Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform

Conference Paper · September 2015

DOI: 10.1109/FPL.2015.7293939

Yaman Umuroglu
Norwegian University of Science and Technol...
9 PUBLICATIONS **18** CITATIONS

Magnus Jahre
Norwegian University of Science and Techno...
38 PUBLICATIONS **126** CITATIONS

Preprint, to appear in FPL 2015 (www.fpl2015.org)

Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform

Yaman Umuroglu, Donn Morrison, Magnus Jahre
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
Email: yamanu,donn.morrison,magnus.jahre@idi.ntnu.no

Abstract—**Large and sparse small-world graphs are ubiquitous across many scientific domains from bioinformatics to computer science. As these graphs grow in scale, traversal algorithms such as breadth-first search (BFS), fundamental to many graph processing applications and metrics, become more costly to compute. The cause is attributed to poor temporal and spatial locality due to the inherently irregular memory access patterns of these algorithms. A large body of research has targeted accelerating and parallelizing BFS on a variety of computing platforms, including hybrid CPU-GPU approaches for exploiting the small-world property. In the same spirit, we show how a single-die FPGA-CPU heterogeneous device can be used to leverage of the varying degree of parallelism in small-world graphs. Additionally, we demonstrate how dense rather than sparse treatment of the BFS frontier vector yields simpler memory access patterns for BFS, trading redundant computation for DRAM bandwidth utilization and faster graph exploration. On a range of synthetic small-world graphs, our hybrid approach performs 7.8x better than software-only and 2x better than accelerator-only implementations. We achieve an average traversal speed of 172 MTEPS (millions of traversed edges per second) on the ZedBoard platform, which is more than twice as effective as the best previously published FPGA BFS implementation in terms of traversals per bandwidth.**

I. Introduction

Graphs as data representations and algorithms that operate on graphs are ubiquitous throughout most scientific domains. Breadth-first search (BFS) is a key building block for exploring graphs, and is fundamental to a variety of graph metrics such as counting connected components, calculating graph diameter and radius [1]. Special cases of BFS such as the independent cascade model (ICS) are used to simulate the spread of information or disease across networks [2].

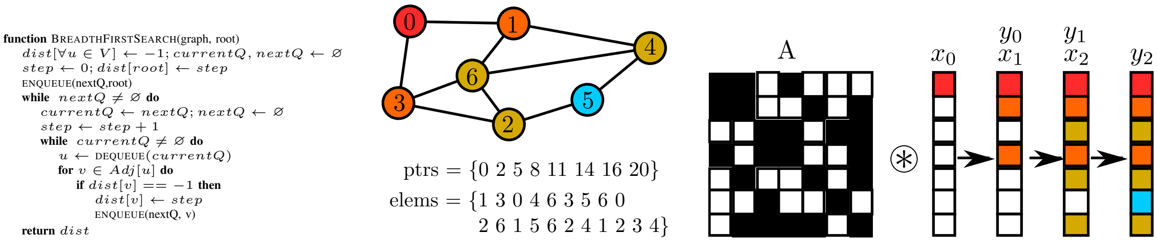
In order to meet the demand for analysis of ever-larger graphs brought by the Big Data trend, high-performance graph processing is of vital importance. Two key characteristics of BFS (and many other graph algorithms) are *irregular memory accesses* due to data-driven computations on the vertex and edge structure of the graph [3], and *low computation-to-memory ratio*. Thus, BFS performance is commonly memory bandwidth limited. These characteristics make accelerating and parallelizing BFS a major challenge, which has motivated a large body of research on different platforms (Section V).

With the increased focus in recent years on energy efficient computing systems, heterogeneous processing with reconfigurable logic and FPGAs is gaining popularity, including in datacenters [4]. Prior work by Betkaoui et al. [3] and Attia et al. [5] showed that reconfigurable logic for accelerating BFS on large graphs is performance-competitive with multi-core CPUs and GPGPUs. There are two main reasons that reconfigurable logic is suitable for energy efficient BFS. First, the memory architecture can be customized to effectively deal with the irregular memory access patterns. Additionally, BFS performance on large graphs is bound by accesses to high-latency external memory, which is a good fit for achieving high performance on FPGAs via ample parallelism and relatively low clock speeds.

However, this suitability is dependent on the *availability* of parallel work in BFS to offer high performance, which can lead to significant waste of execution resources. Large real-world graphs often have the small-world property, where the amount of parallelism available changes significantly during BFS (see Section II-B). Our work explores how this change in parallelism can be exploited in the context of a single-chip FPGA-CPU heterogeneous processor to offer high-performance BFS on large graphs. Through observations on the Boolean matrix-vector representation for BFS, we propose an FPGA BFS accelerator architecture with a stall-free datapath. We describe two architectural variants that treat the BFS frontier as *sparse* or *dense* to show how redundant computations can be traded for bandwidth and increase BFS performance in hybrid execution. Our experimental results on the ZedBoard with synthetic real-world networks indicate that this scheme can utilize up to 78% of the available DRAM bandwidth and achieve over twice as many traversals per unit bandwidth compared to previous work.

