GraphGen for CoRAM: Graph Computation on FPGAs

Gabriel Weisz
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
gweisz@cs.cmu.edu

Eriko Nurvitadhi
Intel Science and Technology Center
Intel Corporation
Pittsburgh, PA, USA
eriko.nurvitadhi@intel.com

James C. Hoe
Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, USA
jhoe@ece.cmu.edu

ABSTRACT

This paper presents a system for executing graph computations on FPGAs. It implements an optimizing FPGA backend for the GraphGen graph algorithm compiler [12] and uses the CoRAM prototype implementation [6] to support FPGAs from both Xilinx and Altera. High performance in the generated implementations is achieved through a combination of data transfer optimizations and utilizing the hardware flexibility available in FPGAs. Evaluation results are presented for Tree-reweighted Message Passing (TRW-S), an algorithm for 3d reconstruction, running on the Xilinx ML605 and Altera DE4 FPGA boards. Our implementation for the DE4 performs 14.5× faster than our best effort for a high end CPU, 10.3× faster than our best effort for a high end GPU, and is only 2.6× slower than a hand made implementation running on a high end Convey HC-1 FPGA computing system.

1. INTRODUCTION

Graph computations are the basis of many applications in machine learning and data mining. Graphs are a powerful abstraction that allow relationships between data elements to be easily expressed, allowing the detection of unrelated computations that may proceed in parallel. Graph computations often perform the same sequence of operations on many graph elements, which suggests that they are amenable to acceleration by parallel hardware, either in a SIMD-like fashion by GPUs, or via custom pipelines in FPGAs or ASICs.

Existing frameworks for executing graph computations are geared towards general purpose computers or clusters of general purpose computers. The literature also includes graph computations on accelerators: manual implementations that are not based on reusable frameworks, and a framework for collecting graph statistics on immutable graphs rather than updating graph data (see Section 6).

The GraphGen compiler [12] converts graph algorithms, expressed using a vertex-centric graph specification used by existing graph frameworks, into an intermediate Graph Execution Model (GEM) program suitable for backend realization on accelerators. This paper describes a backend that produces an optimized implementation of the GEM program for FPGAs. The backend is layered on top of a prototype implementation of the CoRAM abstraction for FPGA computing [6], which supports several FPGA boards (see Section 2.3). A separate ongoing work focuses on a GEM backend for GPUs.

The remainder of this paper is organized as follows: Section 2 presents background information. Section 3 discusses the implementation of the system and various optimizations that can be applied. Sections 4 and 5 present experimental methodology and results. Section 6 discusses related work, and Section 7 presents conclusions and future work.

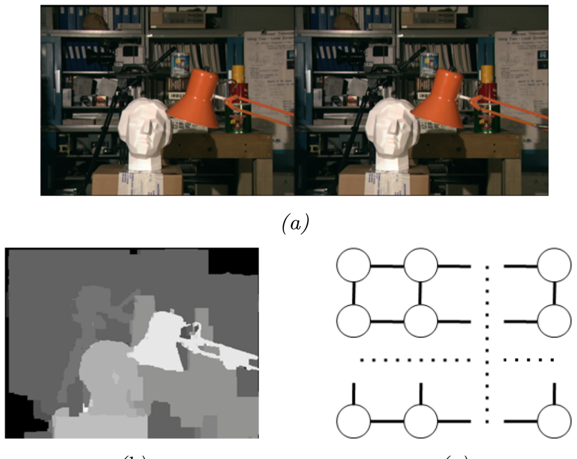


Figure 1: 3d reconstruction: (a) Left and right images; (b) Desired disparity map; (c) Grid graph structure

2. BACKGROUND

2.1 Graph Computation

This work focuses on computations that have a natural mapping to a graph structure. Figure 1 illustrates one such computation: 3d reconstruction [3]. The goal of 3d reconstruction is to recover the three dimensional structure from two stereo images (1a) by creating a depth map (1b) - an image showing the depth at every corresponding pair of pixels in the source images.

