PARALLEL FPGA-BASED ALL PAIRS SHORTEST PATHS FOR SPARSE NETWORKS: A HUMAN BRAIN CONNECTOME CASE STUDY

Brahim Betkaoui[1], *Yu Wang*[2], *David B. Thomas*[3], *Wayne Luk*[4]

[1,4]Department of Computing, Imperial College London
[2]Department of Electronic Engineering, Tsinghua University, China
3Department of Electrical & Electronic Engineering, Imperial College London
[1,3,4]email: {bb105,dt10,wl}@ic.ac.uk
[2]email: yu-wang@mail.tsinghua.edu.cn

ABSTRACT

This paper proposes a highly parallel and scalable reconfigurable design for the All-Pairs Shortest-Paths (APSP) algorithm for very sparse networks. Our work is motivated by a computationally intensive bioinformatics application that employs this memory-latency bound algorithm. The proposed design methodology takes advantage of distributed on-chip memory resources of modern FPGAs to reduce accesses to high-latency off-chip memories. We develop design optimisations that yield different FPGA configurations which are selected at run time based on the input graph data. Using human brain network data, we are able to achieve performance results superior to those from multi-core CPU and GPU, while attaining linear scaling over the number of processors introduced. Our FPGA-based APSP design is over 10 times faster than a quad-core CPU implementation and 2-5 times faster than an AMD Cypress GPU implementation.

1. INTRODUCTION

Many real-world problems, such as social networks and biological interactions, have been represented as *graphs* or *networks* involving millions of vertices and edges. For instance, in bioinformatics the human brain has been modelled as a network based on *blood oxygen level dependent* signals. These human brain networks are analysed using graph theory algorithms [1]. However, one of the human Brain Network Analysis (BNA) limitations is the great computational complexity. Many graph algorithms, such as APSP, become intolerably time-consuming when the network becomes too large. Hence, a solution with a strong computational capability will greatly benefit the BNA research.

Previous work has shown that FPGA-based reconfigurable computing machines can achieve order of magnitude speedups compared to microprocessors for many important applications. Application-specific designs lead to efficient utilisation of hardware resources on FPGAs. In this paper, we leverage the benefits of FPGAs for the Breadth-First Search algorithm used to solve the APSP problem in a directed graph with unweighted edges. Our key contributions are:

- A parallelisation methodology that we have adopted to design a parallel and scalable FPGA-based solver for the APSP problem for sparse networks (Section 4).
- 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).
- Design optimisations that yield different FPGA bitstreams which are selected at run-time based on the input graph data (Section 6).
- An in-depth performance evaluation that analyses performance scalability, and the effects of different design optimisations using real-word data (Section 7).
- 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 7).

2. BACKGROUND

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 [2], which is based on single-source shortest path algorithms such as Dijkstra algorithm [3] and Bellman-Ford algorithm [4]. 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

978-1-4673-2256-0/12/$31.00

graphs. Another class is the Floyd-Warshall (FW) algorithm [5], which has $O(N^3)$ 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 1).

