GraphGen: An FPGA Framework for Vertex-Centric Graph Computation

***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. It improves over existing software-based frameworks by accelerating graph operations using custom pipelined logic on FPGA. GraphGen accepts a vertex-centric graph specification and automatically compiles it onto an application-specific synthesized graph processor, which is customizable by user-defined graph instructions and utilizes a special-purpose memory subsystem for graph computations. To produce an efficient implementation, GraphGen performs both graph-level and FPGA-level optimizations. In addition to implementation artifacts, GraphGen provides software and RTL models to assist validation activities. Our design case studies demonstrate that the framework is flexible by implementing two graph applications (stereo matching and handwriting recognition) on two different FPGA boards (Terasic DE4 and Xilinx ML605). The FPGA implementations automatically generated by GraphGen are up to 14.6x and 2.9x faster than 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 [15], image segmentation [15], handwriting recognition [12], face detection [5], etc).

Vertex-centric graph computation frameworks, such as GraphLab [8], GraphChi [6], and Pregel [10], have been proposed to implement such graph-based applications. Vertex-centric graph specifications are flexible for capturing various graph-based applications with arbitrary graph structures, data types, and graph update functions. They allow application developers (e.g., machine learning experts) to specify graph computation at a high-level abstraction, without being bogged down by low-level optimization intricacies. Given a graph specification, a framework can automatically generate optimized parallel implementations.

The aforementioned vertex-centric graph computation frameworks map graph computations to software running on general-purpose processors (GPPs). They have not explored execution on FPGA platforms. While harder to program, FPGAs are becoming more widely available (e.g., Convey, IBM Power8, Intel QPI, and HP Moonshot) and can offer significantly better performance and energy efficiency. Especially for algorithms with computationally intensive graph operations (e.g., energy minimization [15], neural network [12][5]), there is an opportunity to significantly accelerate these operations by implementing them on custom FPGA logic instead of software on GPPs.

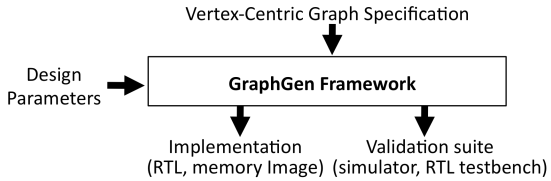

Figure 1. GraphGen framework overview.

There are prior works (e.g., [2][9][13]) that implemented graph computations on FPGA with significant performance and energy improvements over GPPs. However, they have been ad-hoc manual implementations of specific graph algorithms on specific platforms of interest. Thus, they require high development effort and the resulting implementation is not readily portable to other FPGA platforms. One prior work proposed an FPGA-based graph framework [1]. However, it does not support the vertex-centric abstraction and it is limited to only applications where the graph data are read-only.

This paper presents GraphGen, an FPGA framework for vertex-centric graph computation. GraphGen allows application/algorithm programmers without specific platform expertise to take advantage of the performance and energy efficiency of FPGA platforms. GraphGen accepts a vertex-centric graph specification and automatically compiles it to an application-specific synthesized graph processor on FPGA. The processor is customized with user-defined graph instructions, implemented as pipelined custom logic. Its memory subsystem is designed to handle graph data structure. Utilizing the CoRAM technology [3] for realizing the DMA interface between on-chip scratchpad and external DRAM, GraphGen framework can target any FPGA platforms supported by CoRAM. GraphGen also provides simulators and RTL testbenches for automated validation.

This paper also presents design case studies that demonstrate the flexibility of GraphGen in implementing two popular machine learning applications (stereo matching, handwriting recognition) on two different FPGA platforms (Xilinx ML605 and Terasic DE4). The results show that GraphGen implementations (at 150MHz) are up to 14.6x and 2.9x faster than software on 1.87 GHz Intel Core i7 CPU for the two applications, respectively. Comparison against GPUs and prior FPGA implementations are also provided.

The rest of the paper is organized as follows. Section II provides an overview of GraphGen. Section III describes the vertex-centric specification input to GraphGen. The hardware architecture and compilation procedure are detailed in Section IV and V, respectively. Validation capabilities are described in Section VI. Section VII presents the design case studies to evaluate GraphGen. Section VIII discusses related work. Section IX offers concluding remarks.