This computation can be mapped to a grid graph structure (1c): each pixel is mapped to a vertex, and an edge is created between adjacent pixels. Section 5 discusses experiments with a 3d reconstruction algorithm that uses this mapping. This particular algorithm uses a regular grid structure, but a regular grid structure is not a general requirement for the GraphGen compiler.

2.2 The GraphGen Compiler

The GraphGen compiler allows application designers to focus on the area of their expertise – graph applications – rather than the intricacies of a particular hardware plat-

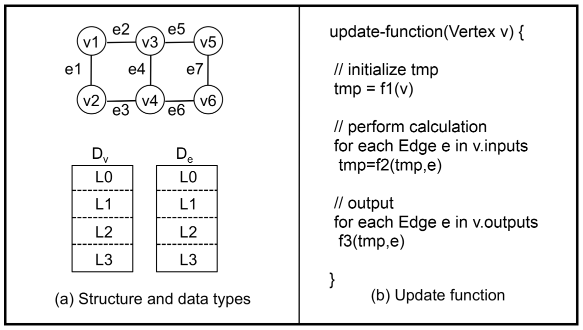
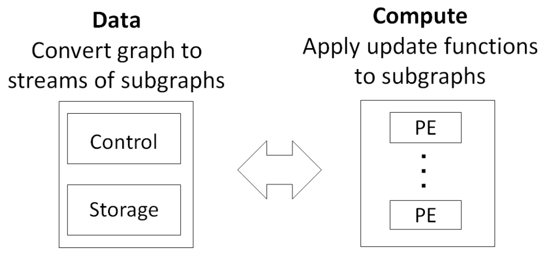


Figure 2: A graph specification



Data
Convert graph to streams of subgraphs
Compute
Apply update functions to subgraphs
Control
Storage
PE
PE

Figure 3: The decoupled GraphGen system architecture

form. The application designer provides a vertex-centric graph specification, illustrated by Figure 2, that includes:

- The graph structure, consisting of vertices and edges.
- Data types for vertex and edge data.
- The update function definition.
- Scheduling hints, such as an evaluation order.

While the application designer provides the graph structure to the GraphGen compiler, the GEM program produced by GraphGen can be utilized by many instances of the same graph. For example, the 3d reconstruction algorithm used in Section 5 can be applied to any pair of images that match the size of the graph specification.

The GraphGen compiler breaks the graph, which generally is too large to fit in local device memory, into subgraphs. In doing so, it compiles the application to a Graph Execution Model (GEM) program. Each subgraph in a GEM program contains vertices and edges, and a "PE Program," which consists of an inlined implementation of the update function for every vertex of the subgraph.

Subgraphs are processed according to scheduling directives provided by the graph specification and the dependencies present in the graph. The graph in Figure 2 can be broken into three subgraphs: {`v1,v2, e1, e2, e3`}, {`v3, v4, e2, e4, e5, e6`}, and {`v5, v6, e5, e6, e7`}. Since there are no edges directly connecting the first group and the third group, they may be processed in parallel, unless the graph specification indicates otherwise.

The subgraphs within the GEM program are intended to be processed by a decoupled architecture shown in Figure 3. Conceptually, this architecture contains a data partition that provides control flow and manages data transport between global and local memory, and a compute partition that includes one or more Processing Elements (PEs).

Each specific implementation of this decoupled architecture is known as as a "template," and serves as a backend for the GEM compiler. The backend is analagous to an OpenCL backend, in that it converts the GEM program into a format usable by the device, and performs device specific optimizations in the process. Among the optimizations are transformations to make the best use of available DRAM bandwidth by reordering and combining data transfers, and arranging data transfers in order to overlap computation with communication. These optimizations need to be run once for any particular graph specification, and can be used with many different instances of the graph.

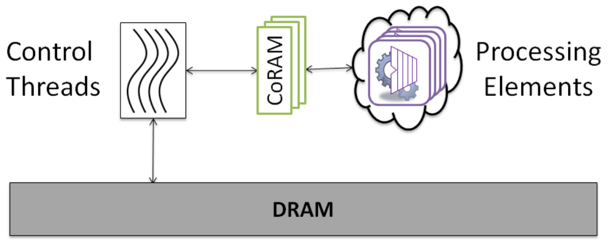


Figure 4: The decoupled CoRAM abstraction

When targeting the FPGA template, an application developer must also provide an RTL implementation of computation components (such as f1, f2, and f3 in Figure 2), which may be produced through any design flow as long as it adheres to a well-defined interface. For example, the PE may be an existing hardware block, or it may be generated by a C-to-gates tool such as ROCCC [20]. The PE is combined with template components to produce an FPGA bitstream.

