CAMs as Synchronizing Caches for Multithreaded Irregular Applications on FPGAs

Invited Paper

Skyler Windh, Prerna Budhkar, Walid A. Najjar
Department of Computer Science
University of California, Riverside
{windhs, pbudh001, najjar}@cs.ucr.edu

***Abstract*—Irregular applications, by their very nature, suffer from poor data locality. This often results in high miss rates for caches, and many long waits to off-chip memory. Historically, long latencies have been dealt with in two ways: (1) latency mitigation using large cache hierarchies, or (2) latency masking where threads relinquish their control after issuing a memory request. Multithreaded CPUs are designed for a fixed maximum number of threads tailored for an average application. FPGAs, however, can be customized to specific applications. Their massive parallelism is well known, and ideally suited to dynamically manage hundreds, or thousands of threads. Multithreading, in essence, trades memory bandwidth for latency. Therefore, to achieve a high throughput, the system must support a large memory bandwidth. Many irregular application, however, must rely on inter-thread synchronization for parallel execution. In-memory synchronization suffers from very long memory latencies. In this paper we describe the use of CAMs (Content Addressable Memories) as synchronizing caches for hardware multithreading. We demonstrate and evaluate this mechanism using graph breadth-first search (BFS).**

Keywords—FPGA; Multithreading; bandwidth; CAM; synchronizing caches.

I. Introduction

Very large memory latency is the central problem facing computer designers. The traditional solution, in CPU architectures, relies on massive cache hierarchies (L1 to L4) to *mitigate the memory latency*. This approach, however, is too costly for FPGA-based systems. Hardware-based multithreading relies on *latency masking* by switching execution to a ready thread while the memory access is performed. Multithreaded CPUs, such as the Tera MTA (Cray XMT) [1] and the Oracle SPARC T series [2], are designed for a fixed number of hardware contexts (threads) and each context is fully provisioned with all the registers required to support a general purpose execution.

Irregular applications suffer from very poor spatial and temporal localities but also exhibit a very large degree of parallelism the exploitation of which is hampered by the long memory latency. FPGA-based hardware multithreading, such as the MT-FPGA model (Section II-B), relies on latency masking to increase the throughput and hence exploit the available parallelism. However, unlike multithreaded CPUs, it configures a custom data-path for each application and hence the state of the computation that must be saved for each thread is kept to a minimum and the number of active threads can be *dynamically* increased as required by the application within the limits of the available on-chip storage.

Some irregular applications, such as sparse linear algebra, can be executed without having to rely on inter-thread synchronization. Graph algorithms in general, and specifically Breadth-First Search (BFS), do require such synchronization. However, the long memory latency makes in-memory synchronization prohibitively expensive. In this paper we explore the use of CAMs (Content Addressable Memories) as *on-chip synchronizing caches*: each thread maintains locally, in its own CAM, the list of nodes it has visited hence avoiding unnecessary memory accesses. In the top-down BFS implementation, all the nodes adjacent to a visited node are marked as visited, in parallel. This approach may result in an exponential growth of work most of it redundant, i.e. accessing already visited nodes. Caching these in a CAM results in a significant reduction of memory accesses and hence a higher throughput.

The main contribution of this paper is the exploration of CAMs as synchronizing caches for hardware multithreaded irregular applications on FPGAs, to our knowledge the first such study[1].

The rest of this paper is organized as follows - Section II covers the background and motivation of multithreaded architectures and the BFS algorithm. Section III discusses the use of CAMs on FPGAs and the design costs and tradeoffs in that space. Section IV demonstrates a motivating example of how to implement a multi-threaded FPGA BFS using CAMs as synchronizing caches. Section V covers the implementation of the design and the resource utilization. The paper then concludes with Section VI.

II. Background and Motivation

This section presents a background on general multithreaded architectures, the MT-FPGA execution model and its use in sparse linear algebra and database applications, and a review of the previous work on accelerating the BFS algorithm on FPGAs.

