GraphGen: An FPGA Framework for Vertex-Centric Graph Computation

Eriko Nurvitadhi[1], Gabriel Weisz[2], Yu Wang[2], Skand Hurkat[3], Marie Nguyen[2], James C. Hoe[2], José F. Martínez[3], Carlos Guestrin[4]

[1]Intel Corporation
Pittsburgh, PA USA
eriko.nurvitadhi@intel.com

[2]Carnegie Mellon University
Pittsburgh, PA USA

[3]Cornell University
Ithaca, NY USA

[4]University of Washington
Seattle, WA USA

Abstract—**Vertex-centric graph computations are widely used in many machine learning and data mining applications that operate on graph data structures. This paper presents GraphGen, a vertex-centric framework that targets FPGA for hardware acceleration of graph computations. GraphGen accepts a vertex-centric graph specification and automatically compiles it onto an application-specific synthesized graph processor and memory system for the target FPGA platform. We report design case studies using GraphGen to implement stereo matching and handwriting recognition graph applications on Terasic DE4 and Xilinx ML605 FPGA boards. Results show up to 14.6x and 2.9x speedups over software on Intel Core i7 CPU for the two applications, respectively.**

Keywords: graph computation, design framework, case studies.

I. Introduction

Computations on graph-based data structures are the basis of many applications in machine learning and data mining, enabling many important capabilities in modern computing (e.g., stereo matching [11], image segmentation [11], handwriting recognition [9], etc). Vertex-centric abstraction [5][7][8] is widely used for capturing such graph-based applications, which have arbitrary graph structures, data types, and graph update functions.

The GraphGen project provides a design framework that automatically compiles a high-level vertex-centric graph specification onto platforms with accelerators (e.g., FPGAs GPGPUs). The goal is to allow application developers without specific platform expertise to take advantage of the performance and energy efficiency of hardware accelerators.

This paper presents the vertex-centric graph specification used by GraphGen for automatic compilation to FPGA platforms. The paper also provides a high-level overview of the GraphGen framework for FPGA targets. The framework automatically compiles a vertex-centric specification onto an application-specific synthesized graph processor on FPGA. The processor uses a memory subsystem designed to handle large graph data structures stored by off-chip DRAMs. The DMA interface to DRAMs is realized using CoRAM [3] in order to flexibly target multiple FPGA platforms. GraphGen also provides simulators and RTL testbenches for validation. Figure 1 illustrates the GraphGen framework.

This paper also reports design case studies that show the flexibility of GraphGen in implementing two applications (stereo matching, handwriting recognition) on two FPGA platforms (Xilinx ML605 and Terasic DE4). The results show that GraphGen implementations are up to 14.6x and 2.9x faster than software on Intel Core i7 CPU for the two applications, respectively.

The rest of the paper is organized as follows. Section II elaborates on GraphGen's vertex-centric specification. Section III provides a high-level overview of the automatic mapping to FPGAs. Section IV reports the design case studies. Section V discusses related work. Section VI offers concluding remarks.

II. GraphGen's Vertex-Centric Specification

A. Review of Vertex-Centric Abstraction

In a vertex-centric specification [5][7][8], graph computation is formulated as a graph G = (V, E, D), where V and E are the vertices and edges of G. An edge e = (u,v) connects two vertices u and v. If the edge is directed, then u is the source and v is the destination. Arbitrary data D can be associated with each vertex, $\{D_v : v \in V\}$, and each edge, $\{D_e : e \in E\}$. The values of D can be updated by the execution, but the structure of G (i.e., V and E) is fixed.

The unit of computation on a vertex is specified as an update-function(v), which is a stateless function that modifies the scope of the vertex v. A scope S_v is the data associated with vertex v and its adjacent edges and vertices. The update-function is executed for each vertex iteratively until a termination condition is met (e.g., desired number of iterations has been reached).