2.3 The CoRAM Abstraction

The decoupled architecture targeted by the GraphGen is a natural fit for the CoRAM abstraction. The CoRAM abstraction [5] is a decoupled abstraction for FPGA computing. Figure 4 presents a high level diagram of the CoRAM abstraction, which provides a portable, easy to use mechanism for specifying data transfers that targets multiple FPGA devices.

The CoRAM abstraction achieves this goal by defining special memory blocks, called "CoRAM buffers", that serve as a portal between application logic and the outside world. CoRAM buffers are similar to the block memories that exist in current FPGA devices.

The CoRAM abstraction also defines "Control Threads," which are multithreaded programs written in a C-like language. Control threads coordinate data movements between external memory and CoRAM buffers, and can be executed by an embedded microcontroller or compiled to state machines. Since they are meant to express bulk data flows and not computation, control threads are not on the critical path for performance. Libraries provide "personalities" implementing FIFOs, caches, and other structures.

While the CoRAM abstraction is intended to be supported natively by future FPGAs, a prototype implementation implements the CoRAM abstraction in soft logic [6]. The CoRAM prototype currently supports the Xilinx ML605 platform and the Terasic DE4 platform, which contains an Altera FPGA. The prototype also includes interfaces to transfer data between a host system and device DRAM and hardware counters to capture performance metrics.

3. DESIGN AND IMPLEMENTATION

A GraphGen template must provide everything necessary to map a GEM program to an accelerator. For an FPGA template, this includes files necessary to produce a device bitstream (given a PE as described in Section 2.2), and a way to perform the computation and retrieve results. The

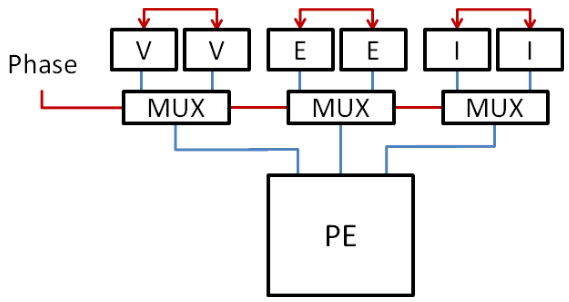


Figure 5: Double buffered template

GraphGen template for FPGAs is comprised of three main components:

1. The Hardware Template instantiates CoRAM buffers, a PE, and logic to connect them. This implements the compute partition of the GraphGen template.
2. The Control Thread coordinates execution, synchronization, and data transfers. This implements the data partition of the GraphGen template.
3. The Preprocessing Script converts a GEM program into a format suitable for use with the template, and performs various optimizations.

3.1 The Hardware template

The Hardware Template instantiates CoRAM buffers that enable data transport, the PE, and logic to connect them and sequence computation. There are several ways that data management can be implemented:

1. Separate CoRAM buffers can each store vertex, edge, or PE program data.
2. CoRAM buffers configured as FIFOs can stream input and output data, and populate memories that serve the PE.

The hardware template uses the second option, which simplifies the hardware design by replacing generic data steering hardware used by CoRAM with a custom datapath, and simplifies the implementation of the optimizations described in Section 3.4.

3.2 The Control Thread

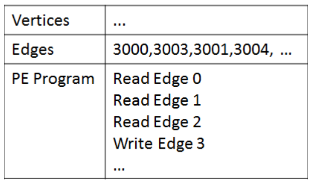
The Control Thread manages overall execution and coordinates data transfers. It iterates through the set of subgraphs specified by the memory image, sending each subgraph from DRAM to the hardware template. Once data has been sent to the hardware template, the control thread sends a signal to the hardware template indicating that computation can begin. When computation has completed, the control thread writes modified data back to DRAM.