II. OVERVIEW OF THE GRAPHGEN FRAMEWORK

To the best of our knowledge, GraphGen is the first vertex-centric graph computation framework that targets FPGAs. The framework consists of a customizable synthesized graph processor architecture and a compiler to map a given vertex-centric graph algorithm specification to an FPGA implementation. Figure 1 gives an overview of the GraphGen framework. The development flow is as follows:

1. First, a user creates a vertex-centric specification for the graph computation of interest. He/she also specifies design parameters for the target FPGA implementation, such as the maximum allowable size of subgraphs to be executed by the graph processor. At present, choosing design parameters is done manually, but can be automated in the future by a design explorer system.
2. Based on these inputs, the framework performs compilation to produce a synthesizable Verilog RTL of the system, which is then synthesized and mapped by standard FPGA tools for final implementation. In addition to the RTL designs, the GraphGen compiler also generates a memory image of the executable (graph program, graph structure and graph data).
3. The framework also offers a validation suite to check the correctness of the implementation.
4. Using GraphGen, a user can rapidly generate and evaluate various implementations by modifying input vertex-centric specification and/or design parameters.

The following four sections provide details on the various aspects of the GraphGen framework.

III. VERTEX-CENTRIC GRAPH SPECIFICATION

A. Overview

In a vertex-centric specification [6][8][10], 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).

Thus, a vertex-centric specification of a graph computation contains the following:

- The graph structure, which consists of vertices in the graph, and the edges connecting them.
- The data structures that represent vertex and edge data.
- The update function definition, indicating how to update the scope for a given vertex.
- Graph traversal order, which dictates possible orders of vertices to apply the update function to.

B. Update function Specification in GraphGen

Unlike existing frameworks that describe an update function as a software function (e.g., C++ code), GraphGen describes an update function as a composition of custom graph instructions, which are mapped to the graph processor on FPGA. GraphGen update function is specified 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 of these custom instructions as part of the specification. These implementations are integrated into the graph processor during the compilation process (detailed in Section V). The RTL implementations of the custom instructions 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 IP, 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). The graph structure is expressed textually in GraphGen specification, but illustrated graphically in Figure 2(a) for clarity.

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.

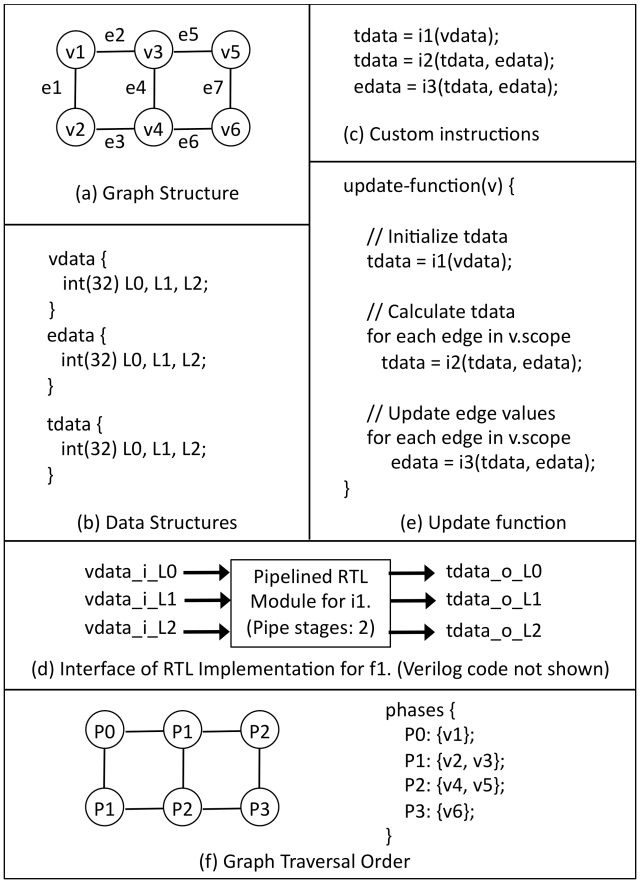

Figure 2. An example vertex-centric graph specification.

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 for brevity). 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 during compilation, which is explained in Section V. In this example, the input goes through two pipeline stages in the module before it leads to the final output. However, since the module is pipelined, it can accept a new input every cycle.