Algorithm 1: 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)
3 end
```

The BFS problem is defined as follows. 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.

Algorithm 2 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.

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

3. 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 [6, 7, 8], or for dedicated hardware [9, 10, 11, 12]. Matsumoto et al. [6] 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 graph problems has used low-latency on-chip memory resources to store graph data [13, 12]. 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.

Some recent publications have described successful parallel implementations of graph problems on reconfigurable hardware [10] and [11]. To the best of our knowledge, no previous FPGA work has tackled the APSP problem for sparse graphs with unweighted edges. Bondhugula et al. [9] proposed a parallel FPGA design to accelerate the FW algorithm which is more suitable for dense networks, and is proven to be inefficient for very sparse networks. We consider our work complementary to Bondhugula et al.'s work.

4. PARALLELISING APSP-BFS FOR FPGAS

Graph algorithms have a low computation to access ratio [14], 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 1) in Section 2 can be parallelised using one of the following three methods:

1. Serial APSP, Parallel BFS: The outer-loop in algorithm 1 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. 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 increasing

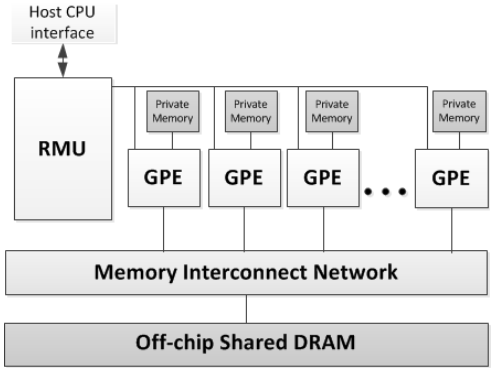

Fig. 1. Reconfigurable hardware architecture template for parallel graph exploration algorithms

the number of PEs will increase access contention overhead.

2. Parallel APSP, Serial BFS. The outer-loop in Algorithm 1 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. 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. Parallel APSP, Parallel BFS: This is the general case where the outer-loop in Algorithm 1 executes in parallel a number of parallel BFS kernels. This 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.

In this work, we adopt the second method for our FPGA-based APSP-BFS, and leave evaluation of the third method for future work. The overall architecture of the FPGA-based APSP design, as illustrated in Figure 1, resembles a scalable many-core style processor architecture, comprising a Run-time Management Unit (RMU), multiple Graph processing elements, and a memory interconnect network. The RMU acts as a control processor that manages the operation of the GPEs, including initialisation, task assignment, and synchronisation of the GPEs. The GPEs are a collection of replicated and parallel processing elements that are application-specific. Each GPE can independently execute a BFS on given source node. Each GPE can have a private local memory accessible only to itself. The interconnect memory network links the GPEs to an off-chip shared-memory subsystem.

5. PARALLEL FPGA DESIGN FOR APSP-BFS

In this section, we describe how we parallelised the APSP-BFS algorithm using our reconfigurable hardware architecture template presented in Section 4. In our approach, we chose to parallelise APSP-BFS algorithm using the sequential version of BFS (Algorithm 2). 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 n-sized array. For each BFS, we require an n-sized array, the *distance* array, to store the BFS level of each vertex, and hence we require an n^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 run-time management unit (Algorithm 1, line 1), and the other part on the GPEs (Algorithm 2). The RMU 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 RMU issues a termination signal to the host CPU to indicate that the APSP routine completed execution.

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) parallelising 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 (Algorithm 3, line 1). 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 n 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 n bits, which can be stored entirely in on-chip memories. In contrast, using a standard FIFO queue requires $log_2(n)$ bits per vertex, and approaches $n \times log_2(n)$ 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.

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 3), the visitation status of vertices requires random memory accesses (Algorithm 3, 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

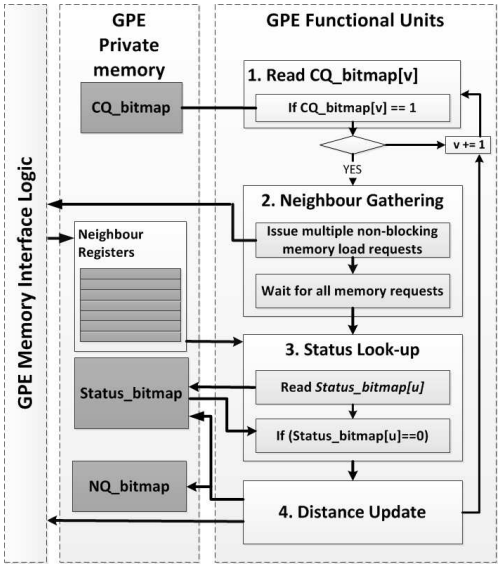

Fig. 2. *Graph Processing Element (GPE)* design for BFS.

in an on-chip bitmap, which can be accessed in one clock cycle, saving on up to tens of clock cycles.

Finally, parallelising memory accesses is achieved by enabling the GPE to 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. 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 parallel 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 2 presents a schematic overview of the GPE design for the BFS kernel (Algorithm 3) that incorporates the design choices discussed above. In Algorithm 3, 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:

1. Read *CQ_bitmap*. This unit checks whether a v_i belongs to the current BFS level by reading *CQ_bitmap* (line 7). If $CQ\_bitmap[v_i]$ is set, its neighbours are explored in the current iteration (steps 2-4).

2. Neighbour gathering. The neighbours of v_i are retrieved from memory in q-sized batches using multiple non-blocking memory requests (lines 9-12). The retrieved neighbouring vertices are stored in local registers (NID registers). For area-efficiency reasons, these registers are implemented using distributed RAM instead of Slice registers.

3. Status look-up. The visitation status of the gathered neighbours is checked (line 15) by reading *Status_bitmap*.

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

Algorithm 3: 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; distance[s] ⟵ bfs_level ;
2  CQ_bitmap ← NQ_bitmap ← Status_bitmap ← 0 ;
3  CQ_bitmap[s] ← Status_bitmap[s] ← 1;
4  repeat
5      done ← 1 ;
6      foreach v ∈ 1..n do
           // Step 1:  read CQ_bitmap[v]
7          if (CQ_bitmap[v]) then
8              CQ_bitmap[v] ← 0;
9              for (offset ← R[v]; offset < R[v+1]; offset += q) do
                   // Step 2:  Neighbour gathering
10                 foreach i ∈ 1..q do
11                     NID[i] ← C[offset + i];
12                 end
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                     end
21                 end
22             end
23         end
24     end
25     bfs_level = bfs_level + 1;
26     Swap(CQ_bitmap, NQ_bitmap);
27 until (done) ;
```