3.3 The Preprocessing Script

The Preprocessing Script converts the subgraph data produced by the GraphGen compiler into an appropriate format for the FPGA, and performs optimizations on the data stream to effectively utilize DRAM bandwidth.

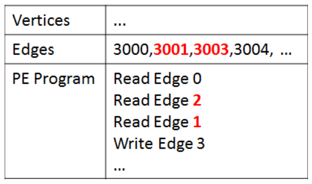
3.4 Optimizations

Several optimizations can be applied in order to increase performance. Two optimizations, discussed in Sections 3.4.1 and 3.4.2, exist solely within the template, requiring only changes to the hardware template, the control thread, and the preprocessing script. These optimizations are concerned with making data transfers as efficient as possible in order



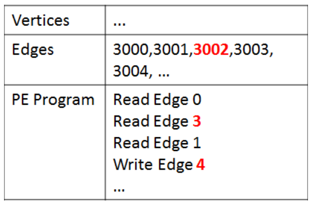
Vertices	...
Edges	3000,3003,3001,3004, ...
PE Program	Read Edge 0 Read Edge 1 Read Edge 2 Write Edge 3 ...

(a) Original subgraph



Vertices	...
Edges	3000,3001,3003,3004, ...
PE Program	Read Edge 0 Read Edge 2 Read Edge 1 Write Edge 3 ...

(b) Sorted subgraph



Vertices	...
Edges	3000,3001,3002,3003, 3004, ...
PE Program	Read Edge 0 Read Edge 3 Read Edge 1 Write Edge 4 ...

(c) Sorted and filled subgraph

Figure 6: Sorting and filling enhancements to coalescing

to fully utilize the PE. Other optimizations, described in Sections 3.4.3 and 3.4.4, change the instruction stream to reduce the number of compute cycles. These optimizations also require the support of the PE, the GraphGen compiler, and the GEM programming model.

3.4.1 *Double Buffering*

One strategy for maximizing computational throughput is overlapping communication with computation, which prevents the PE from stalling while data is being transferred. Double Buffering - instantiating a second set of buffers to hold new data while computation uses data in the first buffer - is commonly used to achieve this goal.

Figure 5 shows how double buffering is implemented. In the diagram, "V" blocks store vertex data, "E" blocks store edge data, and "I" blocks store the PE program. Since the computation is modifying data, it is important to detect read-after-write hazards - situations in which data is written by one phase of execution and read by a subsequent phase - in order to ensure that the calculation is correct. The static data flow in a GEM program removes the need for hazard detection hardware, as hazards can be detected at prepreprocessing time.

3.4.2 *Coalescing*

Coalescing is the practice of combining multiple transfers of small items into fewer large transfers, which allows the fixed cost of data transfers to be amortized. Reducing the total number of data transfers is especially important as indirect transfers are required to support writeable data that exists in multiple subgraphs.

Coalescing requires embedding a token within the data stream to indicate coalesced data. However, otherwise unused space in the vertex and edge list is used to indicate run lengths, making coalescing free in terms of storage overhead. This change requires minimal changes to the control thread. Several coalescing variations are supported:

1. Coalescing requests for items that happen to be adjacent in the vertex or edge list.
2. Sorting items and then coalescing adjacent items.
3. Filling in small gaps in the vertex and edge lists prior to sorting and coalescing.

The second and third coalescing mechanisms move data within the vertex and edge buffers, which can only be done correctly if the preprocessing script has access to the instruction stream. The preprocessing script detects which vertices and edges are moved, and rewrites the instruction stream.

Figure 6 provides an illustration of this process. Subfigure 6a contains a portion of the original subgraph. Note that vertex and edge addresses within the buffer are relative to the local buffer rather than global vertex and edge lists. Subfigure 6b shows the subgraph after sorting and rewriting instructions. Finally, Subfigure 6c shows the subgraph after sorting, filling in the gap, and rewriting. Note that the actual preprocessing script only rewrites instructions once, after both sorting and filling in gaps.