Specifically, our work makes the following contributions:

- A fast FPGA-CPU hybrid BFS architecture with high DRAM bandwidth utilization;
- An analysis of BFS memory request structure and bandwidth utilization for sparse and dense BFS frontier treatment;
- A stall-free BFS datapath using FPGA on-chip RAM to buffer the node visit status;
- A method for decoupling BFS level computation from the traversal to keep the node visit status data small.



```
function BREADTHFIRSTSEARCH(graph, root)
    $dist[\forall u \in V] \leftarrow -1; currentQ, nextQ \leftarrow \varnothing$
    $step \leftarrow 0; dist[root] \leftarrow step$
    ENQUEUE(nextQ,root)
    while $nextQ \neq \varnothing$ do
        $currentQ \leftarrow nextQ; nextQ \leftarrow \varnothing$
        $step \leftarrow step + 1$
        while $currentQ \neq \varnothing$ do
            $u \leftarrow$ DEQUEUE$(currentQ)$
            for $v \in Adj[u]$ do
                if $dist[v] == -1$ then
                    $dist[v] \leftarrow step$
                    ENQUEUE(nextQ, v)
    return $dist$
```

(a) Pseudocode

ptrs = {0 2 5 8 11 14 16 20}
elems = {1 3 0 4 6 3 5 6 0
2 6 1 5 6 2 4 1 2 3 4}

(b) Graph, BFS levels and CSC

(c) BFS operation on the Boolean semiring

Fig. 1: Three representations of the breadth-first search algorithm.

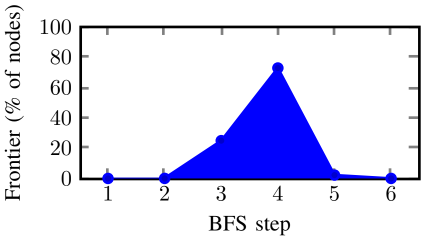


Fig. 2: A typical frontier profile for BFS on a small-world graph. Exposed parallelism increases with the frontier size, which approaches the total number of nodes in the graph for the intermediate levels (here 3, 4) of the BFS algorithm.

II. Background

A. Breadth-first search

We consider undirected, unweighted graphs of the form $G = (V, E)$ with sets of $|V|$ vertices (nodes) V and $|E|$ edges E. A breadth-first search begins at a root node v_r contained within the largest connected component $v_r \in V_c, V_c \subset V$ and traverses each edge e_{rj} for every neighbor v_j. As such, the graph is traversed in *levels*, where all nodes at each level are explored before the next level is processed. In line with previous work exploring BFS performance ([3], [5], [6]), we consider the variant of the kernel that produces the distance array (`dist` in the pseudocode of Figure 1a), which is the distance (in terms of BFS steps) of each visited node from the root node.

B. Sparsity and the Small-World Property

A *small-world* graph is one in which the diameter is small, e.g., "six degrees of separation", and for social graphs such as Facebook has been shown to be as low as four [7]. Small-world graphs generally exhibit scale-free degree distributions of the form $y = x^a$, i.e., consisting of very few high-degree central "hubs" and very many low-degree nodes that form the periphery [8]. This means that when BFS starts there is a high probability that the root node will only be connected to a few neighboring nodes, and those neighbors connected to a few, and so on. Thus, the first iterations of BFS visit a small percentage of the graph. However, as more and more edges are traversed, the frontier size increases dramatically (see Figure 2), constituting a large percentage of the network. As the BFS frontier size is correlated with available parallelism [9], the same figure is representative for how amenable the different steps are for parallelization.

Large small-world networks are generally sparse, meaning that most nodes are not neighbors. To take advantage of this, the graph is typically stored in a sparse adjacency matrix form such as Compressed Sparse Column (CSC)[1], as illustrated in Figure 1b.

C. BFS in the Language of Linear Algebra

Choosing a different representation for an algorithm may expose algorithmic characteristics that can be exploited for accelerator design. Towards this end, we will be using the "matrices over semirings" concept [10] to express BFS as a linear algebra operation. The core idea is to substitute the number data type and the operators for multiplication and addition in linear algebra to express a variety of algorithms as matrix-vector operations.

Specifically, we will make use of the matrix-times-vector operation on the Boolean semiring to perform BFS. In practice, this operation "multiplies" a binary matrix and a binary vector, with the regular multiply and add operations substituted with the Boolean AND and OR operators, respectively. To disambiguate from regular matrix-vector multiply over real numbers, we will use $\circledast$ to denote this operator. As illustrated in Figure 1c, each $y_t = A \circledast x_t$ operation corresponds to a breadth-first step, and each result vector y_t is the representation of the visited nodes in the graph after step t. The matrix A in the operation is the adjacency matrix of the graph, while the initial input vector x_0 is initialized to all zeroes, except a single 1 at the location of the root node. The result vector y_t is used as the input vector x_{t+1} of the next step, which in turn generates more visited nodes in its result vector until the result converges (i.e., no more nodes can be visited).

We note that the properties of the Boolean semiring can be exploited here to perform less work: x_t elements that are zeroes can be simply skipped since AND operation with a 0-input will always return a 0. Furthermore, only a subset of the 1-entries (those that were produced in the previous step) in x_t may actually produce new 1-entries in y_t. From a BFS standpoint, these observations correspond to only the newly-visited (frontier) nodes doing useful work. In terms of linear algebra, we can say that the x_t vector can be treated as *sparse*, though dense treatment is functionally correct.

As we will show in Section III-B, this representation enables us to view BFS in a way that permits trading redundant computations for higher memory bandwidth. Additional advantages of

[1] For an undirected graph, the Compressed Sparse Row (CSR) representation is equivalent to CSC. Our work assumes CSC and column-major traversal.

this approach include the potential for easier integration with software that use linear algebra as a building block, as well as the ability to apply memory system optimizations designed for iterative sparse linear algebra (e.g., [11], [12]).