B. Update Function Specification in GraphGen

Unlike existing software frameworks that describe an update function as a software function (e.g., C++ code), GraphGen's specification describes an update function as a composition of custom graph instructions, which are mapped to the graph processor on FPGA, as follows:

- First, custom instructions used in the update function are defined. A user can define custom instructions that compute any arbitrary combinational functions using the scope and temporary data variables as input and output. The temporary data variables are explicitly declared.
- Then, the user provides pipelined RTL implementations

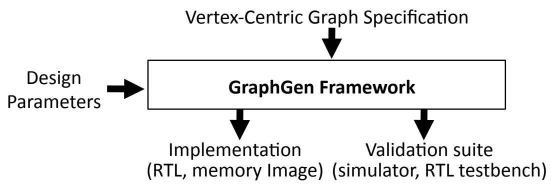

Vertex-Centric Graph Specification
Design Parameters
GraphGen Framework
Implementation (RTL, memory Image)
Validation suite (simulator, RTL testbench)

Figure 1. GraphGen framework overview.

Funding for this work was provided in part by Intel Science and Technology Center for Embedded Computing and NSF SHF-1320725. We thank Jungwook Choi and Prof. Rob A. Rutenbar from University of Illinois for providing their stereo matching hardware for our case study.

of these custom instructions as part of the specification. These implementations are integrated into the graph processor during the compilation process. They must follow the interface declared in the specification. Any hardware design methodology can be used to create these custom instruction implementations. E.g., adapted from existing hardware IPs, manually made from scratch, generated using high-level synthesis tools.

- Finally, the update function is specified as a composition of custom instructions. During compilation, the specification of update function, custom instructions, and graph structure are used to generate a sequence of custom instructions to perform an update function for a given vertex in the graph, i.e., a vertex program.

C. *An Example Specification*

Figure 2(a) shows a simple example of a graph with six vertices (v1 to v6) and seven edges (e1 to e7). Figure 2(b) depicts example data structure definitions for vertex data (D_v) and edge data (D_e). It also shows the data structure definition for temporary data variables used by the update function. This example depicts three 32-bit integers (L0 to L2), but generally it can be any arbitrary structure.

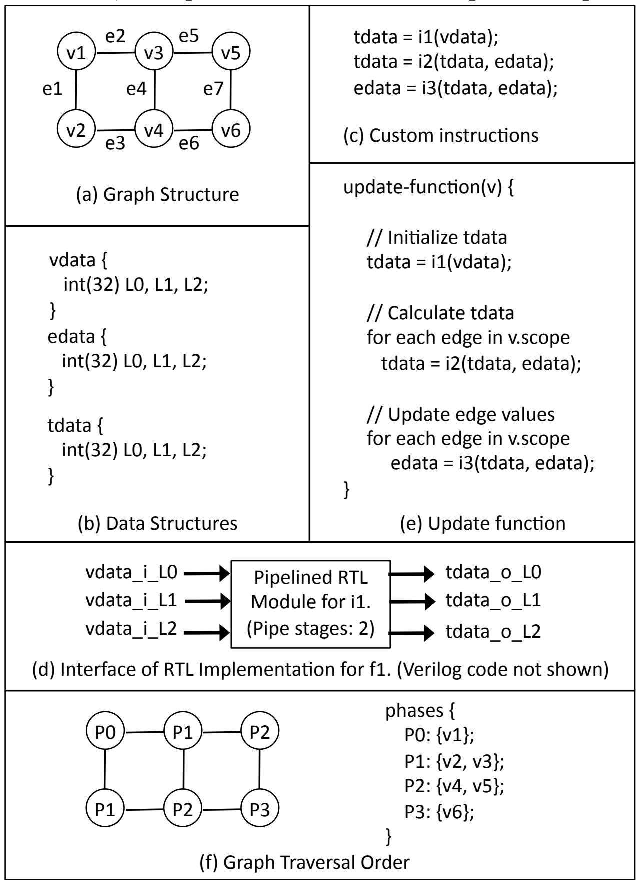
Figure 2(c) shows the declarations of custom instructions named i1, i2, and i3. Pipelined RTL implementations for these instructions are also included as a part of the specification. Figure 2(d) illustrates the RTL module interface for custom instruction i1 (clock and reset signals not shown). It inputs a vertex data and outputs a temporary data value, consistent with the instruction declaration for i1. The module is annotated by the number of pipeline stages that it has. This information is used in compilation.