3.4.3 Pipeline Parallelism

Achieving high clock rates in an FPGA requires pipelining the PE - dividing the execution of the computation into multiple clock cycles - in order to reduce the number of operations that occur within each cycle.

The GraphGen compiler can take advantage of a pipelined PE by scheduling computation so that independent instructions are active simultaneously within the pipeline. This optimization can reduce compute time by increasing pipeline occupancy.

When implementing pipeline parallelism, it is important that the execution schedule and PE ensure that no read-after-write data hazards occur - a situation in which an instruction reads a value that has not yet been written by an in-flight instruction. This requires either including bypass logic within the pipeline, or by scheduling instructions to avoid data hazards.

The GraphGen compiler is able create such an execution schedule. This allows the PE to omit bypass logic, which simplifies the PE, but requires that each subgraph contain sufficient independent instruction streams in order maximize performance.

3.4.4 Multiple Read Ports

Graph computations may use multiple data items for each computation. If the PE programming model only allows one vertex and edge to be specified per instruction, then it contains an inherent inefficiency that will reduce system performance.

The PE programming model has been extended to allow multiple edge reads per cycle in order to alleviate this issue, and associated updates have been made to the GraphGen compiler. The programming model may be extended to support multiple vertex reads in the future.

4. EVALUATION METHODOLOGY

Experiments were performed on two hardware platforms - the Xilinx ML605, containing a Xilinx Virtex-6 FPGA, and the Terasic DE4, which contains an Altera Stratix-IV FPGA. The relevant parameters of each hardware system are described in Table 1. The bitstreams for the Xilinx board used 55% of slices, and 87% of block memory, while the Altera bitstreams had 69% logic utilization (29% of ALUTS), and 48% of block memory. Approximately 20% of the utilization was due to the CoRAM abstraction. An implementation that did not use the CoRAM abstraction would also incur non-trivial overhead for DRAM interfaces and data transfer coordination.

Platform	**Xilinx ML605**	**Terasic DE4**
FPGA	Xilinx Virtex-6 LX240T [22]	Altera Stratix IV EP4SGX530[21]
Logic Cells	241,152	531,200
Block Memory	14,976 Kbit	27,376 Kbit
DSPs	768	1,024
DRAM Bandwidth	6.4 GB/s	2 × 6.4 GB/s
DRAM Capacity	512 MB	2 GB

Table 1: Parameters for test platforms.

In these experiments, the chief performance differentiator is the inclusion of an additional DRAM channel on the Altera board, allowing twice the peak DRAM bandwidth. The DE4 also contains sufficient reconfigurable logic that it may be possible to instantiate multiple processing engines on it in the future.

All experiments were performed by programming the device, loading data into DRAM on the board, performing the computation, reading data back from the board, and validating the results against reference results. Experiments run the reconfigurable fabric at 100 MHz except where otherwise indicated.

Hardware counters capture the number of cycles taken by the computation as a whole, including data transfers, and the number of cycles in which the processing engine was active. Run times are reported in wall clock time, starting from when initialization data is in device memory, and ending when output data has been written.

As the GraphGen compiler and associated FPGA template are still under development, experimental results are presented for a single workload. Other workloads execute successfully, but are not yet processed optimally.

5. EXPERIMENTAL RESULTS

3d reconstruction experiments were performed using the Tree Reweighted message passing (TRW-S) [16] algorithm on the ”Tsukuba” images of the Middlebury benchmark[14], which is shown in Figure 1.

TRW-S seeks to convert two stereo images (1a) into a depth map (1b) - an image containing the depth at each corresponding pair of pixels in the source images. Images map to a grid graph in which pixels are vertices and adjacent pixels are connected by edges (1c). The reference images contain approximately 100,000 vertices and 200,000 edges.

Vertex and edge data is represented by a set of 16 labels, which are each 32 bit fixed point values. Subgraphs are sized at 12×64 vertices in order to maximize pipeline parallelism. Experimental results are shown in Figure 7.

The vertical axis shows the running time of the algorithm in milliseconds (lower is better), and the horizontal axis shows three sets of experiments: using one read port for edges, using two read ports for edges, and using four read ports for edges.