III. Hybrid BFS on an FPGA-CPU Hybrid

Our accelerator system specifically targets in-memory small-world graphs, where breadth-first search exhibits a characteristic profile in terms of the frontier size explained in Section II-B. Since the amount of parallelism available during BFS is closely correlated with the frontier size, the opportunity for heterogeneous mapping onto different types of processing elements presents itself. A single CPU core can be used for the steps with small frontiers, while a high-throughput accelerator can be used for the steps with large frontiers. Following the example of [6] for a CPU-GPGPU system, we adopt the hybrid approach for a single-chip FPGA-CPU heterogeneous processor. The primary advantage of this platform is the low cost of switching execution modes back and forth, effectively adapting to the amount of parallelism available in small-world BFS. As justified by our results in Section IV-B, our strategy is to start the BFS kernel on the CPU, switch to the FPGA accelerator after a few steps to rapidly explore most of the graph, then switch back to the CPU for the last few steps.

In the following sections, we will first develop several ideas around implementing BFS on the Boolean semiring, and afterwards describe the architecture of our accelerator system.

A. Decoupled Distance Generation

The Boolean semiring matrix-vector representation of BFS given in Section II-C is very lean in terms of storage requirements, which makes it suitable for a hardware accelerator implementation. Specifically, the x and y vectors require only one bit of storage per graph node. Since y will be random-accessed due to matrix sparsity, keeping the range and volume of the data to be random-accessed to a minimum is advantageous for performance. Unfortunately, iteratively invoking $\circledast$ is not sufficient[2] for BFS as it only generates the node visited status and not the `dist` array.

We address this shortcoming by introducing a separate *distance generation* (or `DistGen`) step after each $\circledast$ invocation. A node i has distance t if it was visited during the BFS step t, and we know that a node cannot go from being visited to unvisited. Thus, we can conclude that the node has distance t if it is unvisited in x_t and visited in y_t, or $dist[i] = t \iff (x_t[i] == 0 \wedge y_t[i] == 1)$. To generate the distance information, it is sufficient to examine the input and output vectors of each BFS step, after each step is finished. This array compare operation is decoupled from the regular BFS step and can be easily parallelized or implemented in hardware (Section III-C4) to reduce its performance overhead. The complete BFS algorithm expressed with $\circledast$ and `DistGen` is listed in Algorithm 1.

Algorithm 1 BFS with $\circledast$ and `DistGen`.

```
function DistGen(dist[], level, x, y)
    $updates \leftarrow 0$
    for $i \leftarrow 0..N-1$ do
        if $x[i] == 0$ & $y[i] == 1$ then
            $dist[i] \leftarrow level$
            $updates \leftarrow updates + 1$
    return $updates$
function BFSAsLinearAlgebra(A, root)
    $x, y \leftarrow [0, 0, ..0]$; $dist \leftarrow [-1, -1, .. -1]$
    $x[root], y[root] \leftarrow 1$; $dist[root] \leftarrow 0$
    $level \leftarrow 1$; $converged \leftarrow 0$
    while $!converged$ do
        $y \leftarrow A \circledast x$
        $converged \leftarrow$ DistGen($dist, level, x, y$)
        $y \leftarrow x$; $level \leftarrow level + 1$
    return $dist$
```

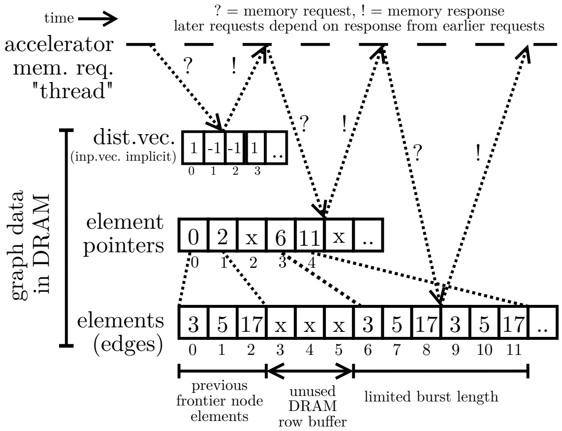


Fig. 3: Structure of memory requests for sparse x_t.

B. Trading Redundant Computation for Bandwidth

As traversal of sparse graphs involves little actual computation and is known to be a memory-bound problem, delivering graph data with high bandwidth is critical for accelerator performance. Therefore, a careful analysis of memory access patterns is a critical step for designing a BFS hardware accelerator. In our accelerator as with many other systems, the BFS inputs are stored in and accessed from DRAM. Due to the inherent latency and three-dimensional organization of modern DRAM chips, three features are key to achieving a substantial portion of available DRAM bandwidth: high request rate to mitigate latency, large bursts, and a sequential access pattern to maximize row buffer hits.

In Section II-C, we observed how the input vector x_t could be treated as sparse to avoid redundant work. But more importantly, the linear algebra notation tells us that we can treat x_t as dense and still get a correct BFS result. In a hybrid accelerator, this seemingly unorthodox idea of treating the BFS input frontier as dense and performing redundant work can be actually beneficial for overall performance due to simpler DRAM access patterns. How we treat the x_t vector influences how the matrix A data will be accessed, and in turn, with how much bandwidth.

Figure 3 depicts how treating x_t as sparse influences the memory requests to the matrix data. The accelerator must first obtain a node index that is a member of the frontier by reading `dist`, then obtain this node's start and end pointers, and finally obtain the list of adjacent edges using these pointers. Visible here is the dependency of requests on responses to

[2] Although using the tropical semiring $(R \cup \{\infty\}, min, +)$ would remedy this, each vector element in the tropical semiring is a number and loses the leanness/storage advantages of the Boolean representation.