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 [15], 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. Figure 3(b) shows vertex program for v2, containing only five instructions since v2 is connected to two edges (e1, e3).

```
// 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

```
// Initialize tdata
Inst 1: tdata = i1(v2)

// First for-each loop
Inst 2: tdata = i2(tdata, e1)
Inst 3: tdata = i2(tdata, e3)

// Second for-each loop
Inst 4: e1 = i3(tdata, e1)
Inst 5: e3 = i3(tdata, e3)
```

(b) The program for vertex v2

Figure 3. The program for vertices v3 and v2, compiled from specification in Figure 2.

```
tdata = i2(tdata, edata(simd:2));
        ▼
Inst 2: tdata = i2(tdata, e2, e4)
Inst 3: tdata = i2(tdata, e5)
```

(a) 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

```
tdata = i2(tdata, edata(simd:3));
        ▼
Inst 2: tdata = i2(tdata, e2, e4, e5)
```

(b) With i2 SIMD-degree of 3, only 1 instruction is needed.

Figure 4. SIMD instructions can be used to improve parallelism and reduce the number of instructions needed in the program.

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 4(a) 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. Consequently, the number of instructions needed to compute one for-each loop in the update function is now reduced by one. Increasing the SIMD-degree to three further reduces the number of instructions to one, as shown in Figure 4(b).

The RTL implementation for a SIMD instruction needs to be included as a part of the specification. In this case, the interface of the RTL implementation will need to incorporate an appropriate number of vertex and/or edge data inputs and/or outputs as specified by the instruction SIMD-degree.

IV. ARCHITECTURE

GraphGen framework targets a system architecture depicted in Figure 5. 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 DMA controller that transfers the subgraphs to/from the processor following the execution schedule.

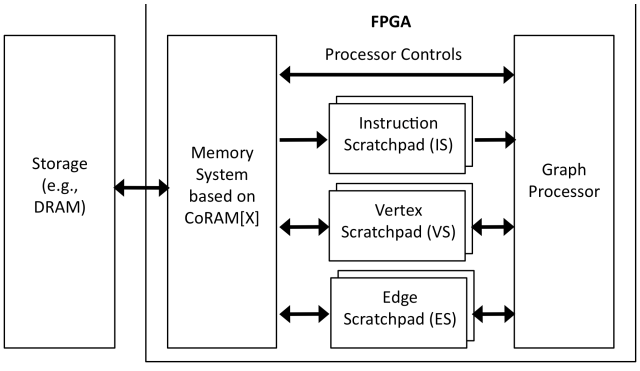

Figure 5. System architecture.

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.

A. Processor Pipeline and Scratchpads

The proposed graph processor uses an in-order pipeline microarchitecture shown in Figure 6. The front-end consists of four stages (F1, F2, D, RD) that fetches the instruction from the instruction scratchpad (IS.rd), decodes it, and read operands from vertex scratchpad (VS.rd), edge scratchpad (ES.rd), and temporary data variables (tdata.rd). Temporary data variables are stored in registers. The Fetch logic interfaces with the program counter (PC) and thread scheduler (Thread) to sequence through the instructions in the IS. The Decode and Unpack logic are generated accordingly based on the input vertex-centric specification. For the example in Figure 2(b), the Unpack logic converts the read data into {L0, L1, L2}.

The back-end stages consist of a customizable number of execution stages (E1 to En), followed by the final stage (WB) that packs (Pack) and writes data back to the vertex scratchpad (VS.wr), edge scratchpad (ES.wr), and temporary data variables (tdata.wr). The execution stages are for the execution unit, which integrates the pipelined RTL implementations of the custom instructions provided in the input specification. The number of stages used by these RTL implementations determines the number of execution stages.

Data widths used by scratchpads, temporary data variable registers, and the corresponding pipeline registers are all customized based on the data structures described in the input specification. Size of the scratchpads is also customizable to accommodate different subgraph sizes.

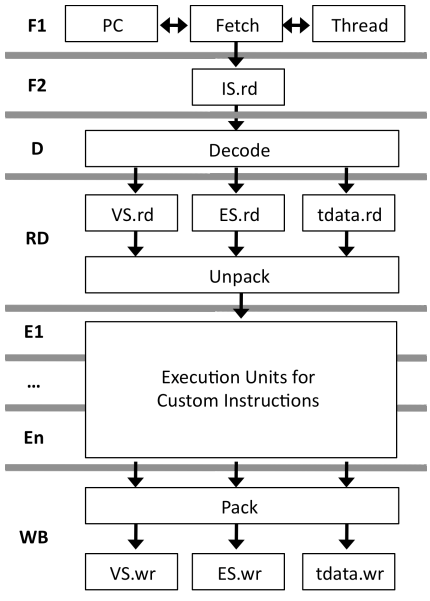

Figure 6. Graph processor pipeline.

The processor pipeline supports multi-threading to improve efficiency. A thread is a sequence of instructions that computes an update function (a vertex program) for a vertex. Different threads of independent update functions from different vertices can be interleaved to improve pipeline utilization (e.g., v2 and v3 update functions in Figure 2).

To simplify design and improve critical path, the pipeline does not include any logic for hazard detection and resolution within one thread. To guarantee that there is no data hazard in the pipeline, the compiler interleaves instruction streams so there can only be a single instruction from any one thread in the pipeline at a given time.

