HARDWARE ACCELERATION OF GRAPH ANALYSIS APPLICATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Kunle Oluwatayo Oguntebi, Jr.
June 2016

This dissertation is online at: http://purl.stanford.edu/xp488mf1321

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Subhasish Mitra

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Acknowledgements

The time and energy invested in this research would not have been possible without the prayers and support of my family. My parents, Zacchaeus and Rachel, were constant sources of encouragement even as they wondered when I was ever going to graduate. My siblings (Blessing, Joy, Grace, James) also took time to ensure that I was okay in academics and in life. My most sincere gratitude to all of them.

I would like to thank my advisor and mentor Kunle Olukotun. Kunle showed extreme patience during the slowest points of my research. He maintained a can-do attitude and willingness to explore new territory that inspires me to this day. I hope that my intuition and vision about the state of computing can one day be a fraction of his. Thanks also to Darlene Hadding for her administrative support.

I would like to thank Christos Kozyrakis and Subhashish Mitra for serving on my orals and reading committees. Christos' energy was instrumental in my early years as a graduate student. Prof. Mitra always had an ecouraging smile and word when I visited him for consultation.

I would like to thank my friends and research colleagues from my first full research project, called FARM: Jared Casper, Sungpack Hong, and Nathan Bronson. Despite a very challenging technical problem, all of these guys pushed me technically with their depth and especially their breadth. I would like to also thank the other members of Kunle's group: Lawrence McAfee, Kevin Brown, HyoukJoong Lee, Arvind Sujeeth, Nicole Rodia, Chris A, Chris D, and others who were great lab mates and encouraging people. Special thanks to Lawrence for many discussions in lab about the meaning of life and whether or not it can be represented digitally.

Acknowledgements also go to the National Science Foundation (NSF) graduate

research fellowship for financial support. I would also like to acknowledge the staff and administration of the Stanford University EE department and the university as a whole for their support, patient encouragement, and efforts into improving student life.

I would like to thank countless friends and acquaintances from various groups at Stanford University, both undergraduate and graduate, for their support and love.

Finally, it is my belief that despite the free will of human beings, we can do nothing without the allowance of good God. I humbly give praise and thanks to Him for allowing me to have this opportunity to research at Stanford University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer memories store data using a variety of mechanisms. The most common involve the use of relational databases or indexed file systems. Although these formats provide efficient access and manipulation of structured data, they are less efficient for unstructured data sets, especially ones in which connections between individual data are of first-class importance. Graph data structures are fundamentally designed to capture such relationships efficiently.

Graph data structures are built on primitives that naturally mirror the real world: vertices (or nodes), edges (or relationships) between them, and properties (or attributes) associated with each. A large number of important data sets are easily and usefully expressed in this less structured manner.

The ability of graphs to capture information about relationships makes them easily amenable to a wide variety of data analytics algorithms. These algorithms can yield valuable insight otherwise difficult to extract from traditional data stores such as relational databases.

1.1 Graph Databases

Analytics on graph data are made all the more useful by the increasing prevalence of large data sets, now becoming commonplace in data centers operated by large corporations and research labs. In commercial production, graphs can be used as the

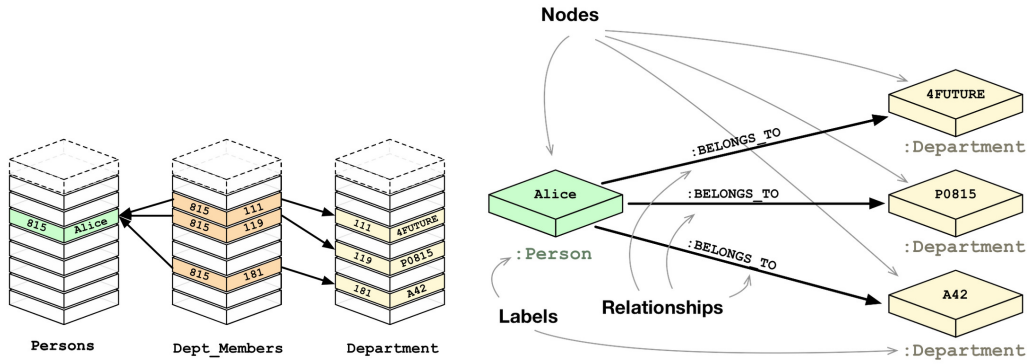

Figure 1.1: A simple relational database and the corresponding graph database. Images are courtesy of Neo Technologies [58].

base storage format for *graph databases* [31, 1]. Such databases provide the ability to serve less structured data and represent the problem in a form that better matches its domain. Graph database models encode data using vertices, edges, and properties and are used to respond to semantic queries. This structure reduces the need for indices or expensive joins, as every element can be connected directly to its adjacent elements.

Figure 1.1 illustrates a simple relational database and its graph equivalent. Note that the graph representation is more compact in this case, because foreign keys (e.g. student ID) and join tables (e.g. Dept_Members) can be omitted. The benefit of the graph database is made evident when queries made against this database are translated into algorithms on the graph representation. An example is the simple question: "How many of John's friends study English?" Answering this question naively using the relational database would require one join to generate the list of friends and another join to determine their courses of study. The graph representation translates joins into graph edges that are traversed with a direct memory access. In this case, a simple graph algorithm would be generated that performs two hops of a breadth-first traversal from the "John" vertex. Despite the plethora of join mitigation techniques in traditional databases such as query optimization and database denormalization, queries that are fundamentally join-heavy are often difficult to avoid [54, 2].

1.2 Challenges in Energy-Efficient Graph Processing: The Case for Hardware Acceleration

Energy-efficiency considerations in large-scale compute environments are of paramount importance due to the financial and environmental impacts of cooling and power costs [33]. There exists a need for frequently-performed operations, such as those involving graph analytics, to be performed efficiently [55].

Unfortunately, graphs are notoriously difficult to process efficiently. In addition to the previously-mentioned problem of large data set sizes, many important graph algorithms also suffer from:

- Poor locality of reference, both spatial and temporal.
- Data-dependent memory accesses, leading to pointer-chasing behaviour.
- Low computation-to-communication (via memory) ratio.

The practical effect of these issues in most mainstream computer architectures is to restrict many graph analytics algorithms to the memory-bound regime. Architectures with higher memory bandwidth and large numbers of outstanding memory requests therefore show superior performance in these applications [41].

One approach to addressing the energy consumption issue while maintaining (or improving) performance is to use dedicated hardware accelerators for common operations. Unfortunately, custom processors are prohibitively expensive to design for most use cases [56]. General purpose processors are typically used in these compute environments, but other architectures are capable of high-performance computation and can be applied to the domain of graph analytics.

1.2.1 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) are massively parallel processors that feature a large number of execution units and a specialized memory system designed to provide immense data bandwidth. Long memory latencies are tolerated using context

switching on a large number of threads. Intuitively, the large number of threads available on a GPU provide a good mapping for large graph analytics algorithms due to the abundant parallelism inherent in processing the many elements of the graph. In practice, however, details of the GPU architecture such as grouping of threads into warps [18] and the coalescing of memory accesses must be carefully considered if one is to achieve efficient performance using the GPU.

Several researchers have studied the use of GPUs in performing analytics on large graphs. Harish et al. [32] demonstrated raw benefits in performance on traversal algorithms using a level-synchronous approach. Merrill et al. [44] used a novel work-optimal traversal algorithm to greatly improve performance in both single- and multi-GPU configurations. Gharaibeh et al. [27] proposed a heterogeneous processing engine that used the CPU and GPU to demonstrate performance in both traversals and page rank.