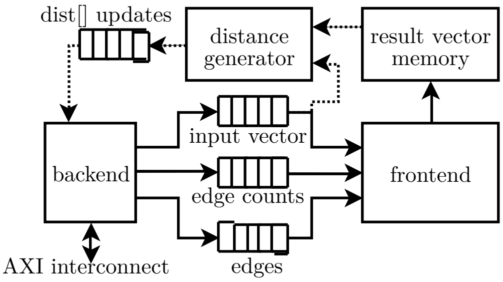


Fig. 4: Architecture overview of a dense x_t processing element. Dotted lines are active only during the distance generation operation.

previous requests; this is typical of applications with indirect accesses, and sparse treatment of x_t leads to two levels of indirection. More concretely, the effects are three-fold. The first is the limitation of the request rate by the response rate. Secondly, the length of read bursts to the `elems` array are limited by the number of edges in the node. Finally, even though the reads to the `elems` array are sequential, there may be large gaps in between the used portions of the array due to frontier nodes being far apart, causing parts of the DRAM row buffer to go unused. This becomes especially prominent with several accelerators requesting different parts of the edges array in parallel. Although this method avoids doing redundant work, these three effects can dramatically decrease DRAM bandwidth utilization, especially for platforms whose memory systems cannot handle a large number of outstanding requests.

In contrast, if we treat x_t as dense and consider every node of the graph, the access pattern of A becomes significantly simpler. In particular, we can simply read out the entire matrix, which can be done with maximum-length burst read operations and without having to wait for responses from previous requests. This is a much simpler and more suitable access pattern for achieving high DRAM bandwidth. Memory bandwidth is, of course, only half the story; the amount of redundant work performed by treating x_t as dense is nontrivial – the smaller the frontier, the more redundant work will be performed. However, since we are building a hybrid CPU-FPGA system where the accelerator handles the BFS steps with large frontiers, the overhead of redundant work is less significant. In fact, as described in Section IV, our experiments on the Zynq platform with scale-free graphs show that the dense x_t treatment always outperforms the sparse treatment in this hybrid approach.

C. Processing Element Architecture

Based on the ideas from Sections III-A and III-B, we now describe a hardware architecture for BFS. To compare the effects of sparse and dense x_t treatment described in Section III-B, we consider two processing element (PE) variants.

1) Dense x variant: The architectural overview of a dense x_t processing element (PE) is illustrated in Figure 4. The architecture is organized in a data-flow manner, and modularized into three main components: a *backend*, which connects to the DRAM via the system interconnect, a *frontend* for performing the $\circledast$ operator, and a *distance generator*. The backend is responsible for all interaction with main memory, which includes reading out the matrix and vector data, and writing updates to the distance vector. The matrix and vector data are requested by the backend in large bursts and made available to the frontend, which performs the BFS step operation and updates the result vector memory. Concretely, the frontend simply writes 1s to result memory addresses indicated by the edges whenever the input vector is 1. The edge counts data is used to determine when to read a new input vector element. An input vector value of 0 implies edges from a yet-unvisited node, and this data is simply dropped without further operation. The control and status interface of the accelerator is provided through memory-mapped registers, which are not shown in the figure.

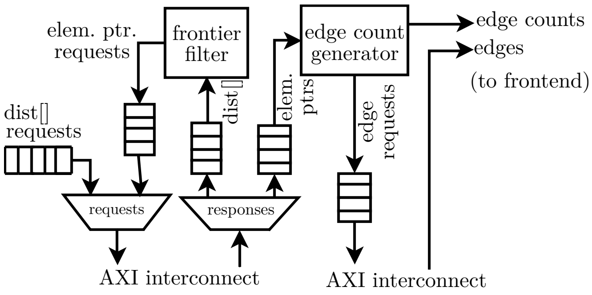


Fig. 5: Backend architecture for sparse x_t variant.

2) Sparse x variant: In terms of the overall architecture, the sparse x_t variant is identical to the dense one except that it does not need the input vector FIFO (since the sparse treatment implies all x_t values are ones). However, as illustrated in Figure 5, the internals of the backend are substantially different from the dense variant. The sparse input vector (or frontier indices) is generated internally by a *frontier filter*, which scans the values of the distance array and emits the indices whole values were written in the previous BFS step. Afterwards, the start- and end-pointers of each generated index are requested. These pointers are used to request the edge data for this node, and also to produce the edge count information for the frontend.

3) Stall-free y Writes: To keep the accelerator running without stalls, it is important that the frontend is able to consume data as fast as the backend is producing it. The result vector y_t is random-accessed by the frontend during the $\circledast$ operation, since the accessed node locations depend on the visited graph edges. Thus, we can abstract the functionality of the frontend as *handling a stream of writes to random addresses*. If the result vector is stored in DRAM, the write request buffers of the interconnect and memory controller can fill up and stall the entire accelerator. To avoid this, our solution exploits the leanness of vector representations. Since our approach requires us to keep only a single bit per graph node, we can effectively utilize dual-port FPGA on-chip RAM to provide two very fast, fine-grained random accesses per cycle. Although this limits the largest graph size we can process, the on-chip RAM capacity of modern FPGAs is quite large and graphs with millions of nodes can still be processed in this manner. Another option is to explore a single BFS step of a large graph across more than one execution by pre-partitioning, which we do not explore in this work.