The gray line shows the optimal run time of the computation for a 100 MHz system. Optimal is defined as incurring no overhead for data transfers, and was measured by counting the number of cycles that the PE was active.

The red dashed line shows an implementation on either the ML605 or DE4 using one memory controller, running the PE at 100 MHz. Both platforms were measured to differ only slightly in performance with one memory controller. Optimal performance is achieved with one read port, but experiments with two and four read ports are limited by avail-

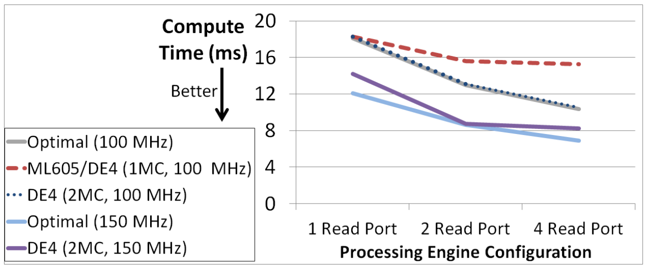


Figure 7: TRW-S results

able DRAM bandwidth. The performance level achieved is consistent with 85% of peak DRAM bandwidth. Since not all data transfers are coalesced - there are still some non-sequential accesses - it is unlikely that a higher data transfer rate could be achieved. Experiments have shown that both platforms can achieve 93-95% of peak DRAM bandwidth using strictly sequential data transfers.

The blue dotted line shows the performance achieved using two memory controllers on the DE4, and running the PE at 100 MHz. Optimal performance is achieved for all experiments, indicating that the performance is compute bound.

Since the DE4 with two memory controllers is compute bound at 100 MHz, experiments were performed on the DE4 with a 150 MHz clock rate for the PE. The light blue line shows the optimal performance numbers for this scenario.

Run times for the DE4 with two memory channels and a 150 MHz PE are shown by the purple line, and which indicates a significant performance improvement over earlier experiments. Optimal performance is achieved with two read ports, but performance barely increases when moving to four read ports. As performance is 1.85× the best achieved with a single memory channel, and the DE4 achieves 79% of peak DRAM bandwidth (comparable to the 85% achieved with a single channel), it is reasonable to conclude that performance is bandwidth limited.

The DE4 performed extremely well as compared to in-house implementations of TRW-S on a high end CPU and GPU: it is about 14.5× faster than a single threaded program on an Intel Core i7, which performed the computation in 120 ms, and 10.3× faster than an NVidia GTX 780, which completed the computation in at 85 ms. GPU performance is limited by the small amount of available local memory, and by difficulty in mapping the diagonal parallelism pattern of the computation to its SIMD engines, which results in significant thread divergence.

The highest performing version of TRW-S that in the literature is in the work of Choi and Rutenbar[4]. They implemented TRW-S on a Convey HC-1 system [1], and achieved approximately 2.65× better performance than the DE4 at 150 MHz, and 4.8× the performance of the ML605.

Some of the performance difference is due to the fact that GraphGen for CoRAM uses a generic framework with an instruction-based PE, rather than a completely custom PE. The Convey implementation read one vertex and four edges every cycle, and wrote two edges per cycle, whereas the GraphGen implementation could only read one vertex or four edges per cycle, or write one edge per cycle.

Performing a cost-to-performance (The Convey HC-1 costs $50k vs $8k for the DE4 and $2k for the ML605) or bandwidth based (The HC-1 has 100 GB/s vs. of bandwidth 12.8 GB/s on the DE4 or 6.4 GB/s for the ML605) reflect favorably on the GraphGen for CoRAM results. It would be possible to implement a GraphGen template for the Convey HC-1, which could fully utilize available DRAM bandwidth and implement multiple PEs on the multiple FPGAs in the Convey HC-1. While some technical challenges would exist to such an implementation, such as hazard avoidance across PEs, these challenges would not be insurmountable.