6. DESIGN OPTIMISATIONS

6.1. 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 k/w neighbouring vertices per memory operation if we use w bits to represent neighbouring vertices. Figure 3 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.

6.2. Hybrid BFS kernel

Beamer et al [15] presented a hybrid approach to the BFS algorithm that combines the conventional top-down algorithm 2 and a bottom-up algorithm. This hybrid approach takes advantage of the small-world property of real-world graphs [16] 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 4 describes the bottom-up algorithm that replaces lines (6-24) in Algorithm 3, i.e. steps 1-4.

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 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.

6.3. 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 2) 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 used human brain networks [17] 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.

7. EXPERIMENTAL RESULTS

In this section, we validate the effectiveness of our parallel FPGA-based APSP design presented in Section 5. For the graph data, we extracted our data from a downloaded dataset

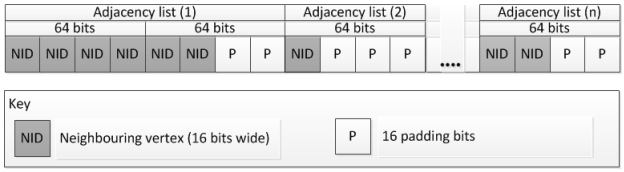

Fig. 3. The encoding format of the adjacency list

Algorithm 4: 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              end
7              foreach i ∈ 1..q do
8                  u ← NID[i];
9                  if CQ_bitmap[u] then
10                     distance[v] ← bfs_level + 1 ;
11                     Status_bitmap[v] ← 1;
12                     NQ_bitmap[v] ← 1;
13                     done ← 0;
14                     break;
15                 end
16             end
17         end
18     end
19 end
```

from a fMRI data sharing project, *1000 Functional Connectomes Project* [17]. The resulting graphs have 38368 vertices with varying sparsity values. This is the same dataset used in [18]. For the high performance reconfigurable computing system, we use the Convey HC-1 server [19] which has four Virtex-5 LX330 FPGAs which are connected to a shared memory subsystem. 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 [18] and [6] respectively. An AMD Phenom II X4 965 quad-core CPU running at 3.4 GHz is used in [18] for APSP-BFS, while an AMD Cypress GPU (Radeon HD 5870) is used in [6] for APSP-FW.

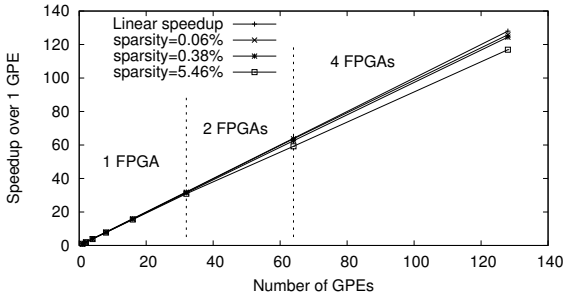

Fig. 4. FPGA design performance: speedup over 1 GPEs.

Figure 4 shows the scalability of our FPGA-based APSP design for graphs with different sparsity values. The number of vertices is set to 38368, and 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 used 2 and 4

FPGA devices for 64 GPEs and 128 GPEs respectively. For different graph sparsity 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 90% 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.

Table 1 compares the performance of our FPGA-based APSP design to that of a quad-core CPU [18], and a GPU [6]. Note that the GPU implementation executes the FW algorithm to solve the APSP problem, and hence the execution time stays the same when the sparsity 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 size and its sparsity. 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 6.3, an FPGA configuration is selected based on the input graph data at run-time.