4) Distance Generator: After each BFS step is finished, the distance generator is invoked to implement `DistGen` as

CPU core	Dual ARM Cortex-A9, 666 MHz
CPU cache	32 KB L1D+L1I, 512 KB L2
DRAM and bandwidth[3]	512 MB DDR3, 3.2 GB/s
FPGA logic resources	13300 logic slices, 53200 slice LUTs
FPGA on-chip RAM	560 KB (BRAM)

[3] DDR controller max is ~75% of the theoretical max of 4.2 GB/s [13]

TABLE I: Characteristics of the ZedBoard.

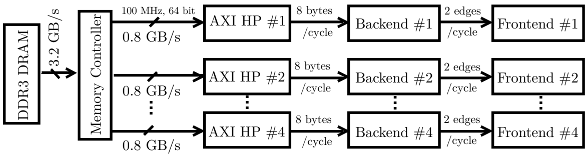


Fig. 6: Rate-balanced design for the ZedBoard.

described in Section III-A. This involves comparing the input and result vectors and finding the nodes that went from being unvisited to visited. The indices of these nodes is passed to the backend for actually writing the current BFS distance to the corresponding memory locations, and also for updating the x vector for the next BFS step for the dense variant.

D. Accelerator System Implementation

Our accelerator system is built and deployed on the ZedBoard platform with the Xilinx Zynq Z7020 FPGA-CPU hybrid [13], whose characteristics are shown in Table I. The accelerator components were first built in Chisel [14]. Verilog descriptions were then generated using the Verilog backend, and imported into Vivado as IP blocks. Vivado IP integrator (version 2014.4) was used afterwards to build the accelerator system, including Xilinx-provided IP blocks for the result memory block RAM (BRAM) and AXI interconnect. The 64-bit AXI high-performance (HP) slave ports, which are capable of utilizing about 80% (3.2 GB/s) of the DRAM bandwidth, are used to feed the accelerators with data.

1) Parallelism and Rate-Balancing: As parallelism is key to achieving high performance with FPGA accelerators, we explore the graph with parallel PEs in our system. We use row-wise partitioning of the input matrix to ensure that the random-access range of each partition fits within the result vector memory of one PE. The entire input vector is read from DRAM by all PEs during a step. After the step, each PE updates a portion of the input vector with its result (during the `DistGen` operation). We use the rate-balancing ideas for FPGA sparse matrix-vector multiplication from [11] to estimate the number of PEs required to consume the available DRAM bandwidth in the platform. Every cycle, each PE backend can fetch a maximum of 8 bytes through the interconnect, and each frontend can process up to two edges of 4 bytes each by issuing writes to dual-port result memory. Assuming $F_{clk} \geq 100$ MHz, we can obtain a rate-balanced design by attaching one PE to each of the four AXI HP ports, as shown in Figure 6.

2) Software BFS Implementation and FPGA-CPU Switching: The software BFS variant runs on a single Cortex-A9 core inside the Zynq system with caches enabled, and uses the bitmap optimization [15] to track node visit status for better performance. Since the visit status bitmap corresponds to an input vector, it makes switching between CPU and FPGA easier. Switching from CPU to FPGA execution requires updating the PE result memories with the node visit status. The accelerator itself can be used to do this switching by using an identity matrix as the graph, and the visit status bitmap as the input vector. Switching back to the CPU after FPGA execution requires a frontier queue to be reconstructed from the distance vector, which is also a highly data-parallel task (i.e., search through an array to find indices with a given value). We include a simple hardware accelerator to keep the performance overheads from the switching to a minimum.

3) Method Switching in Hybrid: After each BFS step is finished, the software uses a simple model to decide which method[3] should be used for the next step. The hybrid BFS starts execution in software, and switches to FPGA execution when the predicted BFS step time for the FPGA is shorter. We exploit the predictability of the dense variant (see Figure 7) to model its execution clock cycles with the following formula:

$$T_{\text{step}} = T_{\text{DistGen}} + T_{\circledast} = \cdot \frac{1}{\beta}\left(\frac{nodes}{\#PE}\right) + \frac{1}{\beta}\left(nodes + \frac{edges}{2 \cdot \#PE}\right)$$

where β is the fraction of utilized bandwidth. The FPGA execution continues until the frontier size drops to below $\theta\%$ of all graph nodes. Afterwards, the software BFS takes over until the search is terminated. β, θ are determined empirically.

IV. Results

We now present the results from the experimental evaluation of our accelerator system. For BFS performance testing, we use synthetically generated RMAT graphs with the Graph500 benchmark parameters (A=0.57, B=0.19, C=0.19) in line with previous work [3], [5]. We refer to an RMAT graph with scale S (2^S nodes) and edge factor E ($E \cdot 2^S$ edges) as RMAT-S-E. To avoid reporting results from trivial searches, we only consider nodes which are in the largest connected component in the graph, whose size is $O(N)$ for RMAT graphs. Due to the limited amount of BRAM available on the Zynq, we were unable to evaluate our approach for graphs larger than scale 21 (two million nodes), but our technique can be applied to larger graphs on bigger FPGAs (e.g., up to scale 29 on the largest UltraScale+ Virtex 7).

A. FPGA Resource Utilization and Clock Frequency

The area and timing results from Vivado 2014.4 for both the dense and sparse variants are similar. For a 4 PE design, the accelerator system can run at up to 150 MHz and uses about 39% of the FPGA logic resources, and 97% of the FPGA on-chip RAM (BRAM). 82% of the BRAM is used for result vector memory, and about 15% for the FIFOs in the memory system and inside the PEs.

