLE I
THE SPECIFICATION OF MACHINES USED OUR EXPERIMENTS AND RELATED WORK [4] AND [6]

| | SC10-EP[6] | SC10-EX[6] | PACT11-NEH[4] | Convey HC-1 |
|---|---|---|---|---|
| Core Architecture | Intel Nehalem | Intel Nehalem | Intel Nehalem | Xilinx Virtex-5 |
| Model No. | Xeon X5570 | Xeon X7560 | Xeon X5550 | XC5VLX330 |
| Lithography | 45 nm | 45 nm | 45 nm | 65 nm |
| Core frequency | 2.93 GHz | 2.26 GHz | 2.66 GHz | 75 MHz |
| Total Num. of Cores | 8 | 32 | 8 | - |
| Total Num. of FPGA devices | - | - | - | 4 |
| Total Num. of threads/PEs | 16 | 64 | 16 | 512 |
| Main Memory | 48 GB | 256 GB | 24 GB | 16 GB |
| Maximum memory bandwidth (theoretical) | 100 GB/ | 266 GB/s | 100 GB/s | 80 GB/s |

TABLE II
DEVICE UTILISATION ON A VIRTEX-5 LX330 DEVICE

| Num. of GPEs | Num. of Neighbour Reg | Slice LUTs | BRAM |
|---|---|---|---|
| 128 | 16 | 80% | 64% |
| 128 | 32 | 83% | 64% |

wrapper that allows the 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 II.

## VII. EXPERIMENTAL RESULTS

In this section, we validate the effectiveness of our reconfigurable computing methodology that we presented in Section IV-B. To begin we measure the performance of our BFS design on the Convey HC-1 machine (See Table I), 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. [6] and Hong et al. [4].

### A. Scalability

Figures 3 and 4 show the processing rate and scalability of our BFS design for both uniformly random graphs and RMAT graphs. The number of graph vertices is set to 16 million vertices with an average vertex degree of 32. The number of GPEs varies from 32 to 512 in our design. We define the efficiency as the ratio of speedup of $g$ GPEs over 32 GPE, divided by the linear or ideal speedup, $\frac{g}{32}$. In our current design we are able to fit up to 128 GPEs per Virtex5 LX330 device, so we used 2 and 4 FPGA devices for 256 GPEs and 512 GPEs respectively. For large uniformly random graphs, we observe that our design not only scales well on one FPGA device giving an efficiency of 92.5-95.5%, but also on multiple FPGA devices as we are able to reach efficiency rates over 98% and over 94% for 2 and 4 FPGA devices respectively. Similarly, for large RMAT graphs, we see a similar efficiency pattern as to that of uniformly random graphs, albeit with slightly lower efficiency rates: 77-82% for a single FPGA device, and 94% and 83% for 2 and 4 FPGA devices respectively.

### B. Graph Size Sensitivity

Figures 5 & 6 show the average processing rate obtained on four Virtex-5 LX330 FPGAs for uniformly random graphs and RMAT graphs respectively. 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 uniformly random 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 [6], tends to decrease as the graph size is scaled up due to the increased rate of last-level cache misses. Similarly,

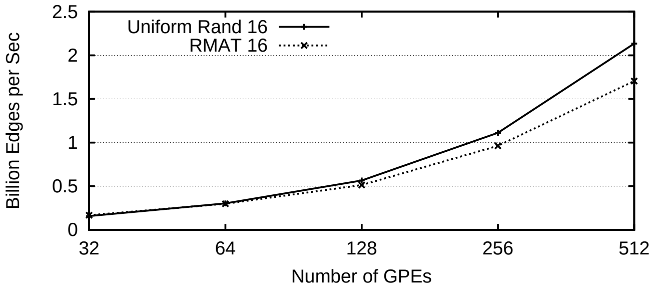

Fig. 3. FPGA-accelerated BFS performance: processing rate.