1. An expanded version of this study is available at http://www.cs.ucr.edu/~windhs/iccad2015.html

978-1-4673-8388-2/15/$31.00 ©2015 IEEE

A. Multithreaded Architecture

Tera Corporation built the Tera MTA (Multi-threaded Architecture) in the early 90s [1]. This design had 256 processors and 64 GB of shared memory organized as a distributed NUMA architecture. The interconnection network allowed better scaling to a larger number of processors. These custom processors supported the issuing of one memory request per thread per cycle. The maximum memory latency from any processor to any memory module was 128 cycles. Each processor could support up to 128 active threads. The MTA design [1] [3] was later evolved into the Cray XMT. While the MTA had only 256 processors the XMT machine could support up to 8,192 processors, but the largest ones built had 512 processors. The shared memory was also increased from 1 TB to 128 TBs for the MTA and the clock speed was improved from 220 MHz to 500 MHz.

B. MT-FPGA Execution Model

The MT-FPGA (Multithreading on FPGAs) [4] is an execution model that combines the memory masking ability of multithreaded execution with a customized data path. This execution model suspends the thread as soon as it performs a read and ready threads are given chance to execute. It performs decoupling by buffering the returned data in the order it was requested. This execution model exhibits following advantages:

1. Can support hundreds of outstanding memory requests, hence massive parallelism
2. Full utilization of the datapath
3. The state of the thread is extremely small, and can therefore cater to more number of pending threads, stored on a FIFO.

A new tool called CHAT **(**Compiled hardware Accelerated Threads**)** uses MT-FPGA model. This compilation tool generates customized hardware support for multithreaded execution on FPGAs and claims to ease the hardware development effort for complex irregular kernels. CHAT was tested and demonstrated using Sparse Vector Matrix Multiplication (SpVM). Using just one accelerator FPGA, CHAT shows a speed-up of up to 50x over a single Intel Xeon on simple irregular kernels.

The first end-to-end in-memory implementation of hash join using a MT-FPGA model is discussed in [5]. In this work FGPA uses massive multithreading during the build and probe phases to mask long memory delays, while it concurrently manages hundreds of thread states locally. Throughput results show a speedup between 2x and 3.4x over the best multi-core approaches with comparable memory bandwidths on uniform and skewed datasets. This work was then extended to perform aggregation operation on the relational database [6]. It implements a simple aggregation computing scheme based on hashing and used the multi-threaded architecture on FPGA to deploy it. The performance of this scheme ranges between 300 and 600 MTuples/sec depending on the key distribution in the dataset. Many real world applications demand high performance and power efficient graph algorithms.

C. Accelerating BFS

Graphs are the most natural and efficient way to represent social and biological systems, where nodes represent entities such as people, web sites and genes whereas edges represent the interactions (relationship, communication and regulations). As these problems grow in scale, parallel computing resources are required to meet their computational and memory requirements. This has motivated a substantial amount of work that deals with the design and optimization of graph exploration algorithms, in particular BFS designs, either for commodity processors [7] [8] [9] [10] [11] or for dedicated hardware [12] [13] [14] [14] [15] [16] [17] [18].

A multi-threaded graph engine was developed by [7] which implements a semantic graph database on commodity clusters. They have addressed the issue of irregular memory accesses by using lightweight software multi-threading and data aggregation. This implementation was able to maintain constant query throughput with the scaling of dataset size.

A parallelized BFS algorithm was described by [19] on multi-core architectures. They used a bitmap to keep track of visitation status of a node and demonstrated speedup over previous work. A significant speedup was shown by [20] on distributed memory machines. GPUs have also been chosen by some to speedup computations in variety of applications, including graph processing. A level synchronous BFS kernel for GPUs was proposed in [21] and showed improved performance over previous implementations. A slightly different approach was used in [22] to elevate the graph processing performance. This method used a prefix sum approach for cooperative allocation. In [17], the authors have presented a GPU programming framework for improving GPU-based graph processing algorithms. Another graph processing framework was proposed by [18]. This method uses G-shards and concatenated window representation to store graphs in GPU global memory and provide better performance over other state-of-art implementations.