B. Comparing Software, Sparse and Dense BFS

To motivate the hybrid BFS solution, we start by comparing the performance of the sparse and dense accelerator variants with software BFS. We perform BFS on RMAT-19-32 with equal number of PEs (4) and clock frequency (100 MHz) for both accelerator variants, and plot the number of clock cycles taken for each BFS step (including distance generation) in Figure 7. Our first observation here is that there is no single best

[3] As our results in Section IV-B indicate that the dense variant outperforms the sparse, we only consider software-dense hybrid BFS.

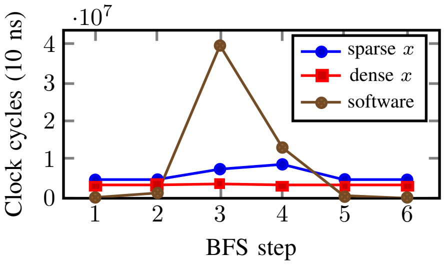


Fig. 7: Execution time per step of accelerator and software BFS on an RMAT-19-32.

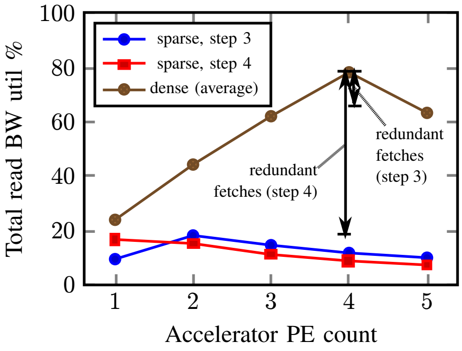


Fig. 8: Aggregate read bandwidth utilization for steps 3 and 4 of an RMAT-19-32.

method; as expected, the fastest method differs from step to step. The goal of the hybrid scheme would be to choose the fastest method for each step, which can be deduced from this plot. We can see that exploring steps 1 and 2 with the CPU, switching to the dense x accelerator for steps 3 and 4, then switching back to the CPU for the last two steps gives the fastest execution. The dense variant outperforming the sparse implies that *the benefits from the increase in DRAM bandwidth is larger than the cost of redundant data fetches* for the middle steps.

To better understand why the dense variant outperforms the sparse during the large-frontier steps, we plot the aggregate read bandwidth utilization (compared to the memory link capacity of 32 bytes/cycle) for the BFS steps 3 and 4, for both variants with increasing PE counts. The reason we vary the PE count is to reveal the effects of increased memory pressure from more parallel requests. For the sparse variant, we actually observe that the total utilized bandwidth decreases by adding more PEs. The particularly low utilization in step 4 is likely caused by the frontier being larger (3x) than the step 3 frontier, causing parallel PE requests all across the edges array and leading to many DRAM bank conflicts and row buffer misses. On the other hand, the bandwidth utilization for the dense variant is much better than the sparse and increases almost linearly with PE count, peaking at 78% for 4 PEs, and does not vary between the two steps. Adding more than 4 PEs requires the AXI HP ports to be shared and decreases bandwidth utilization and performance. Even when we account for the significant cost of redundant data fetches in the dense variant (see annotations in Figure 8), we can see that the dense variant has better bandwidth utilization than the sparse variant.

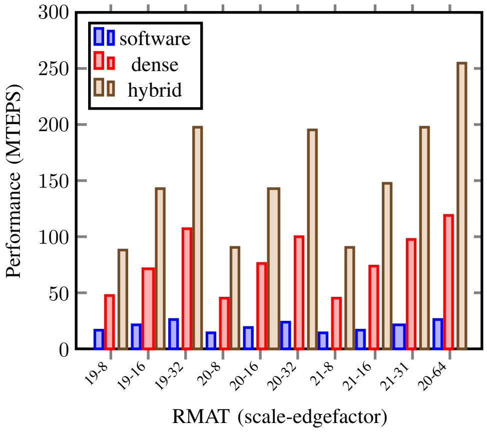


Fig. 9: BFS performance on a range of RMAT graphs.

It is important to keep in mind that the tradeoffs between these methods will depend on the particular platform, graph and number of PEs being used. However, the performance depicted in Figure 7 is representative of all our experiments on the ZedBoard with RMAT graphs; the software solution is best for the start and the end, and the dense variant always outperforms the sparse variant for the middle steps. We will therefore omit results for the sparse variant from the rest of our discussion.

C. Hybrid BFS Performance

We now report performance results for software-only, dense frontier accelerator-only and hybrid BFS approaches on a range of RMAT grahs. The hybrid BFS works as described in Section III-D3. We use $\beta = 0.78$ from Figure 8 and empirically determine that $\theta = 5\%$ performs close-to-ideal switching. We measure performance in MTEPS (millions of traversed edges per second), which is obtained by dividing the graph edge count by the execution time. The results are averaged over 16 BFS operations started from randomly chosen root nodes within the largest connected component for each graph.

Figure 9 summarizes the BFS performance for a range of RMAT graphs. The software-only BFS has a performance of 22 MTEPS on average. Since none of these graphs fit into the CPU cache, a large number of data cache misses (~%25 miss rate) degrade the performance. The accelerator is 3.9x as fast on average as the software-only BFS. Given the 4x frequency advantage of the CPU and large amounts of redundant data fetches and work performed by the accelerator, this speedup further supports the claim that slow-clocked but parallel FPGA accelerators are suitable for irregular, memory-bound applications. Finally, the hybrid method combines the “best of both worlds” and outperforms both the accelerator-only and software-only BFS with speedups of 2x and 7.8x, respectively. The performance of the accelerator is correlated with the graph edge factor, with the hybrid BFS achieving a maximum of 255 MTEPS for edge factor 64.