Figure 2(e) depicts an example update function. It first reads vertex data (vdata) and initializes a temporary variable (tdata) by applying a custom instruction i1. Then, it reads through all the adjacent edges, and uses a custom instruction i2 to calculate a new value for tdata. Finally, adjacent edges are updated by i3 based on input edge data (edata) and tdata.

Figure 2(f) shows an example traversal order from top-left to bottom-right of the graph. The traversal consists of four phases (P0, P1, P2, P3) that need to be executed in order. The vertices within each phase are independent and can be executed in parallel. E.g., the update functions for v2 and v3 in P1 can be executed in parallel.

While relatively simple, this example is representative of many low-level computer vision applications [11], such as the stereo matching application used in our case study.

From the aforementioned specification, the GraphGen compiler can produce a vertex program, which is a sequence of instructions for the graph processor to compute an update function for a given vertex in the graph. Figure 3(a) shows an example program for vertex v3. The first i1 instruction performs initialization to temporary data variables. Then, the for-each loop is elaborated into three i2 instructions operating on the edges (e2, e4, e5) connected to v3. The final three i3 instructions compute the last for-each loop.



```
tdata = i1(vdata);
tdata = i2(tdata, edata);
edata = i3(tdata, edata);
```

(c) Custom instructions

```
vdata {
  int(32) L0, L1, L2;
}
edata {
  int(32) L0, L1, L2;
}

tdata {
  int(32) L0, L1, L2;
}
```

(b) Data Structures

```
update-function(v) {

   // Initialize tdata
   tdata = i1(vdata);

   // Calculate tdata
   for each edge in v.scope
     tdata = i2(tdata, edata);

   // Update edge values
   for each edge in v.scope
       edata = i3(tdata, edata);
}
```

(e) Update function

vdata_i_L0
vdata_i_L1
vdata_i_L2
Pipelined RTL Module for i1. (Pipe stages: 2)
tdata_o_L0
tdata_o_L1
tdata_o_L2
(d) Interface of RTL Implementation for f1. (Verilog code not shown)
P0
P1
P2
P1
P2
P3

```
phases {
   P0: {v1};
   P1: {v2, v3};
   P2: {v4, v5};
   P3: {v6};
}
```

(f) Graph Traversal Order

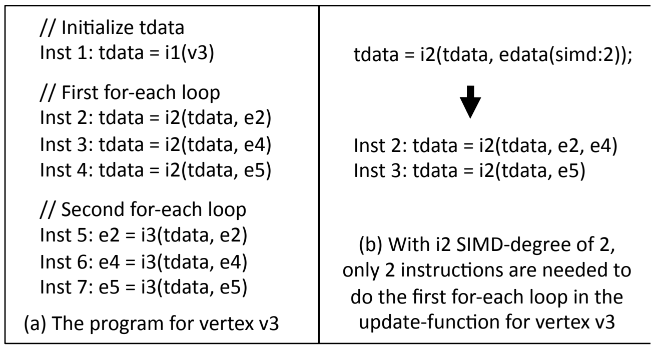
Figure 2. An example vertex-centric graph specification.

D. *Improving Parallelism Using SIMD Graph Instructions*

Since an update function often contains for-each loop operations over the connected edges and/or vertices. There is an opportunity to improve parallelism by using a single instruction that operates on multiple data (SIMD). GraphGen supports such SIMD style custom instructions.

Figure 3(b) shows a SIMD version of the custom instruction i2 from figure 2(c) applied to vertex program for v3 from Figure 3(a). We refer to the number of data processed at a time as SIMD-degree. In this example, two edge data are processed by the i2 instruction. Therefore, one i2 instruction can now process both e2 and e4 edges. As such, the number of instructions needed to compute one for-each loop in the update function is now reduced by one.

The RTL implementation for a SIMD instruction needs to be included as a part of the specification. Its interface will need to incorporate an appropriate number of vertex/edge data inputs/outputs for the instruction's SIMD-degree.



```
// Initialize tdata
Inst 1: tdata = i1(v3)

// First for-each loop
Inst 2: tdata = i2(tdata, e2)
Inst 3: tdata = i2(tdata, e4)
Inst 4: tdata = i2(tdata, e5)

// Second for-each loop
Inst 5: e2 = i3(tdata, e2)
Inst 6: e4 = i3(tdata, e4)
Inst 7: e5 = i3(tdata, e5)
```

(a) The program for vertex v3

```
tdata = i2(tdata, edata(simd:2));
```

$\Downarrow$

```
Inst 2: tdata = i2(tdata, e2, e4)
Inst 3: tdata = i2(tdata, e5)
```

(b) With i2 SIMD-degree of 2, only 2 instructions are needed to do the first for-each loop in the update-function for vertex v3

Figure 3. Example vertex programs for v3.