From the preceding examples and others, it is evident that GPUs are a feasible candidate to be used as a processor for graph analytics algorithms. However, several key characteristics of their architecture prevent them from being optimal:

- The memory capacity available to modern GPUs is insufficient to hold very large graph instances. Current GPUs have a maximum of about 12 GB available to the compute units (early 2015). Coupled with an insufficient interconnect, this memory capacity limits the GPU problem size.
- Energy usage on high performance GPUs is significant, with high-performing GPUs have a TDP in excess of 230 W (early 2015) [18].

1.2.2 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) are chips designed to be reprogrammed by a designer after deployment. The user typically defines the chip's circuit hardware specification using a hardware description language (HDL). Contemporary FPGAs offer a large amount of reconfigurable compute resources, high-speed bi-directional interconnect busses, and on-chip RAMs [17, 37]. The flexibility of FPGAs allows the same chip to target a variety of applications.

From the standpoint of energy efficiency, FPGA-based accelerators are extremely advantageous when compared with general purpose processors and GPUs. They generally operate at lower clock frequencies, about an order of magnitude less than those found in general purpose processors. Logic such as out-of-order circuits, instruction fetch/decode logic, and large active cache arrays are usually not included in the accelerator design, saving power [28, 38].

In addition, many FPGA platforms are paired with high-performance memory subsystems. Large DRAM banks and high-bandwidth interfaces are often available off-chip. As in most computer architectures, superior memory bandwidth is available via on-chip memory. On FPGAs, these are represented as block RAMs (BRAMs). It is left to the designer to provide a parallel architecture to suitably make use of the chip resources.

Unfortunately, making efficient use of an FPGA's significant resources is often non-trivial, because the greatest obstacle to widespread FPGA usage is arguably programmability [15, 20]. In practice, the low level nature of the hardware description languages calls for significant hardware design and architecture expertise in order to develop high-performing implementations. Higher level descriptions and behavioral expressions, particularly those involving complex control flow, are often not well analyzed by compiler tools [43, 30]. In addition, debug and verification cycles for hardware tend to require significantly more time than corresponding software systems.

1.2.3 Dataflow Computation

Dataflow architectures are a special type of computer architecture in which there is no traditional program counter dictating the control flow of the program [51]. The idea was first proposed in the 1970s, along with the related notion of a *dataflow graph* (DFG) [22, 24, 23]. Unlike standard control flow based architectures in which instructions are executed sequentially according to a program counter, data flow tasks are activated immediately upon availability of the required inputs. The data flow model is well-suited for parallel architectures, because execution is inherently independent

```
// inputs i1, i2, i3, i4
// outputs z1, z2

m1 <= i1 + i2;
m2 <= i1 * i3;
z1 <= m1 + i4;
m3 <= m2 – i4;
z2 <= m3 + i3;
```

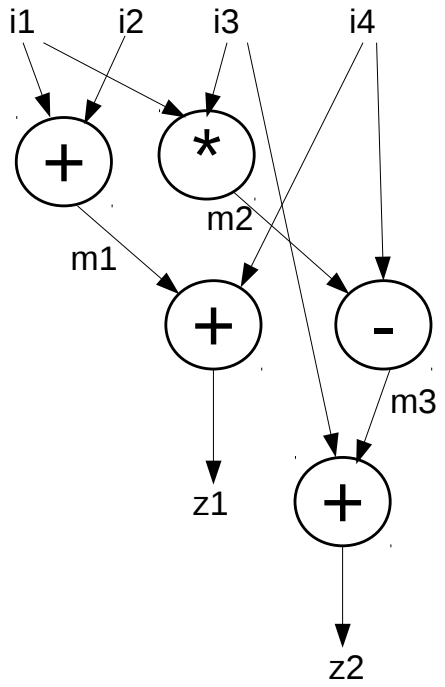

Figure 1.2: A fine-grained DFG for a simple arithmetic computation.

of the traditional sequential program counter. Multiple tasks will execute simultaneously by default, or in arbitrary order, due to the lack of dependency between those tasks [46].

The execution engines for each task are often referred to as *kernels*. Data flows between task kernels according to the DFG of the program, but data necessarily also flows from memory to the kernels. Ultimately, system performance is restricted by the existence of program data dependencies as well as the ability of the memory subsystem to supply data to the task kernels. The dataflow computational model is sometimes referred to as a "spatial processing" model. The architecture obviates the need for functionality such as instruction decode, branch prediction, or out-of-order logic. It has been successfully used in applications such as digital signal processing [7], network routing [11], and graphics processing [39].

Figure 1.2 illustrates a DFG for a simple computation. The kernels, in this case, are simple arithmetic operations. Each kernel consumes data from its inputs and

produces data for its outputs. In standard dataflow formulations, a kernel begins processing as soon as all required inputs are available and halts in the case that any required input does not have data. This necessitates the construction and fine-tuning of queuing structures for programs in which kernels may vary their processing rates in different phases of execution.

1.3 GraphOps: A Hardware Library for Graph-Specific Computation

Our research takes a different approach to enabling the design of high-performance and energy-efficient accelerators for graph analytics. This dissertation presents *GraphOps*, a hardware library for quickly and easily constructing energy-efficient accelerators for graph analytics algorithms. The GraphOps library is built on top of the dataflow paradigm and provides a hardware designer with a set of composable graph-specific building blocks capable of describing a wide array of graph analytics algorithms. The target user is a hardware designer who may struggle to design his own logic and would benefit from a library of pre-verified building blocks that are tailored to the domain of graph analytics. In GraphOps-based systems, the graph is pre-processed on the host and stored in a locality-optimized format that maximizes spatial locality when accessing properties of the graph. Stubborn hardware implementation details such as flow control, input buffering, rate throttling, and host/interrupt interaction are automatically handled and built into the design of the GraphOps, greatly reducing design time.

1.4 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 describes two alternative approaches we initially used to develop FPGA-based graph analytics accelerators. For each approach, we detail the design decisions and eventual pitfalls. We conclude by summarizing the research contributions and

takeaways.

Chapter 3 presents a novel graph representation optimized for increased spatial locality. This memory representation is used by several GraphOps components when accessing the properties of the graph elements.

Chapter 4 presents the design and implementation of GraphOps, a library of flexible, composable hardware modules used to construct high-performance streaming graph analytics accelerators. We enumerate the various components of the library, explain the design decisions therein, and evaluate the performance of accelerators constructed using the library.

Chapter 5 concludes the dissertation by discussing the research contributions in the context of the results from all previous chapters. We discuss directions for future work and finish with some final thoughts on the role of FPGAs and accelerators in data analytics applications.

Chapter 2

Initial Research Architectures

In this chapter, we describe the evolution of our research in targeting reconfigurable hardware for efficient processing of graph analytics algorithms. Before arriving at the final GraphOps version, the library underwent two previous design and implementation iterations. Each iteration shed light on limitations in our design decisions and suggested new components necessary to create a modular reconfigurable architecture for graph analytics.

This chapter is composed of three sections. The first section describes the target hardware platform used for all experiments. The second and third sections describe the two previous GraphOps versions. In each section, we present the design decisions and implementation details of the system. We then describe obstacles encountered in completing the research goals and detail the research contributions generated by the work.

2.1 Target Hardware System

Our target hardware system was a platform developed by Maxeler Technologies [57]. The FPGA was a Xilinx Virtex-6 (XC6VSX475T) chip with 475k logic cells and 1,064 36 Kb RAM blocks for a total of 4.67 MB of block memory. The FPGA was connected to 24 GB of DRAM via a single 384-bit memory channel with a maximum frequency of 400 MHz DDR. This corresponded to a peak line bandwidth of 38.4 GB/s.