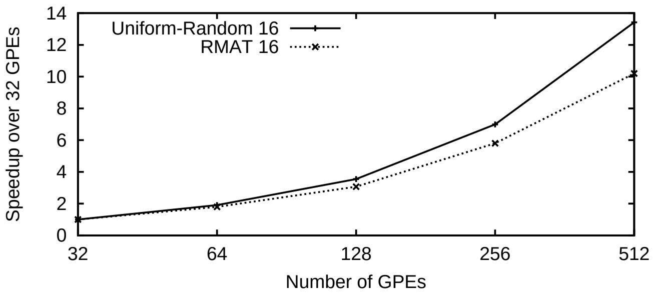

Fig. 4. FPGA-accelerated BFS performance: speedup over 32 GPEs.

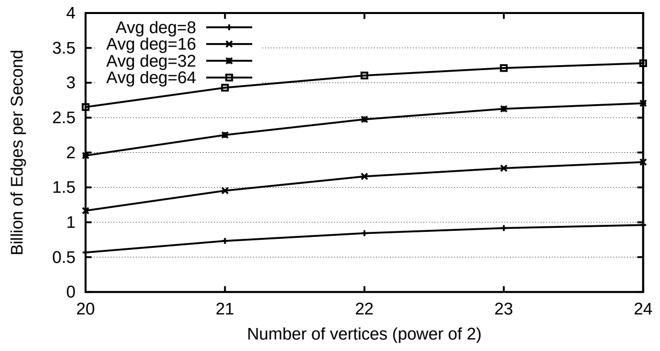

Fig. 5. FPGA-accelerated BFS design: performance scaling with respect to graph size in terms of both vertex count and average vertex degree (Avg deg) for uniformly random graphs.

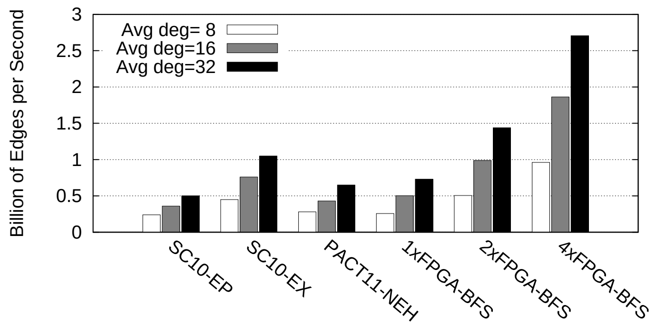

Fig. 7. Performance comparison of BFS execution on various machines using uniformly random graph instances with 16 million vertices and an average vertex degree (Avg deg) of 8, 16, and 32.

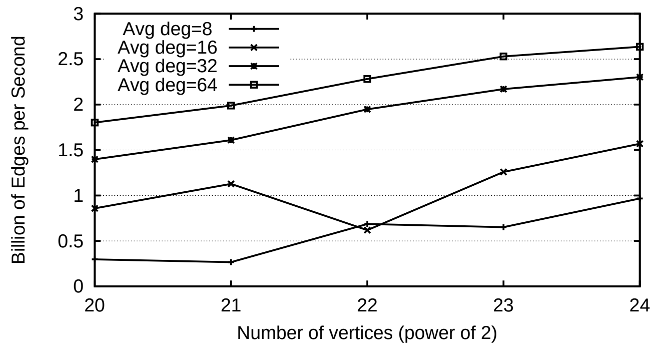

Fig. 6. FPGA-accelerated BFS design: performance scaling with respect to graph size in terms of both vertex count and average vertex degree (Avg deg) for RMAT graphs.

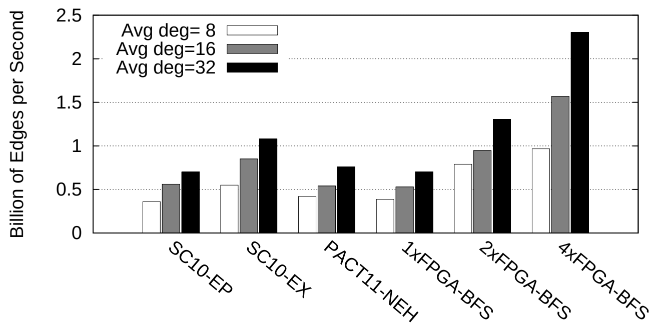

Fig. 8. Performance comparison of BFS execution on various machines using RMAT graph instances with 16 million vertices and an average vertex degree (Avg deg) of 8, 16, and 32.

for R-MAT graphs, the performance increases as the graph grows in size in general. However, there are few instances of RMAT graphs where performance decreases as the graph size is increased. This may be caused by workload imbalance that can occur in RMAT graphs due to their skewed degree distribution.