III. Automatic Mapping to FPGAs

Due to space limitation, we provide only a high-level overview of GraphGen's automated mapping to FPGAs.

A. *Architecture*

The GraphGen framework targets a system architecture depicted in Figure 4. It consists of a graph processor and the memory subsystem, intermediated by scratchpads.

The processor is customizable to integrate user-defined graph instructions provided in the input specification. The processor executes update functions for a set of vertices (i.e., a subgraph) at a time. The subgraph data and vertex programs are stored in the processor's vertex scratchpad (VS), edge scratchpad (ES), and instruction scratchpad (IS), which are implemented using FPGA Block RAMs.

The graph data and vertex programs for the entire graph are stored in external memory (DRAMs). The compiler partitions the input graph into subgraphs and determines the execution schedule for them. The memory system contains a CoRAM-based DMA controller that transfers the subgraphs to/from the processor following the execution schedule.

The processor is a slave to the DMA controller. Once the DMA controller brings a subgraph into the processor's scratchpads, it tells the processor to start execution. When finished, the processor signals the DMA controller indicating it is now idle and ready to execute another subgraph.

To overlap data transfer and computation, the processor uses two sets of scratchpads for double buffering. While it is operating on one set, the DMA controller pre-fetches the next subgraph to execute to the second set.

B. *Compiler*

The key steps taken by GraphGen to compile an FPGA implementation from an input specification are as follows.

First, the graph is partitioned into smaller subgraphs so they can fit onto the processor's scratchpads. GraphGen provides both manual and automatic partitioning capabilities.

After the graph is partitioned, the compiler produces a program for each of the subgraphs. Based on the input update function specification and graph structure, GraphGen first creates program for every vertices. SIMD-degree is considered accordingly (e.g., as in Figure 3(b)). Then, it combines the programs for the vertices in a given subgraph into a subgraph program for execution by the processor.

The next compilation step produces a memory image for the FPGA. Various optimizations are applied to improve the DRAM bandwidth use. Please refer to [12] for details.

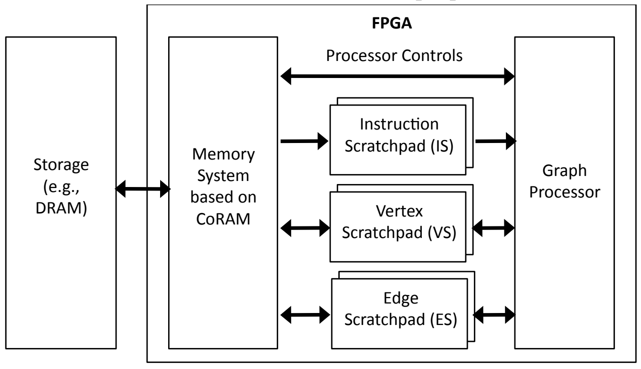


Figure 4. System architecture.

Finally, the compiler generates the synthesizable RTL implementation (Verilog), consisting of the graph processor, scratchpads, and the memory subsystem.

IV. Design Case Studies

We carried out design case studies to evaluate the effectiveness of the GraphGen framework. Experiments were performed on the ML605 and DE4 platforms. The main difference between them is their DRAM bandwidth. DE4 has two DDR channels while ML605 only have one.

To demonstrate the flexibility of GraphGen, we implemented two popular graph applications with widely different attributes, as detailed in Table I. The first application is stereo matching [11], which is illustrated in Figure 5(a). This application accepts a stereo image pair (left and right 2D images) and infers the disparity map containing depth information for each pixel. We used the Tree-Reweighted Message Passing (TRW-S) algorithm, which provides superior inference quality over other alternatives [11]. The second application we studied is handwriting recognition, as shown in Figure 5(b). It accepts an image of a handwritten digit and outputs an inference of what the digit should be. We used Convolutional Neural Network (CNN) [9], a popular algorithm for handwriting recognition.