The FPGA was connected via PCIe x8 to the host processor system. The host system had two 2.67 GHz Xeon 5650 multi-processors, each having six multi-threaded cores. This corresponded to a total count of 24 hardware threads. Each of the two processors had a peak line memory bandwidth of 32 GB/s and three 64-bit channels to memory.

The FPGA programming environment included a compiler framework called MaxCompiler [51]. For our purposes in these two initial versions, MaxCompiler provided a convenient way to integrate the necessary memory controller needed for our designs. We made little use of the primary MaxCompiler primitives, e.g. kernels, for these initial architectures.

We were attracted to the Maxeler platform primarily because of the ample memory bandwidth available to the FPGA. In addition, this platform offered an optimized memory controller and compiler that handled proper instantiation of the memory interface.

2.2 Version 1: Verilog-Embedded Hardware Generation

The first version of GraphOps was initially devised as a high-level synthesis (HLS) approach to hardware for graph analytics. Our goal was to create a hardware back end for a domain-specific language called Green Marl [36]. Green Marl is a graph-centric language in which the syntax allows an end user to easily express a graph analytics algorithm. Its compilation framework performs domain-specific optimizations on various intermediate representations. The standard Green Marl compiler supported back ends and code generators for multi-threaded CPUs (OpenMP in C++) and GPUs (CUDA in C++).

We instrumented the Green Marl compiler by adding an FPGA-centric optimization pass. This pass was composed of IR modifications and optimizations designed to target a spatial architecture in reconfigurable logic. Our plan was to use the final result as a driver for code generation in the Verilog HDL. In addition to modifying

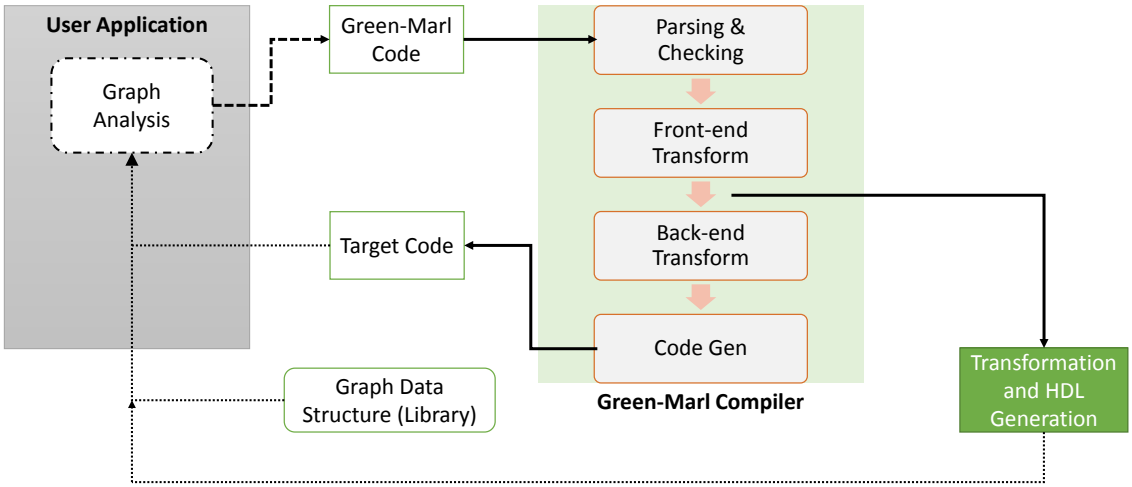

Figure 2.1: Compiler architecture for the Verilog back end.

the IR, the compiler performed static analysis on the hardware design in order to instantiate boilerplate code for support hardware such as memory system interfaces. Figure 2.1 shows the architecture of the overall system.

2.2.1 Implementation

The engineering workflow undertaken to implement this Verilog-based compiler back end was the following:

Step 1: Compiler Framework Skeleton The Green Marl compiler organizes the back end targets into classes (e.g. C++/OpenMP and CUDA/GPU). Each class is composed of target-specific elements such as a code generator, syntax keywords, and optimization passes. The first step was to design and instantiate a preliminary back end data structure to be used for targeting hardware. This allowed us to become familiar with the architecture and implementation of the compiler's internals.

Step 2: Initial Hardware Design After the core of the compiler back end was in place, the next step was to implement a full sample application in Verilog. The first purpose of this step was to settle on hardware implementation details such as the design of the memory interface. The second purpose was to ensure functionality of a custom Verilog design on our target hardware platform. This was because we needed

to ascertain that complex arbitrary Verilog logic could integrate properly with the underlying compilation MaxCompiler framework.

Step 3: Implementation of Code Generator The final step was to implement the optimization passes and the code generator. This step involved processing the IR, instantiating the required hardware modules and connecting logic.

2.2.2 Final Status of Version 1

We completed the first step of the workflow by implementing a basic placeholder Verilog back end for the Green Marl compiler. Our back end was composed of a syntax keyword library and basic generation logic for Verilog constructs such as *always* blocks, *assign* statements, and *generate* blocks.

For the second step of the Verilog workflow, we chose to implement two different kernels as proof-of-concepts of using Verilog on our target hardware platform: a simple memory writing kernel and a breadth first traversal (BFS) of a network expressed using adjacency lists. Figure 2.2 shows a diagram of the architecture for our Verilog-based graph analytics accelerator. One goal of this architecture was to abstract away details of the system memory commands, presenting a simplified interface to the graph analytics application. This greatly simplified the code generation logic. The block labeled *Graph Computation Kernel* contains the output of the compiler. Building these two kernels provided great insight into how best to structure the code generator and the potential obstacles therein.

We verified correctness of the simple memory assignment kernel. However, verification of the BFS kernel proved to be intractable. In order to be compiled by the underlying MaxCompiler software framework, our full Verilog BFS design was embedded inside an additional software wrapper. During standard debug and verification of our custom design, it became clear that the framework's wrapper did not support convenient use of standard Verilog simulation environments such as ModelSim, especially when the DRAM memory controller was involved.

After deliberation and thought, we decided to fundamentally re-design our graph analytics hardware generation approach for the following reasons:

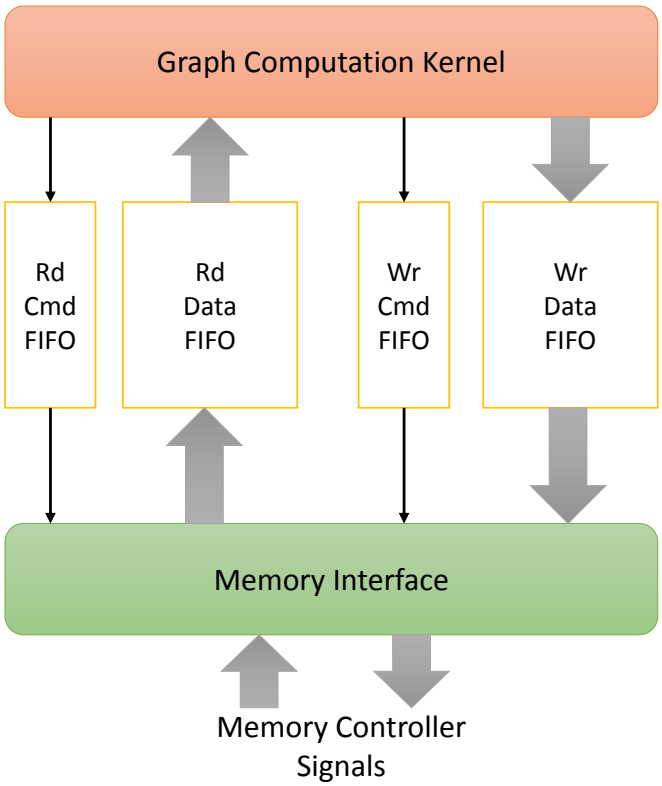

Figure 2.2: Architecture of the Verilog-based accelerator designs. The "Graph Computation Kernel" is a placeholder for the specific graph analytics computation code generated by the compiler.