6. RELATED WORK

Numerous graph-centric paradigms have been investigated, including GraphLab [10] [19], GraphChi[8], Pregel[11], and GreenMarl [7], which all support a vertex-centric abstraction. These systems focus on CPU-based systems or cloud based systems, rather than accelerators such as FPGAs, but they do provide a rich set of techniques for processing graph applications, including machinisms for partitioning graphs into subgraphs, and scheduling subgraph computations.

Most of the works that implement graph centric algorithms on accelerators do so in an ad-hoc manor ([4] [9] [15] [17] [18] [13]), which results in very good performance, but are not part of a platform that can easily be applied to multiple applications.

Betkaoui et. al propose a framework for FPGA acceleration of large graph problems [2]. Their work is intended for use with large graphs, similar to GraphGen for CoRAM, but focuses on parallel computation of statistics on an immutable graph structure. Both frameworks do assume that the graph structure does not change, but GraphGen for CoRAM is designed for algorithms that modify data associated with graph vertices or edges.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented an optimizing FPGA back-end for the GraphGen graph compiler. The primary conclusion that can be drawn from this work is that the data model used by the GraphGen compiler enables a number of optimizations. These optimizations allow FPGAs to perform better than CPUs and GPUs, in spite of running at a much lower clock rate. GraphGen for CoRAM also allows cost effective performance: a $2,000 Xilinx ML605 is only 4.8× slower than a $50,000 Convey HC-1, which is also only 2.6× faster than an $8,000 Altera DE4.

Future enhancements to this project will support better performance and more applications, including:

- Support for multiple PEs.
- Updates to the GraphGen compiler for more control over graph traversal, which will enable better performance on more applications.
- Support more hardware platforms, such as the Convey HC-1.

This work is supported, in part, by the National Science Foundation CCF-1320725 and by the Intel Science and Technology Center in Embedded Computing. We would like to thank Altera, Xilinx, and Bluespec for their generous donation of hardware and tools.

8. REFERENCES

[1] J.D. Bakos. High-Performance Heterogeneous Computing with the Convey HC-1. *Computing in Science Engineering*, 12(6):80 –87, nov.-dec. 2010.

[2] B. Betkaoui, D.B. Thomas, W. Luk, and N. Przulj. A framework for fpga acceleration of large graph

problems: Graphlet counting case study. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8, 2011.

[3] Chao-Chung Cheng, Chung-Te Li, Chia-Kai Liang, Yen-Chieh Lai, and Liang-Gee Chen. Architecture design of stereo matching using belief propagation. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 4109–4112, 2010.

[4] Jungwook Choi and R.A. Rutenbar. Hardware implementation of mrf map inference on an fpga platform. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 209–216, 2012.

[5] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[6] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and evaluation of the CoRAM Memory Architecture for FPGA-Based Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, New York, NY, USA, 2012. ACM.

[7] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[8] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

[9] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and H.H. Chen. Hardware-efficient belief propagation. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(5):525–537, 2011.

[10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[11] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[12] Eriko Nurvitadhi, James C. Weisz, Gabriel Hoe, and Carlos Guestrin. Graphgen: Design compiler for graph computation. *SHAW-4 Workshop, February 2013.*

[13] Sungchan Park, Chao Chen, and Hong Jeong. Vlsi architecture for mrf based stereo matching. In *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation*, SAMOS'07, pages 55–64, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47:7–42, 2002.

[15] D. Strigl, K. Kofler, and S. Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 317–324, 2010.

[16] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(7):787–800, July 2003.

[17] Xueqin Xiang, Mingmin Zhang, Guangxia Li, Yuyong He, and Zhigeng Pan. Real-time stereo matching based on fast belief propagation. *Mach. Vis. Appl.*, 23(6):1219–1227, 2012.

[18] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nistér. Real-time global stereo matching using hierarchical belief propagation. In *BMVC*, pages 989–998. British Machine Vision Association, 2006.

[19] Aapo Kyrola Danny Bickson Carlos Guestrin Yucheng Low, Joseph Gonzalez and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[20] Riverside optimizing compiler for configurable computing. `www.jacquardcomputing.com/roccc/`.

[21] Stratix iv device handbook. `http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdff`.

[22] Virtex-6 family overview. `www.xilinx.com/support/documentation/data_sheets/ds150.pdf`.