Using GraphGen, we generated implementations with SIMD-degrees of 1, 2, and 4 for ML605 and DE4 boards. The only exception was CNN for the ML605, which did not have enough routing resources to accommodate SIMD-degree 4. We target clock frequency of 100 MHz for the graph processors on the ML605. For DE4, we could run the graph processor at both 100MHz and 150MHz. All DE4 implementations use 2 DDR channels, except for 150MHz CNN with SIMD-degree 4, which did not meet timing.

Figure 6 depicts the performance for the implementations generated by GraphGen. The y-axis shows performance as runtime for one iteration (i.e., full graph traversal) of the

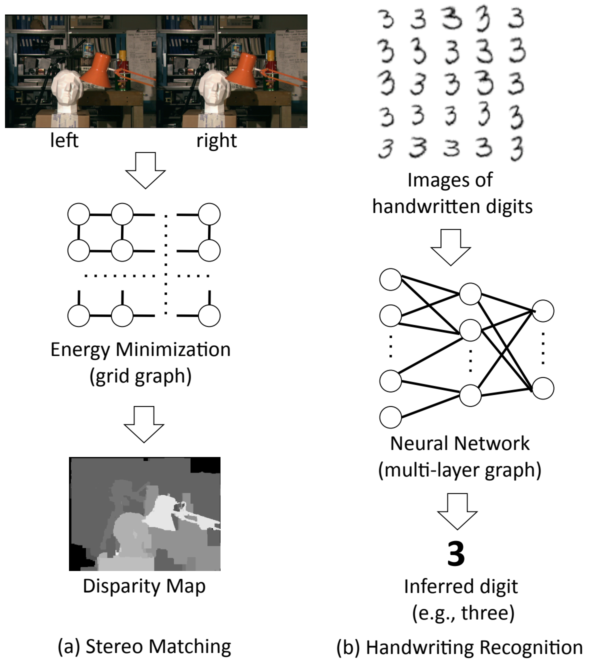

left
right
Energy Minimization (grid graph)
Disparity Map
(a) Stereo Matching
Images of handwritten digits
Neural Network (multi-layer graph)
3
Inferred digit (e.g., three)
(b) Handwriting Recognition

Figure 5. Graph applications under study.

TABLE I. ATTTRIBUTES OF GRAPH APPLICATIONS UNDER STUDY

Applications	**Stereo Matching**	**Handwriting Recognition**
Algorithm	Tree-Reweighted Message Passing (TRW-S)	Convolutional Neural Network (CNN)
CPU software base	Middlebury [11]	CodeProject [9]
Dataset	Tsukuba (384x288 images)	MNIST database (29x29 images)
Graph size	110,592 vertices, 221,184 edges	5,589 vertices, 341,224 edges
Graph shape	Grid, regular	Multi-layer, irregular
Graph traversal	Diagonal	First to last layer
Graph partitioning	Manual (partition to tiles)	Automatic
Custom instruction implementation	Adapted an existing hardware IP from [2]	Manually made
Subgraph size	Tile of 12x64 vertices	Up to 2K vertices and 16K edges
Vertex/edge size	16 x 32-bit	32-bit

algorithm. The x-axis shows the various implementations evaluated. Runtimes are broken down into the time when the processor is active (i.e., Compute) and the time when it is waiting for data to be loaded to its scratchpads (i.e., Data).

The results show that the SIMD optimization helps improve performance, especially in CNN where there are many edges for each node in its graph. Further, for most designs, the processor stalls to wait for data is negligible. This indicates that GraphGen memory optimizations (e.g., double buffering) successfully overlap data transfer with computation. For designs where memory optimizations do not completely hide data transfers (e.g., TRW-S on ML605 with SIMD-degree of 2 and 4), utilizing two memory channels on the DE4 results in further improvement.

We also compared the best performing designs (DE4-150) with software running on 1.87 GHz Intel Core i7 CPU. GraphGen implementations were 14.6x and 2.9x faster for stereo matching and handwriting recognition, respectively. Against hand-made GPU implementations on Nvidia GTX 680m, the speedups were 10.8x and 1.3x.