With the advent of heterogeneous machines, such as Convey HC-1/HC-2 [15], which support cache coherent shared virtual memory accesses from both the software (CPU execution) and the hardware (FPGA execution), application acceleration has become much more feasible . For example, the Convey HC-2 has four Virtex-6 LX760 FPGAs, further allowing multiple sections of an application to be written to a FPGA without need of reconfiguration at runtime. A reconfigurable architecture for parallel BFS is presented in [13]. This method worked well for graphs with out-degree up to 32, but does not scale to outperform the approach in [14] for higher degrees. It uses level-synchronous BFS algorithm and uses a customized CSR representation to achieve a high throughput.

The design approach, proposed in [14] is based upon serializing execution and processing of data within an engine and parallelizing access to off-chip memory. The processing engine sequentially issues multiple requests to memory and use on-chip RAM to store data from memory.

There are multiple reference implementations of the BFS implementation. In the top-down approach, each parent vertex in the current queue visits all its children and adds them to a next

queue – then next becomes current. In the bottom-up approach, the processor continually checks each unvisited vertex to see if it has a neighbor that was visited during the previous level. The work presented in [23] emphasized that both top-down and bottom-up approaches are beneficial when applied to different parts of the graph. This method starts with top-down traversal on the host side and switches to bottom-up on the coprocessor after a specific cut-off level. Based on the similar observation [12] came up with a hybrid approach to perform breadth first search with concurrent processing on both the host and the coprocessor and achieved significant performance. This method initially starts with the top-down approach on the host, copies the result in the coprocessor memory, which is later used by BFS. For larger frontiers, bottom-up BFS is said to be more efficient [23] because once the vertex has found a parent it need not check rest of its neighbors.

In this work we have tried to improve the performance of top-down approach by using synchronizing caches. These caches help us to avoid some atomic operations thereby reducing the traffic to the off-chip memory and reduce the amount of redundant work done in the graph traversal.

III. CAM MECHANISM ON FPGAS

As we discussed in Section II, there has been a growing body of work using the MT-FPGA model. Moreover, like most models dealing with multiple, concurrent threads, there must be some consideration for synchronizing threads. With the high cost of trying to synchronize threads through the memory system, this works explores using CAMs as on-chip synchronizing caches.

A recent work describing Near-Associative Memories [24] provides many insights into the uses and challenges with CAMs. Also known as associative memories, CAMs provide a conflict-free mapping from an input key to a data value. CAMs typically operate in the inverse behavior of a Random Access Memory (RAM). Where in a RAM the user provides an address and the memory provides the data at that location, with a CAM, the user specifies the data and the CAM will return a match if the data is stored. Generally, CAMs are configurable to return all matching addresses as well.

However, that ability to do a content-based search comes at high energy and area cost – the hardware must support programmable and parallel matches over the entire memory structure. This limitation generally means that only shallow CAMs are feasible. Nonetheless, even shallow CAMs have proven very useful in domains such as networking and FPGA database operations.

Dhawan et al. [24] showed that a 512 entry CAM with 40 bit keys requires 60 Xilinx Block RAMs (BRAMs) on a Virtex 6 FPGA. That implies a 60x overhead over the stored memory capacity to implement the match logic. It also cites work that shows the overhead in Altera generated CAMs is about equivalent to that of Xilinx generated CAMs. The overhead shows that CAMs are difficult to implement on reconfigurable fabrics. Current FPGAs offer little custom hardware support (i.e. BRAMs, DSPs, etc) for CAMs. In addition, the parallel search within a single cycle results in large fan-out rates that grow quickly with CAM size.