D. Hybrid Execution Time Breakdown and Scaling

Figure 10 provides a breakdown of the execution time for the hybrid BFS on scale 19 graphs with different edge factors. To show how performance scales with parallel PEs, we provide

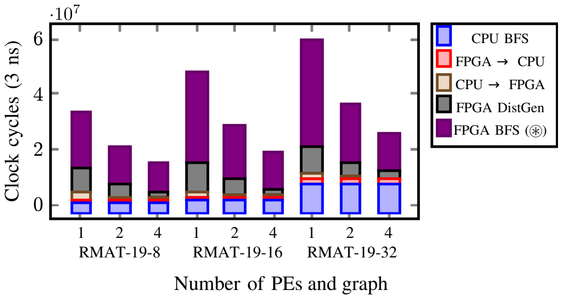


Fig. 10: Hybrid execution time breakdown.

Work	Platform	Avg. MTEPS	BW (GB/s)	MTEPS/BW
[3]	Convey HC-2	~1600	80	20
[5]	Convey HC-2	~1900	80	24.375
[6]	Nehalem+Fermi	~800	128	6.25
This work	ZedBoard	172	3.2	**53.7**

TABLE II: Comparison to prior work.

data points for 1, 2 and 4 PEs. We observe that the execution time of FPGA DistGen and ⊛ operations are almost halved by doubling the PE count. This is consistent with the bandwidth scaling in Figure 8. The FPGA ↔ CPU switching overheads account for about 10% of the execution time on average. The overhead of the DistGen operation decreases with increasing graph edge factor, and ranges between 9%-15% of the total execution time for 4 PEs. Since the CPU execution time does not vary with increased PE count, we observe an "Amdahl's Law" trend in performance scaling. With 4 PEs on RMAT-19-32, the CPU execution time accounts for 30% of the total, and would eventually become a performance bottleneck on a larger system with more bandwidth and PEs. This can be remedied by using a faster CPU with a better memory system.

E. Comparison to Prior Work

As most works targeting high-performance BFS use MTEPS as a metric, comparing raw traversal performance is possible but the available memory bandwidth in the hardware platform sets a hard limit on achievable BFS performance. Our experimental results are from a ZedBoard with much less (about 1/20th of those in Table II) DRAM bandwidth than platforms in prior work and thus is comparatively slow. However, as indicated by our results in Sections IV-D and IV-B, the performance of our method scales well as more bandwidth becomes available. Low memory bandwidth is not an inherent problem with single-chip FPGA-CPU solutions and we believe more powerful versions of these devices, such as the Xilinx UltraScale+ Zynq and Altera Stratix 10 SoCs, are likely to be deployed for high-performance computing in the near future.

Taking into account the memory-bandwidth-bound nature of BFS on sparse graphs, we use *traversals per unit bandwidth* as a metric to enable fair comparison with prior work, which we obtain by dividing traversal speed in MTEPS by external memory bandwidth in GB/s. Table II presents a comparison with several related works on reported average BFS performance, available DRAM bandwidth and traversals per bandwidth over RMAT graphs similar to the ones we used. Our method is more than twice as effective in terms of traversals per bandwidth compared to the next-best solution, which is also on an FPGA.

V. Related Work

Fast graph traversal has been approached from a range of architectural methods from general-purpose CPU and multi-core/supercomputing approaches exposing parallelism [15]–[20] to graphics processing units (GPUs) [21]–[24], as well as hybrid CPU-GPU methods [6], to more recent methods taking advantage of reconfigurable hardware [3], [5], [25], [26]. Many principles are constant across architectures, for example, the performance hit associated with irregular memory accesses similarly affects GPU systems, single and multi-CPU systems, and FPGAs. For brevity in the following text, we focus primarily on FPGA-based related work.

Early reconfigurable hardware approaches attempted to solve graph traversal problems on clusters of FPGAs [27], [28], but were limited by graph size and synthesis times because the reconfigurable logic was used to model the graph itself. Recent works have implemented optimizations for BFS and other irregular applications on multi-softcore processors in FPGAs, yielding promising results [19], [26], [29]. More closely related research to ours has explored highly parallelized processing elements (PEs) and decoupled computation-memory [3], [5]. Observing that the execution time of BFS on small-world networks is dominated by the intermediate levels, Betkaoui et al. [3] decouple the communication and computation elements in an FPGA to maintain throughput of irregular memory accesses, arguing that on-chip memories in FPGAs are too small for contemporary graphs. In the same vein, the authors of [5] present optimizations to BFS that essentially merge the first two request-response arrows in Figure 3 and report increased performance due to fewer memory requests.

Parallel BFS implementations on GPUs are numerous [21]–[24], with research typically focusing on level-synchronous [6], [23] or fixed-point [16] methods. CPU and multiprocessor-based approaches attempt to hide memory operation latencies with caches, but for irregular algorithms such as BFS this is not effective. Other techniques, such as using cache-efficient data structures are often employed [17], [24]. A notable approach by Agarwal et al. [15] makes locality optimizations on a quad-socket system in order to reduce memory traffic and proposed a bitmap to keep track of visited nodes in a compact format (1 bit per node). This study is widely cited for the latter aspect [20], [22], [23], and in our work we adopt this optimization as well. Beamer et al. [20] argue for reducing the number of edges traversed through a direction-optimizing approach that switches between parent-child and child-parent traversal depending on frontier heuristics.