1. The lack of visibility in the verification process would have proven to be prohibitively costly, from a time standpoint, as we proceeded with additional Verilog designs.

2. Implementing a complex, fine-grained breadth-first traversal in hardware convinced us that direct Verilog generation of arbitrary graph analytics logic as expressed in the Green Marl language would require significant additional information than was already available in the current compiler IR.

3. Targeting pure Verilog revealed to us that including and harnessing a memory controller in the design was not well-supported by our underlying framework wrapper layer.

2.3 Version 2: Hardware Generation using Nestable State Machines

Given the obstacles we encountered in our first version, we re-designed our hardware generation framework to address these issues.

- We addressed the visibility issue by choosing a design which did not rely on the custom Verilog wrapper offered by our underlying framework. Instead, we targeted the HDL layer that was directly supported by our underlying framework. Targeting this layer changed our verification flow, but allowed us far more visibility into the design.

- To address the IR information issue, we chose a design that made greater use of boilerplate logic. For instance, when traversing a vertex's neighbors (a common paradigm in graph computation), the majority of the logic necessary is constant and can be instantiated from a template. The functionality that needs to be encoded dynamically is expressing what kind of computation is performed with the neighbor's data.

- Targeting the framework's HDL layer also mitigated the memory interface issue. Using MaxCompiler's primitives allowed us to make use of the framework's automated generation of memory interface components.

With these modifications in place, the second version of GraphOps was more easily generated by a compiler and allowed for manageable visibility during verification. This version further incorporated the research lessons from the first version by pulling in external support for key system utilities such as the memory subsystem.

2.3.1 Implementation

The final product of this version was a system in which the application syntax tree drove generation of hardware by nesting state machines of different types. Each IR node mapped directly to a hardware *template*. When a state machine finished execution, it automatically reverted "control" back to the calling state machine. The HDL we targeted was the MaxJ state machine language, a component of the MaxCompiler framework [57].

The motivation for this design was driven by the low-level IR nodes available in the compiler's syntax tree. We observed that some of the nodes were not easily decomposed. For example, a complex IR node such as the global breadth first traversal was not naturally decomposed to smaller pieces of computation that could be integrated with other computations in the application.

We therefore designed a system in which optimized boilerplate hardware for general "outer loop" functionality such as a neighbor traversal could be instantiated via a template. The templates would provide functional hooks for a *user function* to be supplied which would serve as the "inner loop" to the template.

Figure 2.3 portrays the general design of a template. The template of optimized boilerplate hardware is represented by the blue outer shell labeled "General Template". The template description was static in structure, but parameterizable via template-specific parameter variables as specified in the diagram.

The hooks for the user functions are represented by the "Inner User Function" block. The code for this inner block was generated by descending further down the

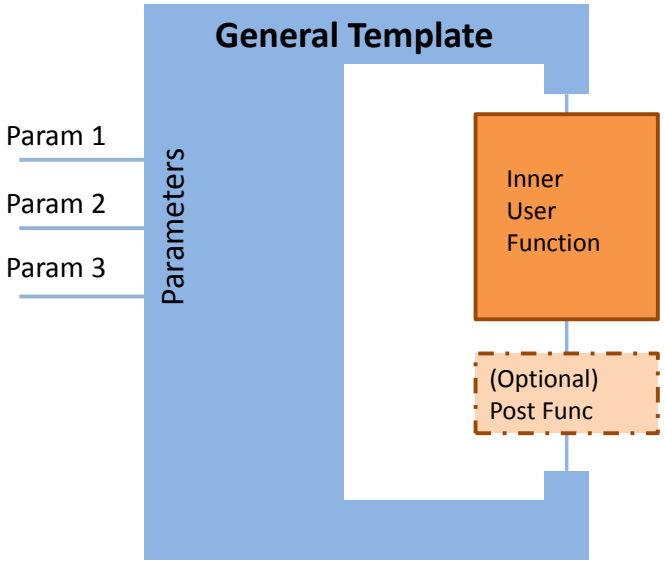

Figure 2.3: General design of a state machine hardware template.

syntax tree and processing child nodes. Along with the user function, some templates had placeholders for an optional "Post Function". This logic specified logic that should execute after the main user function had completed.

Example State Machine Design

The goal behind the system described was to allow us to generate hardware for codes such as the following application in Listing 2.1:

Listing 2.1: Sample Green Marl program used for state machine generation.

```
1 InBFS (v: G.Nodes From s) {
2     Foreach (t: v.Nbrs) {
3         t.sigma = value0;
4     }
5     v.sigma = value1;
6 }
```

In Listing 2.1, every vertex that is touched by the breadth first traversal undergoes a traversal of its neighbors. At each neighbor of the vertex, an update is performed on a vertex property. Expressing this computation in hardware requires the generator

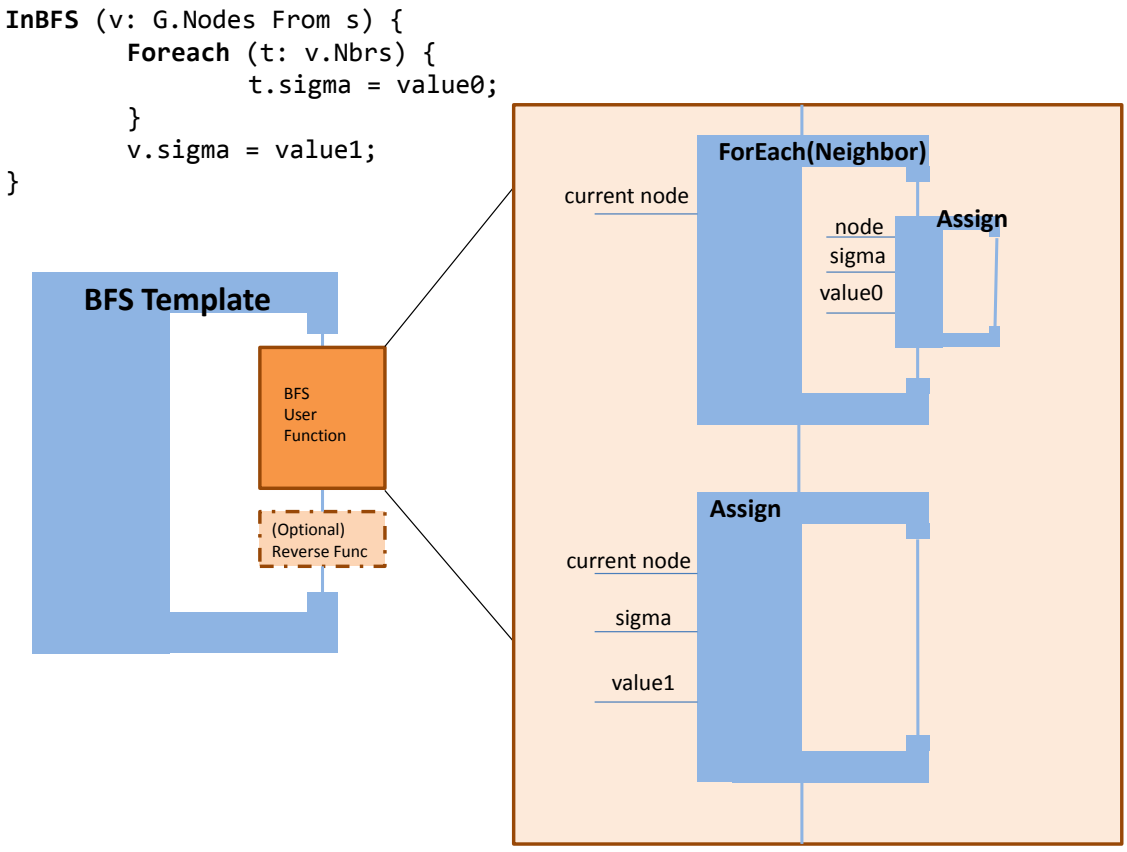

Figure 2.4: State machine diagram for a sample application.

to automatically reason about the nesting behaviour: assignment takes place within a "Foreach" structure which takes place within a BFS traversal.