To support SIMD instructions, the scratchpads support customizable data access ports. For the SIMD instruction example in Figure 4(a) that reads two edges, the edge scratchpad is customized to use two read ports. Accordingly, the processor pipeline is customized to use two ES.rd interfaces. While possible to support multiple data writes, our current GraphGen prototype does not support this yet.

B. Memory Subsystem

The memory subsystem stores the architectural states shown in Figure 7. It holds the graph data and an execution schedule described as an ordered list of subgraph descriptors. A subgraph descriptor contains the following information:

- Header information for bookkeeping (e.g., number of vertices and edges in the subgraph).
- Lists of vertices and edges that belong to the subgraph.
- A subgraph program, which is the combination of all programs for the vertices in the subgraph.

The DMA controller goes to each subgraph descriptor in the execution schedule, brings the subgraph (i.e., vertex data, edge data, and subgraph program) into the processor's scratchpads, and tells the processor to start execution.

While the processor is executing, the DMA controller pre-fetches the next subgraph into the second set of scratchpads to overlap computation and data transfer.

Since the pre-fetched subgraph may share some vertices and edges with the current subgraph being executed, the memory system ensures coherency by transferring the most up-to-date results of such shared data across the scratchpad sets paired for double buffering. This transfer is done after the processor finishes executing the current subgraph and before it starts executing the next (pre-fetched) subgraph.

The DMA-based memory subsystem is implemented using CoRAM [3], which can automatically generate a DMA controller implementation for a target FPGA platform from a C-like description. Using CoRAM as a back-end allows GraphGen to target any platform supported by CoRAM.

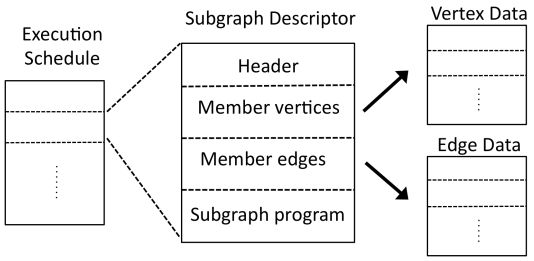

Figure 7. Architectural states stored by memory system.

V. COMPILER

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

Graph Partitioning. First, for a graph application where the dataset does not fit in the FPGA Block RAMs, the graph is partitioned into smaller subgraphs so they can fit onto the processor's scratchpads. The choice of graph partitioning strategy is an open research problem. The most appropriate strategy depends on the graph algorithm and structure. For example, in the stereo matching application we studied, prior work has already suggested tile-based partitioning [7].

As such, GraphGen provides both manual and automatic partitioning capabilities. To partition a graph manually, the user provides a list of subgraphs and their member vertices as input design parameters. Alternatively, the user can set the maximum allowable subgraph size, and GraphGen can automatically partition the graph to subgraphs abiding by the size constraint. The current GraphGen prototype's automatic partitioning strategy is to go over each execution phase in the graph traversal order and form as large subgraph as possible containing independent vertices within each phase. Other automatic graph partitioning strategies is left for future work.

Subgraph Programs. After the graph is partitioned, the next compilation step produces the 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 4 example). Then, it combines the programs for the vertices in a given subgraph into a subgraph program. Each vertex program is associated with a thread in the processor. Instructions from different threads are interleaved to form a subgraph program. The number of threads supported by the pipeline is derived from the pipeline depth, calculated based on the provided RTL implementations of custom instructions. The compiler interleaving optimization attempts to fill all the available thread slots in the processor.