The FPGA implementation of APSP outperforms both the quad-core CPU and the GPU for up to 5% graph sparsity. 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 graph sparsity values. Note that the GPU is faster than our FPGA design as the graph sparsity percentage increases beyond 5%. This presents an opportunity to build a heterogeneous system comprising CPUs, GPUs, and FPGAs to accelerate the APSP problem for different graph sparsity values.

Table 1. Performance comparison with CPU and GPU.

Sparsity	CPU [18]	GPU [6]	FPGA-1	FPGA-2
0.06%	39s	167s (0.2x)	**37s(1.1x)**	55s (0.7x)
0.13%	74s	167s (0.4x)	**45s(1.6x)**	76s (0.9x)
0.38%	191s	167s (1.1x)	**72s(2.7x)**	81s (2.5x)
1.39%	633s	167s (3.8x)	185s(3.4x)	**93s (6.7x)**
5.46%	2430s	167s (14.5x)	612s(3.9x)	**177s(13.7x)**

8. CONCLUSION AND FUTURE WORK

This paper proposes a parallel and scalable FPGA design for the All-Pairs Shortest Paths problem for sparse graphs with unweighted edges. 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 a multi-core CPU as well as a hybrid CPU-GPU system. 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.

Acknowledgments

The research leading to these results has received funding from European Union Seventh Framework Programme under grant agreement number 287804, 248976 and 257906. This work has been supported by National Science and Technology Major Project (2010ZX01030-001), IBM/Microsoft, NSF of China (61171002), Tsinghua University Initiative Scientific Research Program, the Royal Academy of Engineering, UK EPSRC, the HiPEAC NoE, Convey Computer Corporation and Xilinx.

9. REFERENCES

[1] Y. He *et al.*, "Uncovering intrinsic modular organization of spontaneous brain activity in humans," *PLoS ONE*, 4(4):e5226, 2009.

[2] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *JACM*, 24(1):1-13, 1977.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1:269-271, 1959.

[4] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, 16:87-90, 1958.

[5] R. W. Floyd, "Algorithm 97: Shortest path," *CACM*, 5(6):345, 1962.

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

[7] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU." *Parallel Computing*, 36(5-6):241-253, 2010.

[8] K. Matsumoto and S. G. Sedukhin, "A solution of the all-pairs shortest paths problem on the Cell Broadband Engine processor," *IEICE Transactions on Information nd Systems*, E92.D(6):1225-1231, 2009.

[9] U. Bondhugula *et al.*, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," in *IPDPS*, 2006.

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

[11] M. deLorimier *et al.*, "GraphStep: A system architecture for sparse-graph algorithms," in *FCCM*, 2006.

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

[13] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient multi-context graph processors," in *FPL*, 2002.

[14] A. Lumsdaine *et al.*, "Challenges in parallel graph processing." *Parallel Processing Letters*, 17(1):5-20, 2007.

[15] S. Beamer, K. Asanović, 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.

[16] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks." *Nature*, 393(6684):440-442, 1998.

[17] NITRC: the source of neuroimaging tools and resources. [Online]. Available: http://www.nitrc.org/projects/fcon_1000

[18] Y. Wang *et al.*, "A heterogeneous accelerator platform for multi-subject voxel-based brain network analysis," in *ICCAD*, 2011.

[19] J. D. Bakos, "High-performance heterogeneous computing with Convey HC-1," *Computing in Science and Engineering*, 12(6):80-87, 2010.