Figure 2.4 portrays a diagram of this application as expressed in our state machine paradigm. The user function of the BFS Template, composed of everything within the outer loop, is studied in detail inside the larger box. Within the BFS user function, two templates are instantiated serially, corresponding to the syntax tree IR for the application. The first template, a ForEach template, is processed further and its user function populated with an Assign template.

States within a state machine were used to organize control flow and allow for more natural integration of all templates and user functions. Every template and user function implemented a special *tick()* function. This function served as a single

point of control, allowing the parent instance to move execution forward in the child instance. Control returned to the parent when the child completed execution.

Example Pseudocode for User Function

Listing 2.2 shows pseudocode for the BFS user function studied in Figure 2.4. The structure and logic of the user function were kept simple, since user functions were to be generated by the compiler. The primary areas of logic that required generation were the constructor function and the *tick()* function.

Within the constructor function, the compiler enumerated the number of child templates and generated construction statements for those children. For those templates that required user functions of their own, the compiler also generated placeholder constructor statements for those objects. These user functions were populated by the compiler in their own source code files.

Within the *tick()* function, the *IDLE* and *FINISH* states were automatically instantiated as boilerplate. In addition, one state was instantiated for each child template that was enumerated by the compiler. The pseudocode in the listing omits details in the setup before calling *tick()* such as setting the objects parameters and initialization variables. In this example, two states were instantiated, one each for the FE_Nbr (*ST0*) and the Assign templates (*ST1*). For the active state, the user function calls the template's *tick()* function, advancing its execution once per cycle. All other states are not processed. Each template also has a *isDone()* function, which returns a boolean true when execution has finished. For this purpose,

The core logic of each template was embedded in the template hardware description, leaving a simpler user function for the purposes of hardware generation.

Listing 2.2: Pseudocode for the BFS user function.

```
1  UserFunc()
2  {
3    // Generated user functions
4    UF_Reduce1 = new Reduce_UF()
5
6    // Generated templates
7    FE_Nbr1 = new FE_Nbr_Template(UF_Reduce1)
8    Assign1 = new Assign_Template()
9  }
10
11 bool tick()
12 {
13   Switch(state):
14
15     Case(IDLE):
16       state.next <== ST0
17
18     Case(ST0): // FE_Nbr template state
19       FE_Nbr1.tick()
20       IF (FE_Nbr1.isDone())
21         state.next <== ST1
22
23     Case(ST1): // Assignment state
24       // set parameters for the assign template
25       Assign1.tick()
26       IF (Assign1.isDone())
27         state.next <== FINISH
28
29     Case(FINISH):
30       state.next <== IDLE
31
32   return (state == FINISH)
33 }
```

Example Pseudocode for Hardware Template

Listing 2.3 shows pseudocode for the ForEach-Nbr hardware template. The structure and logic of the hardware templates could be arbitrarily complex, as they were not generated by the compiler.

The constructor registered the user function that served as the inner loop to this hardware template. For hardware templates, the notion of "inner loop" was different depending on the nature of the computation. In this ForEach-Nbr example, the user function was executed on each neighbor of the vertex. In a different example, such as a BFS, the user function would have been executed on every vertex encountered during the traversal. The constructor also initialized all the state registers and memories used in this template.

The states of the hardware templates differed from those of user functions, because the hardware templates used the state machine to control logic and execute a data-path, in addition to just advancing their child nodes. In ForEach-Nbr, for example, the first three (non-*IDLE*) states handled the requesting and storing of the neighbor property values used in the computation described in the user function. The fourth state (*ST3*) applied the user function that was supplied in the constructor to each stored neighbor property value.

Note the presence of the *isDone()* function in the hardware template. As shown in Listing 2.3, its implementation was quite trivially a statement of whether the template's state had reached the *FINISH* state.

Listing 2.3: Pseudocode for the ForEach-Nbr template.

```
1  BFS_UserFunc(UserFunc uf)
2  {
3    this.uf = uf
4    // initialize hardware state, registers, memory
5  }
6
7  void tick()
8  {
9    Switch(state):
10
11     Case(IDLE):
12       state.next <== ST0
13     Case(ST0):
14       // initiate read memory command for adjacency list
15       state.next <== ST1
16     Case(ST1):
17       // parse inputs, initiate read command for nbr lists
18       IF (finished parsing inputs)
19         state.next <== ST2
20     Case(ST2):
21       // store nbrs in memory
22       IF (finished parsing nbrs)
23         state.next <== ST3
24     Case(ST3):
25       // call user function's tick() on all nbrs
26       IF (finished processing nbrs)
27         state.next <== FINISH
28     Case(FINISH):
29       state.next <== IDLE
30 }
31
32 bool isDone() {
33   return (state == FINISH)
34 }
```

A particular benefit of this architecture was that the HDL for such an implementation could be naturally generated via a traversal of the IR syntax tree.

2.3.2 Final Status of Version 2

Using the state machine system, we successfully generated simple hardware graph analytics applications. The basic BFS-based application presented in Listing 2.1 was one of the development kernels that we ran in hardware.

The primary bottleneck to continuing development of the state machine system was application complexity when executing the place-and-route (PAR) procedures. During the development phase of the system, we began to build up more complex applications to ensure that: (a) the MaxCompiler framework and Xilinx tools were capable of handling our deeply nested hardware designs, and (b) applications of interest to us would fit within the hardware resources of the Xilinx FPGA.

Betweenness centrality (BC) [10] was one algorithm that was of particular interest to our group. The BC algorithm computes a metric which indicates a vertex's centrality in a network. Listing 2.4 displays a version of the BC calculation expressed in Green-Marl.

Listing 2.4: Betweenness centrality algorithm expressed in Green-Marl.

```
1  Procedure comp_BC(G: Graph, BC: N_P<Float>, Seeds: Node_Sequence)
2  {
3    G.BC = 0; // Initialize
4
5    For (s: Seeds.Items) {
6
7      // temporary values per Node
8      Node_Property<Float> sigma;
9      Node_Property<Float> delta;
10     G.sigma = 0;
11     s.sigma = 1;
12
13     // BFS order iteration from s
14     InBFS(v: G.Nodes From s) {
15       // Summing over BFS parents
16       v.sigma = Sum(w:v.UpNbrs) { w.sigma };
17     }
18     InReverse { // Reverse-BFS order iteration to s
19       v.delta =  // Summing over BFS children
20         Sum (w:v.DownNbrs) {
21           v.sigma / w.sigma * (1+ w.delta) };
22
23       v.BC += v.delta @ s; // accumulate BC
24     }
25   }
26 }
```

Generating hardware for betweenness centrality proved to be prohibitively expensive for the underlying PAR tools to successfully generate a design.