Memory Image. The next compilation step produces a memory image for the FPGA. It first lays out vertex and edge data in a contiguous DRAM spaces. Based on the DRAM layout, it determines the layout of each subgraph data when it is brought into the processor's scratchpads. As detailed in section IV, the DMA controller transfers each subgraph to these scratchpads prior to processing. For optimal use of DRAM bandwidth, the scratchpads should lay out the graph data such that it maps to a contiguous set of data in DRAM so coalesced burst data transfers can be done between DRAM and scratchpads. However, since input graph can be of any arbitrary structure, mapping contiguous set of data in DRAM to processor's scratchpads is not always possible. The compiler attempts to maximize coalescing by re-ordering data placement in the scratchpad and even padding with a small amount of unused data if it results in better coalesced burst transfers. Once data layout in the processor scratchpads is determined, the subgraph programs are finalized to include pre-computed scratchpad indices to point to the appropriate operands in the scratchpad.

RTL Design. The last step of the compilation is to generate the synthesizable RTL implementation (Verilog). This step first produces the graph processor RTL implementation. Then, it is integrated with the CoRAM-based memory subsystem. Finally, CoRAM system [3] is used to produce the final Verilog implementation.

TABLE I. FPGA PLATFORMS USED IN EVALUATION

Platform	Xilinx ML605	Terasic DE4
FPGA	Xilinx Virtex-6 LX240T	Altera Stratix IV EP4SGX530
Logic Cells	241,152	531,200
Block Memory	14,976 Kbit	27,376 Kbit
DSPs	768	1024
DRAM Bandwidth	6.4 GB/s	12.8 GB/s
DRAM Capacity	512 MB	2 GB

VI. VALIDATION

One challenge in an automated framework such as GraphGen is to validate that the generated implementation is functionally equivalent to the given input specification. GraphGen provides facilities to validate correctness across different levels of the framework, as follows:

- **Vertex-level functional software simulator.** This simulator models the behavior of the given vertex-centric input specification. The simulator applies update function to each vertex in the graph one at a time, abiding to the graph traversal order. It is useful in ensuring that the specification behaves as expected, and as a reference to validate lower levels of the framework.
- **Instruction-level software simulator**. This simulator models the execution of subgraph programs, one instruction at a time. It is used to validate that the subgraphs and their programs are correct. In this simulator, an instruction executes in a single step. Pipeline-level details are not modeled. It also models only the functionality of the memory and processor's scratchpads (not cycle accurate).
- **Testbench for validating the generated graph processor RTL implementation**. Validation compares instruction traces generated by the testbench against those produced by the instruction-level simulator.
- **Testbench for the RTL implementation of the entire system.** Validation compares instruction traces and final outputs from higher-level models against those generated by this testbench.

Optionally, debugging hooks and performance counters can be included in the final RTL to support on-FPGA validation.

VII. EVALUATION

We carried out design case studies to evaluate the effectiveness of the GraphGen framework. Experiments were performed on the ML605 and DE4 platforms, detailed in Table I. All experiments were performed by programming the FPGA, loading initial data into on-board DRAM, performing the graph computation, reading final data from the board, and validating the results against reference results.

Hardware performance counters were used to measure runtimes during the graph computation part only.

A. *Graph applications under study*

The first application we studied is stereo matching [15], as shown in Figure 8(a). This application accepts a stereo image pair (left and right 2D images) and infers the disparity map containing depth information for each pixel. E.g., In Figure 8(a), the lamp is inferred to be closer than the statue. This study uses the Tree-Reweighted Message Passing (TRW-S) algorithm for stereo matching, which provides superior inference quality over other alternatives [15].

The second application we studied is handwriting recognition, as shown in Figure 8(b). It accepts an image of a handwritten digit and outputs an inference of what the digit should be. The figure illustrates various possible inputs of handwritten digit "3" from the MNIST database [12]. This study uses Convolutional Neural Network (CNN) [12], which is a popular algorithm for handwriting recognition.