The CPU-GPU hybrid method of [6] is similar to our work in that a switching approach is employed: a queue-based method is efficient when the frontier size is small and a read-based method that sequentially reads the adjacency list is more efficient when the frontier size is large (as is typical for small-world graphs). Our approach differs in two main points: we can switch back to executing on the CPU owing to tight CPU-FPGA integration (which is avoided in [6] due to high overhead) and we exploit the frontier density in the middle BFS steps for trading redundant computations for increased bandwidth. Finally, the authors of [30] emphasize the benefits of sequential patterns for achieving high storage device bandwidth in graph algorithms, and propose an edge-centric scatter-gather framework with streaming partitions. Although it also exploits redundant work for increased bandwidth, this approach sacrifices some efficiency (e.g., random accesses to nodes) to achieve general applicability for in- and out-of-memory graphs. It is also intended for cache-based multiprocessors, whose efficiency on BFS are limited. To the best of our knowledge, ours is the first approach to consider redundant work-bandwidth tradeoffs in BFS for a hybrid FPGA-CPU system, or from a hardware-near perspective in general.

VI. Conclusion and Future Work

In this work, we have presented a hardware-accelerated BFS architecture for FPGA-CPU hybrid systems that can effectively take advantage of the varying degree of parallelism in small-world graphs. Viewing BFS as a matrix-vector operation on the Boolean semiring, we showed how the volume of the random-accessed data can be reduced significantly and kept in on-chip RAM for a stall-free BFS datapath. Another revelation from the representation, the idea of treating the input vector as dense instead of sparse, allowed us to trade redundant computations for increased DRAM bandwidth. Our experiments on the ZedBoard platform suggest that the hybrid system performance scales well with increased bandwidth and outperforms previous techniques in terms of traversals per bandwidth.

Future work will include evaluating this technique on more powerful FPGA-CPU platforms and exploring more graph algorithms with the matrices-on-semirings idea. The source code for our accelerator system can be obtained from http://git.io/veGTL for further investigations.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *et al.*, *Introduction to algorithms*, vol. 2. 2001.

[2] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proc. of the ninth ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, 2003.

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

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Comp. Arch. (ISCA), 2014 ACM/IEEE 41st Intl. Symp. on*, 2014.

[5] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "CyGraph: A reconfigurable architecture for parallel breadth-first search," in *Parallel & Distributed Processing Symp. Workshops (IPDPSW)*, 2014.

[6] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Parallel Architectures and Compilation Techniques (PACT), 2011 Intl. Conf. on*, 2011.

[7] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna, "Four degrees of separation," in *Proc. of the 4th Annual ACM Web Science Conf.*, 2012.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining.," in *SDM*, vol. 4, 2004.

[9] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *ACM SIGPLAN Notices*, vol. 46, 2011.

[10] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*, vol. 22. SIAM, 2011.

[11] Y. Umuroglu and M. Jahre, "An energy efficient column-major backend for FPGA SpMV accelerators," in *Computer Design (ICCD), 2014 32nd IEEE Intl. Conf. on*, 2014.

[12] Y. Umuroglu and M. Jahre, "A Vector Caching Scheme for Streaming FPGA SpMV Accelerators," in *Applied Reconfigurable Computing*, vol. 9040, 2015.

[13] Xilinx, "Zynq-7000 All Programmable SoC Technical Reference," Feb. 2015.

[14] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proc. of the 49th Annual Design Automation Conf.*, 2012.

[15] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. of the 2010 ACM/IEEE Intl. Conf. for High Perf. Computing, Netw., Storage and Analysis*, 2010.

[16] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Supercomputing, 2005. Proc. of the ACM/IEEE SC 2005 Conf.*, 2005.

[17] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proc. of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2010.

[18] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems," in *FUTURE COMPUTING 2014, The Sixth Intl. Conf. on Future Computational Technologies and Applications*, 2014.

[19] M. Ceriani, G. Palermo, S. Secchi, A. Tumeo, and O. Villa, "Exploring Manycore Multinode Systems for Irregular Applications with FPGA Prototyping," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual Intl. Symp. on*, 2013.

[20] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, 2013.

[21] E. Mastrostefano and M. Bernaschi, "Efficient breadth first search on multi-GPU systems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 9, 2013.

[22] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the Kepler architecture," *arXiv preprint arXiv:1408.1605*, 2014.

[23] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, 2012.

[24] U. A. Acar, A. Charguéraud, and M. Rainey, "Fast parallel graph-search with splittable and catenable frontiers," tech. rep., 2015.

[25] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight Jr, and A. DeHon, "GraphStep: A system architecture for sparse-graph algorithms," in *Field-Programmable Custom Computing Machines, 14th Annual IEEE Symp. on*, 2006.

[26] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on FPGA for Breadth-first Search," in *Field-Programmable Technology (FPT), 2010 Intl. Conf. on*, 2010.

[27] J. W. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures," in *Photonics East'96*, 1996.

[28] A. Dandalis, A. Mei, and V. K. Prasanna, "Domain specific mapping for solving graph problems on reconfigurable devices," in *Parallel and Distributed Processing*, 1999.

[29] S. Secchi, M. Ceriani, A. Tumeo, O. Villa, G. Palermo, and L. Raffo, "Exploring hardware support for scaling irregular applications on multi-node multi-core architectures," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th Intl. Conf. on*, 2013.

[30] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*, 2013.