BC used a larger number of hardware templates than other previously studied algorithms. The reader should note the heavily nested nature of the computation. Some of the templates required were resource-heavy, with BFS template being the most expensive. BFS, ForEach-Nbr, and any other templates that read and store graph elements all made heavy use of on-chip block memories, for example.

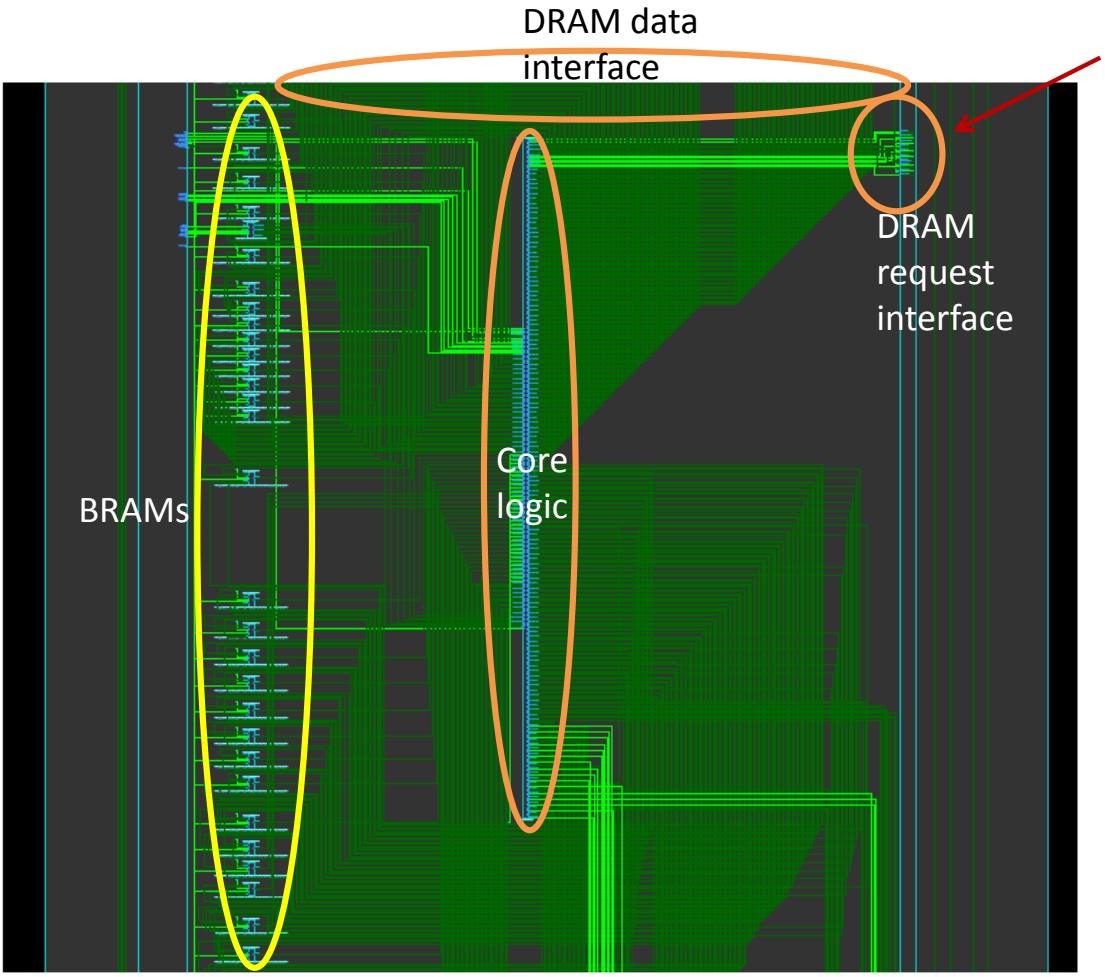

Figure 2.5: Logic and wiring schematic of the betweenness centrality application shown in Listing 2.4 on a Xilinx Virtex-6 FPGA.

The second, and more serious, source of contention was access to the chip's memory interface. Generating different hardware templates in a spatially disparate manner had the unintended consequence of creating wiring and muxing bottlenecks to serve all templates which used memory. Figure 2.5 displays a schematic of the betweenness centrality application.

The target frequency was 100 MHz. The worst paths failed by about 70%, meaning a 17ns delay on a 10ns clock period. The design used about 20% of the chip's logic and memory resources, indicating that wiring was the actual bottleneck. Further analysis of the critical paths showed that they were about 80% wire and 20% logic.

Figure 2.5 shows a significant number of wires being routed sub-optimally from the DRAM data pins to the core logic and BRAMs.

After significant deliberation, we realized that this wiring bottleneck was an unfortunate byproduct of the fundamental design of our hardware generation approach. The lack of hardware re-use allowed for an elegant and simple template-based approach to automation but at a prohibitively expensive cost in resource usage and therefore PAR complexity. With these realizations in mind, we settled on a final re-design of our graph analytics hardware acceleration system, making use of lessons learned in the two initial versions discussed in this chapter.

Chapter 3

Accelerator-Friendly Graph Representations

As a precursor to presenting the GraphOps library in the next chapter, this chapter briefly summarizes our work in storage representations of graph data structures for accelerator architectures. We ask that the reader tolerate forward references to GraphOps design details which are fully described in the next chapter. An understanding of our graph representation is necessary in order to understand the design of the library components.

3.1 Introduction

It has long been known that computational performance of many graph analytics codes is bound by memory bandwidth [13], as they feature large communication to computation ratios. The memory bandwidth constraint is compounded by a dearth of spatial locality, given the inherent sparseness of graph data structures. Unfortunately, the memory controllers of most mainstream computer systems and FPGA accelerators are optimized for spatial locality.

In fact, there has been a trend in memory systems towards even coarser granularities, e.g. longer burst fetches [45]. This is particularly true in memory systems

of accelerator architectures. This trend allows for function-specific circuits to maximize data bandwidth and processing throughput with minimal increases in clock frequency and control circuitry. As of 2015, for example, current high end GPU memory interfaces are 384 bits wide on a quad-pumped 1 GHz bus [18]. Similarly, some FPGA-based memory controllers optimize for data throughput by limiting physical DDR operations to burst fetches and exposing very wide memory blocks to the end user. This method is used by the Maxeler platform we used for our experiments. Our platform, introduced in Section 2.1, exposed a minimum memory block width of 1.5 Kb to the programming environment.

Because of the large blocks exposed by the memory system, the GraphOps library uses data parallelism on the graph elements contained in each block in order to maximize throughput. It was necessary to devise a novel data structure that would provide the spatial locality necessary to achieve this. This chapter proposes such a data structure, a *locality-optimized graph representation*, that trades off compactness for spatial locality via data duplication.

3.2 Traditional Graph Representations

There is abundant prior research on data structures for representing graphs in computer memory [61, 13, 12]. Graph formats are designed and optimized based on factors such as the size of the graph, its connectivity, desired compactness, the nature of the computations done on the graph, and the mutability of the data in the graph. A graph's topology can be fully specified via an adjacency matrix, with the matrix elements corresponding to graph properties. Therefore much of the literature on graph representation is expressed using the language of sparse matrix representation [5]. Common sparse formats include the diagonal format, coordinates list, ELLPACK/ITPACK, and compressed sparse row (CSR),

The diagonal format [53] is used for diagonal matrices, in which nonzero elements are found along diagonals in the matrix. This format is not considered general purpose, as it is useful only for graphs whose matrix exhibit the diagonal pattern.