GraphGen is flexible enough to support these two applications, which have widely different attributes, as shown in Table II. TRW-S operates on a regular grid graph where each vertex corresponds to a pixel in the input image, while CNN uses a multi-layer irregular graph where each vertex represents a neuron. Each graph has its own traversal order. TRW-S graph is manually partitioned into tiles as suggested by [7]. The partitioning of CNN graph is not well studied and is left to the auto-partitioning capability of GraphGen. The RTL implementation of the custom instructions for TRW-S is adapted from an existing hardware IP [2]. For CNN, we made the RTL implementation manually. Graph partitioning is set to target subgraphs with sizes that provide sufficient independent vertices to best utilize the processor pipeline. (e.g., TRW-S tile height is set to match the pipeline depth).

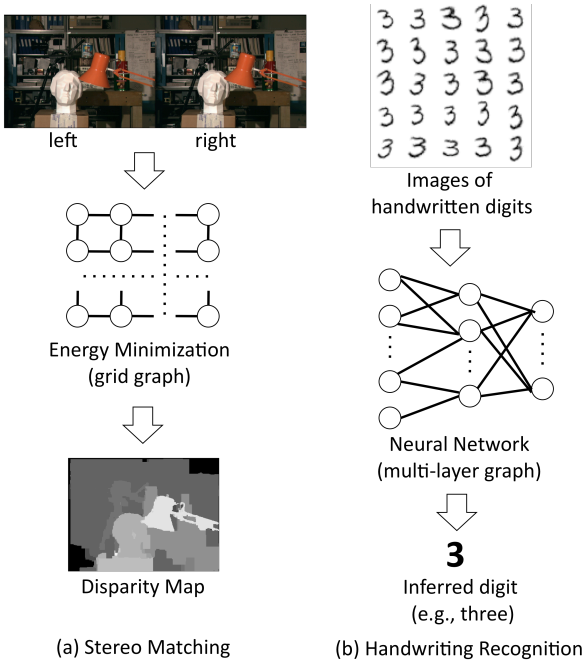

Figure 8. Graph applications under study.

TABLE II. 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 [15]	CodeProject [12]
GPU software base	Made in house	CodeProject [4]
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, as in [7])	Automatic
Subgraph size	Tile of 12x64 vertices	Up to 2K vertices and 16K edges
Vertex/edge size	16 x 32-bit	32-bit
Custom instruction implementation	Adapted an existing hardware IP from [2]	Manually made
Pipeline depth	12	14

B. *Performance of GraphGen generated implementations*

We used GraphGen to generate 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. Ideally, the SIMD-degree should be high enough to match the available parallelism in the graph. For TRW-S, a vertex has only up to 4 edges (i.e., left, right, up, down). So, using SIMD-degree above 4 to process more than 4 edges in parallel will not be useful. For CNN, a vertex can have more than 1000 edges, and can benefit from a higher SIMD-degree. However, we found that our designs are more limited by routing than hardware utilization. This is due to creating large scratchpads, which are replicated in designs with multiple read ports when using SIMD. This makes it difficult to meet timing with SIMD-degree higher than 4. We target clock frequency of 100 MHz for the graph processors on the

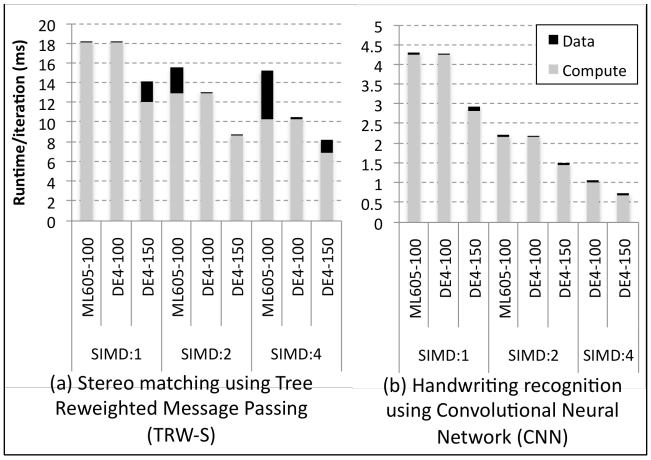

Figure 9. Performance of GraphGen implementations.

ML605. For DE4, we could run the graph processor at both 100MHz and 150MHz. The DE4 implementations also have 2 DDR channels for higher DRAM bandwidth. All DE4 implementations use 2 DDR channels, except for 150MHz CNN with SIMD-degree 4, which did not meet timing.

Figure 9 depicts the performance for the aforementioned implementations generated by GraphGen. The y-axis shows performance as runtime for one iteration of the algorithm. One iteration consists of a full graph traversal (i.e., top-left to bottom-right and back for TRWS; first to last layer for CNN). 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).

SIMD. The results show that the SIMD optimization helps improve performance. For CNN, since almost all of the instructions can take advantage of reading and processing multiple edges, doubling the SIMD degree reduces the instruction count as well as the compute time by almost half.

For TRW-S, only about half of the instructions can take advantage from reading and processing multiple edges. The other half of the instructions could take advantage of processing and updating of multiple edges. However, since GraphGen does not yet support parallel updates, we could not capitalize on this opportunity. Therefore, all together, increasing SIMD-degree only reduces compute time of TRW-S by ~20%, a more modest improvement than in CNN.

Double Buffering. The results also show that for most of the designs, the time processor stalls to wait for data to be loaded to its scratchpads is negligible. This indicates that double buffering successfully overlap data transfer with computation. To achieve this, each subgraph transfer time has to be smaller than or equal to the compute time. The transfer time is affected by the amount of data being transferred and the transfer rate. To improve transfer rate, GraphGen performed optimizations to coalesce memory transfers. The resulting implementations for TRW-S achieved 85% and 79% of peak DRAM bandwidth utilization with one and two DDR channels, respectively. Based on the transfer rate, we chose the appropriate subgraph size to optimally overlap data transfer time with computation time. The CNN implementations are not limited by memory bandwidth since its dataset is much smaller than TRW-S.