### C. *Performance Comparison*

Figures 7 & 8 compare our BFS performance to performance values in previous work of Agarwal et al. [6] and Hong et al. [4], which are reported to be the fastest BFS implementations in comparison with other related work [7], [8], [3], and [21]. The last three sets of bars on the left of each figure represent the measured performance of our hardware implementation of 128 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 75 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 RMAT graphs. For uniformly random graphs, our FPGA design outperforms PACT11-NEH by a factor of 1.4x. With Four Virtex-5 FPGAs, we are able to achieve a speedup of 5.4x and 3.2x for large instances of uniformly random graphs and RMAT 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 1.33x for uniformly random graphs, and 1.2x for RMAT graphs. Using all four FPGAs, our BFS design performed about 2.5x and 2.13x faster than SC10-EX for uniformly random and RMAT graphs with 16 million vertices and 512 million edges.

From these comparison results, we can say that our FPGA design outperforms the multi-core CPU implementations as it is able to achieve higher parallelism through 512 custom GPEs compared to 16 and 64 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 1, line 10). By issuing a large number of concurrent memory requests (up to 512*32=16384 requests), our FPGA design can cope better with irregular memory accesses. On the other hand, the CPU-based systems try to hide memory latency using cache memories, which is ineffective for random and irregular memory access patterns.

## VIII. Conclusion & Future Work

In this paper, we propose a reconfigurable computing solution for efficient parallel graph exploration algorithms. Using a common graph exploration algorithm, namely the BFS algorithm, we have shown through experimental study that our approach is able to outperform the state-of-the-art BFS implementations in recent high performance computing literature by more than 2 times for graphs with millions of vertices and edges. Future work include investigating ways to improve the performance by making use of on-chip RAM resources in FPGAs to reduce off-chip memory traffic, dynamic task scheduling to improve workload balance, and exploring other graph algorithms such as *ST-connectivity*, and *All Pairs Shortest Path* algorithms. Eventually, we aim integrate our reconfigurable computing solution onto a heterogeneous accelerator platform that employs graph algorithms for mutli-subject voxel-based brain network analysis [15].

## Acknowledgment

This work was supported by the Royal Academy of Engineering, National Science and Technology Major Project (2010ZX01030-001), National Natural Science Foundation of China (60870001, 61171002), UK EPSRC, Microsoft, IBM, AMD, Convey and Xilinx. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreements number 248976, 257906 and 287804.

## 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] 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’05:*, 2005, p. 25.

[4] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient parallel graph exploration on multi-core CPU and GPU.” in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.

[5] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA.” in *International Conference on High Performance Computing*, vol. 4873. Springer, 2007, pp. 197–208.

[6] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, “Scalable graph exploration on multicore processors,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC’10. IEEE Computer Society, 2010, pp. 1–11.

[7] D. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and ST-connectivity on the Cray MTA-2,” in *International Conference on Parallel Processing, 2006*, Aug. 2006, pp. 523 –530.

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

[9] 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.

[10] O. Mencer, Z. Huang, and L. Huelsbergen, “HAGAR: Efficient multi-context graph processors,” in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, ser. FPL ’02. Springer-Verlag, 2002, pp. 915–924.

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

[12] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. Knight, and A. DeHon, “GraphStep: A system architecture for sparse-graph algorithms,” in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2006.*, April 2006, pp. 143 –151.

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

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

[15] 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 *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2011*, Nov 2011, pp. 339 –344.

[16] D. J. Watts and S. H. Strogatz, “Collective dynamics of ’small-world’ networks.” *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[17] R. J. Halstead, J. Villarreal, and W. Najjar, “Exploring irregular memory accesses on FPGAs,” in *Proceedings of the first workshop on Irregular applications: architectures and algorithm*, ser. IAAA ’11. ACM, 2011, pp. 31–34.

[18] 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

[19] “The Graph 500 List,” 2010. [Online]. Available: http://www.graph500.org/index.html

[20] 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.

[21] Y. Xia and V. K. Prasanna, “Topologically adaptive parallel breadth-first search on multicore processors,” in *21st International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov 2009.