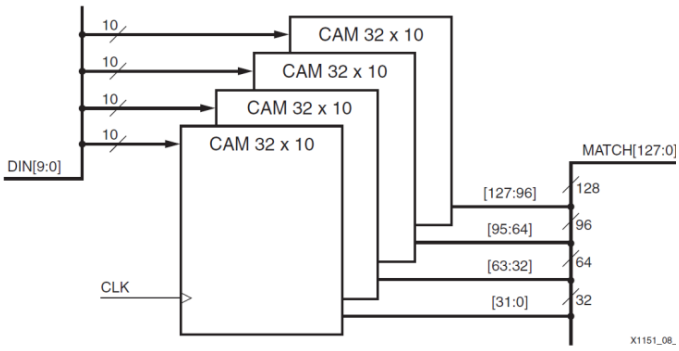

Figure 1- BRAM layout of a CAM using Xilinx tools [25]. As width grows beyond 10-bits, more BRAMs are added in parallel

The proposed near-associative memories are a great step in improving the usability and scalability of CAMs on FPGAs. A 512-entry memory only consumes 6 BRAMs instead of the Xilinx 60. However, the multiple cycle write latency can be a limiting factor for some high-throughput designs and kept our design to using custom CAMs.

IV. HARDWARE MULTITHREADED BFS

We designed the BFS kernel for large-scale graphs that would be too large to store locally on the FPGA. Since memory requests incur long latencies, we use the MT-FPGA approach to mask latency and utilize the available bandwidth. The implementation also uses custom CAMs as on chip synchronizing caches to reduce the number of memory requests and redundant jobs. We accomplish this by allowing the kernels to merge any requests to the same node into a single job. This is an important optimization for handling nodes the have multiple shared children.

Figure 4 shows the flow chart of one job through the BFS engine. The left half of Figure 3 shows the in-memory graph representation. Much of the previous work on BFS relies on using a CSR representation for increased memory density. However, we argue that this representation is limiting to the representational power of graphs. This is evident with the recent growth in node-based graph databases and models like the Property Graph Model [25]. Our graphs our formed with the property graph model in mind, thus we use an adjacency-list style representation. It would only require a minor modification