DE4 vs. ML605. There are still some designs where double buffering does not completely hide data transfers, such as TRW-S on ML605 with SIMD-degree of 2 and 4. In this case, utilizing two memory channels on the DE4 board to tap into 2x more DRAM bandwidth than ML605 result in improvement in data transfer time, such that processor stalls waiting for data are almost completely eliminated.

Finally, using higher frequency processor in the DE4 implementations lead to reduction in computation time. For some designs (i.e., TRW-S with SIMD-degree of 1 and 4), such shorter compute leads to exposure in data transfer time.

C. Comparison against CPU and GPU implementations

To evaluate the quality of GraphGen implementation, we compared the best design we have (i.e., DE4-150, SIMD:4) against CPU and GPU implementations. Table II shows the CPU and GPU software that we used. The GraphGen implementations use fixed-point instead of floating point values, as suggested by prior work [2][5]. CNN uses a hyperbolic tangent function in its computation, which is approximated in FPGA using a lookup-table approach [11]. We ensure that the CPU, GPU, and FPGA implementations produce similar results (i.e., inference accuracy within 1%).

TABLE III. COMPARISON AGAINST CPU AND GPU

	CPU	**GPU**	**GraphGen**
Platform	Intel® Core i7	Nvidia GTX 680m	Terasic DE4
Frequency	1.87 GHz	719 MHz	150 MHz
Max Power	45 W	100 W	21 W
Stereo Matching using TRW-S			
Performance(iteration/sec)	8.3 (1x)	11.2 (1.3x)	121 (14.6x)
Energy (iteration/joule)	0.2 (1x)	0.1 (0.6x)	5.8 (31.2x)
Handwriting Recognition using CNN			
Performance(iteration/sec)	458 (1x)	1041 (2.3x)	1343 (2.9x)
Energy (iteration/joule)	10.2 (1x)	10.4 (1x)	64 (6.3x)

We evaluated both performance and energy efficiency of each implementation. The performance is represented as iteration per second, and the energy efficiency is estimated by normalizing performance by power consumption (iteration/joule). Due to the difficulty in precisely measuring power consumption, we use vendor provided peak power for each platform. Table III shows the maximum power and operating frequency of the platforms under study.

Comparison results are summarized in Table III. For both applications, GraphGen FPGA implementations outperform both CPU and GPU. For TRW-S, GraphGen is 14.6x faster than CPU and 10.8x faster than GPU. (GPU is only 1.3x faster than CPU for TRW-S). The GPU performance is limited by their small local memory and the inefficiencies when mapping the diagonal graph traversal to SIMD. For CNN, GraphGen FPGA implementation is 2.9x faster than CPU, while GPU is only 2.3x faster. The performance improvement of FPGA and GPU over CPU are both modest because the CNN benchmark is almost ideal for the CPU—the graph data size is small (29x29 images) and fits completely in the CPU's cache.

When the energy efficiency (iteration/joule) is considered, GraphGen's FPGA implementations compare even more favorably to the CPU and GPU implementations. The estimated energy efficiency of GraphGen implementation is 31.2x and 6.3x better than CPU for TRW-S and CNN, respectively. For GPU, energy efficiency is worse than CPU for TRW-S and about the same for CNN.

D. Comparison against Ad HoC FPGA implementations

TRW-S. A recent work [2] has demonstrated the highest performing FPGA implementation of stereo matching using TRW-S. Their implementation was manually developed and uses a high-end Convey HC-1 system. It achieves 2.6x better performance than the best GraphGen implementation (i.e.,

based on their reported runtime running the same Tsukuba stereo image we used in our study). Keep in mind, this is not an apples-to-apples comparison but serves to establish that our generated implementations are comparable with hand-designed implementations. While both implementations run at 150 MHz, the HC-1 has 20GB/s DRAM bandwidth per FPGA whereas the DE4 has only 12.8 GB/s. (Note that HC-1 costs $50K while the DE4 only costs $8K.)

Another reason for the performance difference is because the implementation on HC-1 is specially built for TRW-S. As such, the HC-1 implementation can perform one update function with a single round of access to the edge and vertex data by its processing pipeline (1 cycle per update function). On the other hand, in GraphGen, an update function has to be broken down into multiple instructions to be acceptable by the GraphGen compiler.