The ELLPACK/ITPACK format [29] reduces the sparse MxN adjacency matrix

to a dense MxK matrix in which K is the maximum number of nonzero elements in any given row of the matrix. All rows are zero-padded to achieve the length of K. The blocked, regular structure of the format makes it well-suited to vector and SIMD architectures. However, the zero-padding causes immense overhead for graphs whose distribution of neighbor count (nonzero elements in a row) follows a *power law* distribution. Unfortunately, this phenomenon has been shown to be common in many interesting real world graphs, social networks being a prominent example [3].

The compressed sparse row format is perhaps the most popular general sparse matrix representation. It is extremely compact, because it abstains from zero-padding rows, as done in the ELLPACK format. As a result, it is an ideal representation for compactly storing irregular power law graphs. For the duration of this thesis, we restrict our study to static graphs and formats, as opposed to those which effectively support rapid mutation.

Given a graph with N nodes and M edges, the CSR representation is composed of a vertex array (or node array) of length N and an edge array of length M. Figure 3.1 portrays this data structure for a simple graph, which is also shown in the figure. The vertex array is effectively an *index* array. This array is itself indexed by the vertex ID (an integer less than N) and stores indices into the edge array. Each of these indices is the start index (in the edge array) of the list of edges outgoing from the vertex. The edge array stores the destination vertices of each edge as a concatenated grouping of all the "lists of neighbors". In fact, the compressed sparse row representation is sometimes referred to as a format based on "adjacency lists".

3.3 Design Philosophy

The compressed sparse row storage representation is the most optimally compact for static sparse graphs. Unfortunately, this compactness introduces inefficiencies in memory behaviour for some codes. For instance, the variable number of neighbors in the graph causes severe thread divergence and workload imbalance in codes involving neighbor processing or frontier exploration. This is exacerbated by graphs defined by the power law distribution due to the existence of very few nodes which have a large

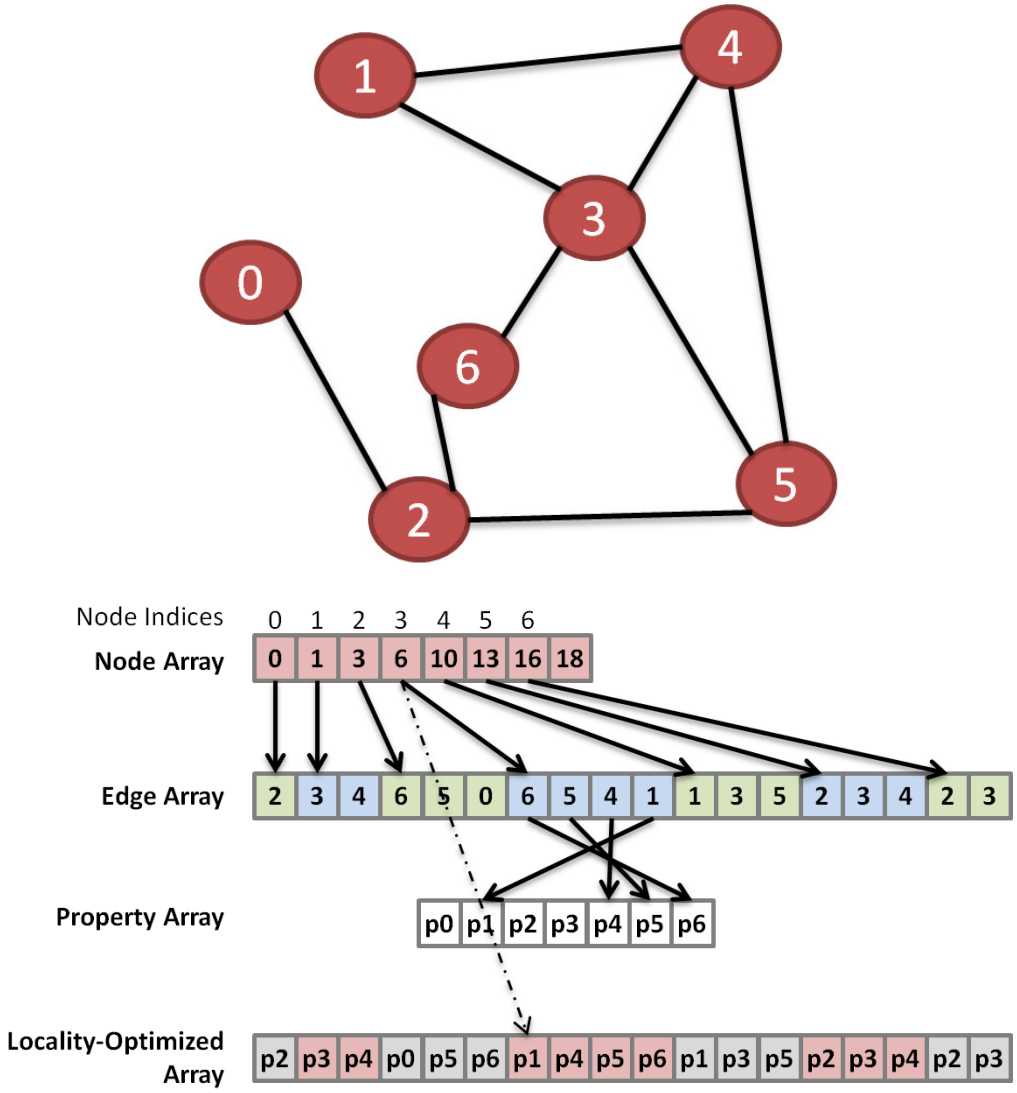

Figure 3.1: A simple graph and its associated data structures.

percentage of all edges in the graph. Programming techniques, such as warp-centric row processing, can mitigate these effects [34, 4].

In addition to the divergence issue, CSR also suffers from the more fundamental problem of irregular memory accesses when traversing neighbor edges. Consider a simple example in Figure 3.1. Beginning at the node array, index 3, consider an algorithm that needs to access the properties of the neighbors of vertex 3. Reading the vertex array returns an index into the edge array. Next, reading the edge array returns a set of destination vertices, the neighbors of vertex 3. We encounter the irregular memory behaviour when accessing associated properties for that neighbor set. At that point, the system must perform random access into a property array. This scattering effect is visualized by the crossing arrows pointing into the property array in the figure. Unfortunately, accessing neighbor properties is a common paradigm in graph algorithms–many important computations are concerned with the data elements of a vertex's neighbor.

A naively implemented neighbor exploration would unnecessarily serialize the memory accesses and make poor use of data blocks fetched from memory. This effect would be particularly felt in: (a) SIMD or vector architectures, as most of the lanes would be inactive and (b) systems with memory controllers that enforce wide burst fetches from memory, such as the Maxeler platform used in our experiments.

To address this issue, we propose an augmentation to the CSR representation called the locality-optimized array (LO-array), also shown in the figure. This new data structure is similar to the edge array, but instead of storing lists of neighbors, it stores properties associated with those neighbors. Reading neighbor properties now is achieved without random access, providing the spatial locality necessary for data parallel architectures. Updates are still performed to the original property array.

For algorithms involving calculations with multiple graph properties (and therefore multiple property arrays), the user instantiates multiple LO-arrays, one for each property array.

Although an LO-array has the same number of elements as the edge array, the storage footprint of the LO-array may differ from that of the edge array depending on the data type of the properties in that array.