V. RELATED WORK

Several vertex-centric frameworks, such as GraphLab

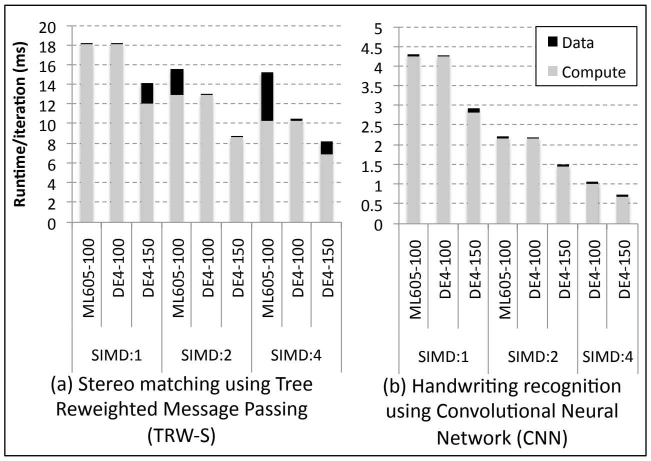


Figure 6. Performance of GraphGen implementations.

[7], GraphChi [5], and Pregel [8], have been widely used in the machine learning community. However, these frameworks are purely based on software, and do not take advantage of FPGA-based acceleration.

In comparison to prior FPGA-based graph computation frameworks [1][4][6], GraphGen is unique because it is the only one that supports all of these features: (1) read and write operation on the graph data, (2) the use of off-chip DRAMs to manage the increasingly large graph dataset in modern graph-based applications, (3) an automated end-to-end compilation flow starting from a high-level vertex-centric specification to an FPGA implementation, and (4) support for multiple FPGA platforms.

VI. CONCLUSION

This paper has presented GraphGen, an FPGA framework for vertex-centric graph computation. The framework accepts a vertex-centric specification and produces an FPGA implementation for the target platform. Design case studies demonstrate that GraphGen is flexible to handle different graph applications targeting different FPGA platforms. They also show that GraphGen implementations achieve significant speedups over software implementations.

REFERENCES

[1] B. Betkaoui, D.B. Thomas, W. Luk, and N. Przulj. A framework for fpga acceleration of large graph problems: Graphlet counting case study. 1nternational Conf. on Field-Programmable Technology, 2011.

[2] J. Choi, R. Rutenbar, "Hardware implementation of MRF map inference on an FPGA platform," Field Programmable Logic and Applications, 2012.

[3] E. S. Chung, J. C. Hoe and K. Mai, "CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing", ACM International Symposium on Field-Programmable Gate Arrays, 2011.

[4] M. deLorimier, et. al., "GraphStep: A System Architecture for Sparse-Graph Algorithms," Field-Programmable Custom Computing Machines, 2006.

[5] A. Kyrola, G. Blelloch, C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC", Symposium on Operating System Design and Implementation (OSDI), 2012.

[6] M. Lin, I. Lebedev, J. Wawrzynek "High-throughput bayesian computing machine with reconfigurable hardware", ACM International Symposium on Field-Programmable Gate Arrays, 2010.

[7] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein, "GraphLab: A New Parallel Framework for Machine Learning", Conference on Uncertainty in Artificial Intelligence, 2010.

[8] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," ACM SIGMOD International Conference on Management of data, 2010.

[9] M. O'Neill, "Neural Network for Recognition of Handwritten Digits," CodeProject.

[10] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, H. Yang, "FPMR: MapReduce Framework on FPGA A Case Study of RankBoost Acceleration", ACM International Symposium on Field-Programmable Gate Arrays, 2010.

[11] R. Szeliski, et. al., "A Comparative Study of Energy Minimization Methods for Markov Random Fields with Smoothness-Based Priors", IEEE Trans. on Pattern Analysis and Machine Intelligence, 2008.

[12] G. Weisz, E. Nurvitadhi, J. Hoe, "GraphGen for CoRAM: Graph Computation on FPGAs", Workshop on the Intersections of Computer Architecture and Reconfigurable Logic, 2013.