CNN. The most relevant prior work that implements CNN on FPGA is the CNP architecture [5], which combines a general purpose soft processor combined with a custom co-processor for CNN. The co-processor is especially designed to perform many multiply-accumulate in parallel, which is the most common operation in CNN.

The result reported in their study is for face detection. It uses CNN algorithm on a much larger graph than one used in our study. They reported an average performance of $3.4x10^9$ edges per second. The highest performing GraphGen design in our study has 7.4x lower throughput. This is because the CNP co-processor has 49 parallel multiply-accumulate units, while our GraphGen implementation (with only one graph processor) only supports 4 parallel multiply-accumulates.

VIII. Related Work

There are several existing vertex-centric graph computation frameworks, such as GraphLab [8], GraphChi [6], and Pregel [10]. These frameworks are based on software and target standard general-purpose processors. Their main focus has been graph computation on very large dataset, such as social network or web graphs. They provide facilities for handling very large graph, including techniques to partition the graph and distribute the computation among many machines. GraphChi targets effective use of disk-stored data to handle very large graph data. GraphGen can complement these systems by providing the capability to map the computation onto FPGA for acceleration.

Many prior works that implement graph computation on FPGAs do so in an ad-hoc manner (e.g., [2][9][13]). While hand-designed implementations result in a very good performance, they are fixed to specific target applications and platforms. On the other hand, GraphGen allows mapping any vertex-centric graph computation onto any FPGA platforms supported by CoRAM.

Prior works have studied several FPGA frameworks, but none of them targets the vertex-centric abstraction. For example, the CNP framework [5] supports convolutional neural networks, while the FPMR [14] supports map-reduce abstraction. The work most relevant to GraphGen is done by Betkaoui et. al., which developed an FPGA framework for large graph problems [1]. However, their framework can only support problems where the graph data is read-only. These are useful for collecting graph statistics, but not useful for applications that modifies data (including stereo matching and handwriting applications used in our study).

IX. 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. It also provides validation models. Design case studies demonstrate that GraphGen is flexible to handle different graph applications targeting different FPGA platforms. They also show that GraphGen implementations are up to 14.6x and 2.9x faster than software on Intel Core i7 CPU for stereo matching and handwriting recognition, respectively.

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] B. Conan, K. Guy, “A Neural Network on GPU,” CodeProject.

[5] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, “CNP: An FPGA-based Processor for Convolutional Networks”, International Conference on Field Programmable Logic and Applications, 2009.

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

[7] C.-K. Liang, C.-C. Cheng, Y.-C. Lai, H. H. Chen, L.-G. Chen, “Hardware-efficient belief propagation”, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

[8] 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.

[9] E. Magdaleno, J. P. Lüke, M. Rodríguez, J. M. Rodríguez-Ramos, “Design of Belief Propagation Based on FPGA for the Multistereo CAFADIS Camera,” Sensors, 2010.

[10] 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.

[11] A. H. Namin, K. Leboeuf, R. Muscedere, W. Huapeng, "Efficient Hardware Implementation of the Hyperbolic Tangent Sigmoid Function", IEEE International Symp. on Circuits and Systems, 2009

[12] M. O’Neill, “Neural Network for Recognition of Handwritten Digits,” CodeProject.

[13] J. M Perez, P. Sanchez, M. Martinez, “High memory throughput FPGA architecture for high-definition belief propagation stereo matching,” International Conference on Signals, Circuits and Systems, 2009.

[14] 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.

[15] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, C. Rother, “A Comparative Study of Energy Minimization Methods for Markov Random Fields with Smoothness-Based Priors”, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2008.