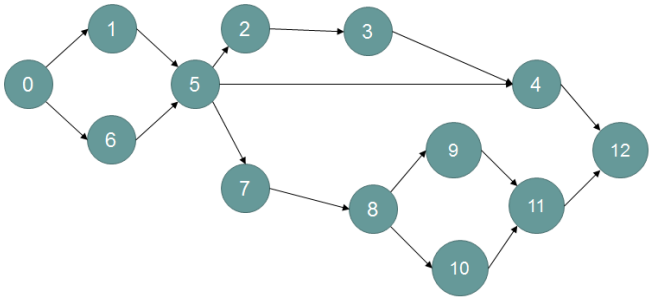

Figure 2 - Example graph for BFS

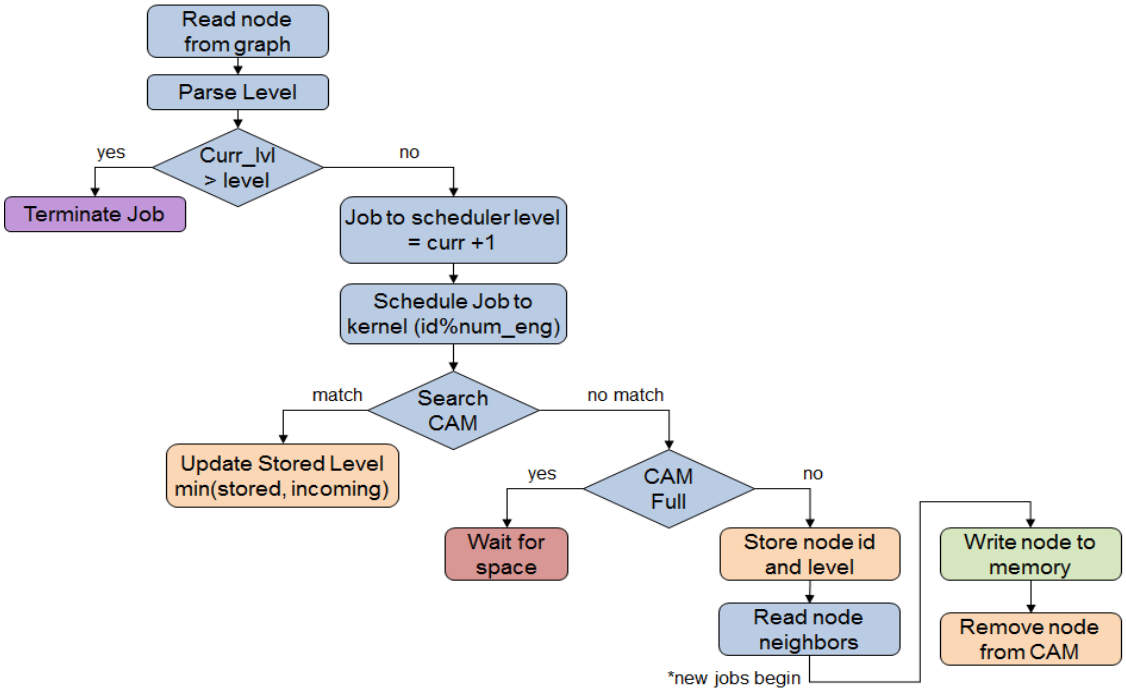

Figure 4 - A flow chart for a BFS job through the kernel

of the kernel to fetch additional key-value pairs from within the node object and support filtered graph searches.

Each edge in the graph is a unique job and assigned a thread on the FPGA. We start by initializing the engine with the start node in the graph. Following the example graph in Figure 2 that would be node 0. The software hand-off to the FPGA initializes the start id to 0 and sets the level to 0. The scheduler assigns the node to its designated kernel (0 in this case) by doing the modulus of the number of engines. Since this is in binary and the number of engines is a power of two, modulus is simply a bitwise-and.

The kernel starts by adding an entry into its CAM for (id:0, level:1). It then requests the node data from memory. This will bring the stored level, count of neighbors, and the pointer to the neighbor list. Once all requests have been made, we set the level for the arriving nodes (node 0's level + 1) in a queue and the thread can write node 0's level to memory and terminate.

Following the graph, the requests that node 0 made to memory would return nodes 1 and 6. Those return to the scheduler with the updated level information from node 0 and are scheduled to their respective engines. The same process happens for those nodes in parallel, and both request node 5. Eventually, both requests for node 5 are scheduled into kernel 5, where the jobs will merge in the internal CAM.

One of the general limitations of a top-down approach to BFS is the chance of enumerating multiple redundant edges as nodes can have multiple common siblings. Using CAMs in this way can compress each of those possible jobs into a single job. This provides a mechanism to synchronize these threads on the FPGA instead of having to block on each thread finishing a level and writing out to memory. As long latency requests to memory are costly, so is synchronizing in memory.

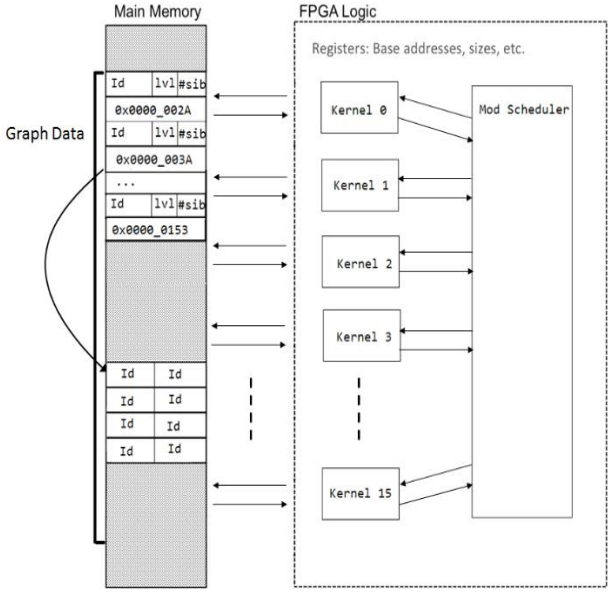

Figure 3 - FPGA BFS engine

V. Implementation and Results

In this paper, we implemented the FPGA designs for the Convey HC-2ex platform; however, the designs are platform independent and only require in-order responses to memory

# Kernels	Registers	LUTs	BRAMs
1	6,595 (0.7%)	12,790 (2%)	3 (0.1%)
4	27,011 (2%)	56, 673 (11%)	10 (1%)
16	104,776 (11%)	216,215 (45%)	40 (5%)

Table 1- FPGA Resource Utilization

requests to port to other platforms. The HC-2ex provides 4 Virtex 6 FPGAs with 16 independent memory channels connected through a full cross-bar with a theoretical peak bandwidth of 20 GB/s. Table 1 shows the resource utilization (registers, LUTs and BRAMs used) of the design using 4 kernels. Peak performance is dependent on the number of kernels used and the clock frequency. Since the kernel runs at a 150 MHz system clock, assuming no stalls, using 4 kernels and taking into account our 2 word node representation would provide a throughput of 300 MNodes/s per FPGA. Since each kernel occupies only one memory channel, this design can scale to 16 independent kernels per FPGA, giving a theoretical peak of 1.2 GNodes/s per FPGA.

The right portion of Figure 3 shows the layout and memory channels of the BFS engine. Our implementation uses a master scheduler to dictate the work of each kernel. This allows the kernels to work independently and not individually coordinate

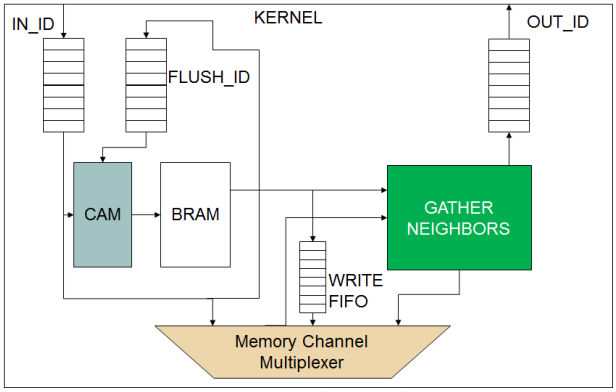

Figure 5 - Kernel FPGA design

with each other. The kernel design is comprised of several FIFOs for coordination, a CAM, a BRAM, and a custom unit we call GATHER_NEIGHBORS. A job enters the kernel and first checks the CAM if it can merge and terminate. Otherwise, it continues in the kernel fetching node information from memory. Once the memory responds, the kernel flushes the job from the CAM, checks if the node has already been visited, and either terminates the job or sends the neighbor list address to GATHER_NEIGHBORS and writes the nodes level value to memory. GATHER_NEIGHBORS, issues requests for all the ids in the neighbor list and sends the responses back to the scheduler with the previous nodes level plus one. The layout can be seen in Figure 5.

VI. Conclusion

In this paper, we have motivated the use of CAMs as synchronizing caches for irregular applications. In conjunction with using a highly multithreaded datapath to support hundreds of threads masking memory latency, the CAMs can take advantage of this long latency to merge jobs and reduce the number of memory requests overall. We demonstrated this design using a breadth first search through a graph, using a graph representation that could easily expand to support the property graph model and richly annotated graphs. This work showed the resource usage and estimated performance of a few engines on a single FPGA, and showed that it could scale to using all the FPGAs on the Convey HC-2ex to completely utilize all available memory bandwidth and provide good throughput for graph-based algorithms. We are currently looking at how we could expand this technique to improve the throughput of a bottom up approach to BFS.

Acknowledgment

This work was supported in part by NSF Grants CCF-1219180 and NSF IIS-1161997.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith, "The Tera Computer System," in *4th Int. Conf. on Supercomputing, ICS '90*, New York, NY, 1990.

[2] R. Jordan and P. Golla, "T4: A Highly Threaded Server-on-a-Chip with Native Support for Heterogeneous Computing," in *Hot Chips*, 2011.

[3] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo and B. Koblenz, "Multi-processor Performance on the Tera MTA," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 1998.

[4] R. J. Halstead, J. R. Villarreal and W. A. Najjar, "Compiling irregular applications for reconfigurable systems," *Int. J. of High-Performance Computing and Networking* , vol. 7, no. 4, pp. 258-268, 2014.

[5] R. J. Halstead, I. Absalyamov, W. A. Najjar and V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins," *Conference on Innovative Data Systems Research* , 2015.

[6] R. J. Halstead, "Using Multithreaded Techniques to Mask Memory Latency on FPGA Accelerators," 2015.

[7] A. Morari, V. Castellana, D. Haglin, J. Feo, J. Weaver, A. Tumeo and O. Villa, "Accelerating semantic graph databases on commodity clusters," in *Big Data, 2013 IEEE International Conference on* , 2013.

[8] A. Morari, V. Castellana, O. Villa, A. Tumeo, J. Weaver, D. Haglin, S. Choudhury and J. Feo, "Scaling Semantic Graph Databases in Size and Performance," *IEEE Micro,* vol. 34, no. 4, pp. 16-26, July 2014.

[9] 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. Proceedings of the ACM/IEEE SC 2005 Conference* , 2005.

[10] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing* , 2007.

[11] D. Scarpazza, O. Villa and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," *Parallel and Distributed Systems, IEEE Transactions on* , vol. 19, no. 10, pp. 1381-1395, October 2008.

[12] K. Wadleigh, J. Amelio, K. Collins and G. Edwards, "Hybrid Breadth First Search Implementation for Hybrid-Core Computers},," in *High*

Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, Piscataway, NJ, USA, 2012.

[13] O. G. Attia, T. Johnson, K. Townsend, P. Jones and J. Zambreno, "CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, Washington, DC, USA, 2014.

[14] 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 International Conference on*, 2012.

[15] J. D. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," *Computing in Science and Engineering,* vol. 12, no. 6, pp. 80-87, November 2010.

[16] S. Hong, S. K. Kim, T. Oguntebi and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," *SIGPLAN Notes,* vol. 46, no. 8, pp. 267-276, February 2011.

[17] J. Zhong and B. He, "An Overview of Medusa: Simplified Graph Processing on GPUs," *SIGPLAN Notes,* vol. 47, no. 8, pp. 283-284, February 2012.

[18] F. Khorasani, K. Vora, R. Gupta and L. N. Bhuyan, "CuSha: Vertex-centric Graph Processing on GPUs," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, New York, NY, USA, 2014.

[19] 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*, Washington, DC, USA, 2010.

[20] S. Beamer, K. Asanović and D. Patterson, "Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-First Search Implementation for Graph500," 2011.

[21] S. Hong, S. K. Kim, T. Oguntebi and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," *SIGPLAN,* vol. 46, no. 8, pp. 267-276, feb 2011.

[22] D. Merrill, M. Garland and A. Grimshaw, "Scalable GPU Graph Traversal," *SIGPLAN,* vol. 47, no. 8, pp. 117-128, February 2012.

[23] S. Beamer, A. Buluc, K. Asanovic and D. Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, Washington, DC, USA, 2013.

[24] U. Dhawan and A. Dehon, "Area-Efficient Near-Associative Memories on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.,* vol. 7, no. 4, pp. 30:1-30:22, jan 2015.

[25] tinkerpop, "Property Graph Model · tinkerpop/blueprints Wiki · GitHub," [Online]. Available: https://github.com/tinkerpop/blueprints/wiki/property-graph-model. [Accessed 2 August 2015].

[26] i. 2. Xilinx, "Parameterizable Content-Addressable Memory," Xilinx, Inc., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf. [Accessed 23 July 2015].

[27] J. Zhong and B. He, "An Overview of Medusa: Simplified Graph Processing on GPUs," *SIGPLAN,* vol. 47, no. 8, pp. 283-284, February 2012.

[28] 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 International Conference on* , 2011.

[29] D. Merrill, M. Garland and A. Grimshaw, "Scalable GPU Graph Traversal," *SIGPLAN Notes,* vol. 47, no. 8, pp. 117-128, February 2012.

[30] 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 International Conference on* , 2010.