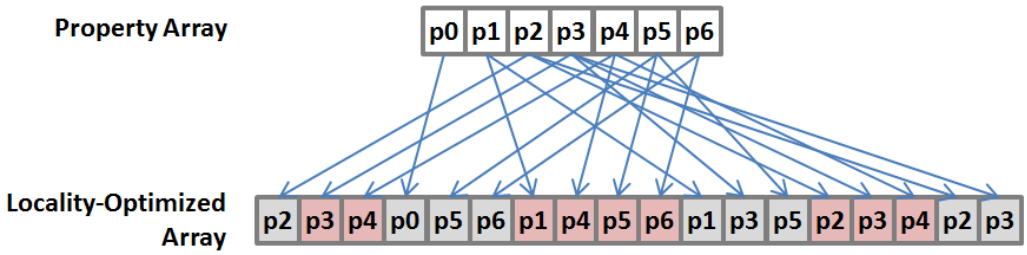

Figure 3.2: The locality-optimized array is generated and updated offline on the host.

One potential question is runaway storage overhead as the number of LO-arrays grows. LO-arrays are O(M) and we propose to instantiate one LO-array per property. We respond to this question with the observation that very few of the algorithms we encountered used a large number of graph properties simultaneously, limiting the storage impact of the additional data structures. In addition, modern FPGA acceleration platforms support significant amounts of DRAM–some systems feature up to 48 GB per FPGA, as of 2015. We view the significantly improved spatial locality, at the expense of storage compactness, to be a useful architectural space-time tradeoff.

The reader may notice that the LO-array data structure effectively achieves locality by replicating graph properties of neighbor vertices and placing them within modified adjacency lists. The position of the adjacency lists corresponds to the position of the edges which are incident to the neighbor property in the LO-array.

The LO-array needs to be prepared and updated offline as part of the user's application code. Figure 3.2 portrays the scattering operation that is performed in order to prepare and maintain the array. The maintenance operation is performed on the host machine once per computation iteration. Note that consistency issues between the LO-arrays and the original property arrays are non-existent because all writes are performed to the property arrays, identical to the standard CSR implementation. In addition, each property array has only a single writer, by the design of the GraphOps blocks we will describe in the next chapter. The LO-arrays behave as read-only data structures during the computation (non-maintenance) phase of the iteration.

Listing 3.1: Code snippet for updating the LO-array using the property array. This code is also used to create the array. Both loops can be parallelized in a multi-threaded environment.

```
1  // scatter values from prop array to rep array
2  void rep_scatter(node_t *n_ptr, edge_t *e_ptr,
3                   prop_f48_t* prop_ptr, prop_f48_t* rep_ptr) {
4    for (int i = 0; i < NUM_NODES; i++) {
5      for (uint64_t j = getval48(&n_ptr[i]); j < getval48(&n_ptr[i+1])
6        uint64_t dn = getval48(&e_ptr[j]);
7        float48_t* p_ptr = (float48_t*)(&prop_ptr[dn]);
8        float prop_val = getval_f48(p_ptr);
9        float48_t* r_ptr = (float48_t*)(&rep_ptr[j]);
10       assign_f48(r_ptr, prop_val);
11     }
12   }
13 }
```

Listing 3.1 shows a code snippet for the scatter routine denoted in Figure 3.2. The outer and inner loops traverse the node and edge arrays, respectively, in order to generate position indices for the LO-array. Accessors and mutators for the graph data structures are encapsulated in macros: `getval48` and `assign_f48`. Macros allow for abstraction over details concerning use of the arrays, e.g. the data width of array elements.

3.4 Analysis

The LO-array is optimized entirely for spatial locality in reads. In this section, we briefly discuss the performance benefits enabled by this graph representation and the storage overhead induced by its design.

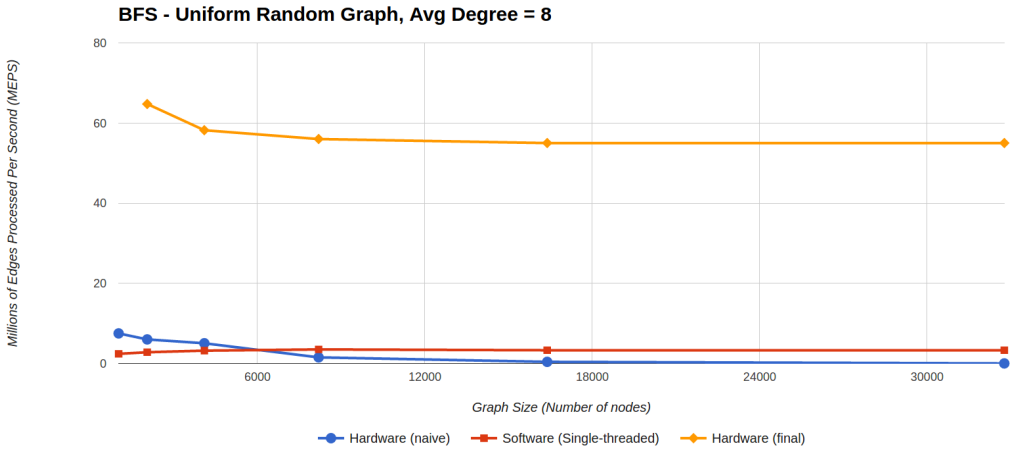

Figure 3.3: Results of the breadth-first search algorithm, compared with a standard single-threaded software implementation.

3.4.1 Performance

We have established that graph analytics applications incur large numbers of random memory requests. We also know that many memory architectures, particularly those of accelerators, are optimized for spatial locality. Given this landscape, one would expect poor performance for naively-implemented graph analytics applications when executed on these memory architectures. To study the extent of this effect, we implemented the BFS algorithm on our hardware platform (see Section 2.1) using standard CSR arrays (hence "naive"). In our simple hardware study, we used a small on-chip cache, as very large graphs would not benefit from caching due to lack of locality in the application. Figure 3.3 shows the results of our study.

Results show that the naive hardware performance approaches zero and is quickly bypassed by that of a single software thread, after the data set size outgrows the cache. It is evident that a brute force memory access approach is untenable for applications that lack locality. The final hardware implementation differs from the naive one in its use of the LO-array storage format and a frontier bitmap, a common BFS optimization.

This small study confirmed the necessity of a novel storage approach centered around spatial locality.

Algorithm	Number of Properties
pagerank	1
bfs	1
conduct	2
spmv	1
sssp	2
vcover	2

Table 3.1: Number of properties (and hence LO-arrays) used by each algorithm.

3.4.2 Storage Overhead

In this section, we analyze the storage overhead of LO-arrays by surveying the common applications studied in this dissertation (described in the next chapter, Section 4.5.2) followed by a discussion of asymptotic behavior as the number of graph properties grows.

Table 3.1 summarizes the number of node properties used by each of the algorithms studied. Although this list of algorithms is certainly not exhaustive, we believe it is diverse and representative of an interesting subset of the graph analytics space. None of the algorithms studied use a large number of properties in their calculations, so the storage overhead was not prohibitive. For the worst case of two additional LO-arrays, the overhead was on the order of hundreds of MBs.

The standard CSR storage format maintains the following data structures:

1. One O(N) array for the node indices
2. One O(M) array for the edges
3. N_p O(N) arrays for each of the properties

Our LO-array format maintains all of the preceding data structures but also includes:

4. N_p O(M) arrays for each of the LO-arrays (properties)

The storage overhead in terms of percentage can therefore be expressed as:

$$\frac{N_p * M}{N + M + N_p * N}$$

As N_p goes to infinity, the storage overhead approaches a factor of:

$$\frac{M}{N}$$

This is simply the average degree of the graph. The reader should note that "real-world" power-law graphs tend to be have lower average degree. In these graphs, a small percentage of nodes have large degree. In general, most real-world graph computations are concerned with extracting some insight from the structure of the network itself. The per-node information necessary for this insight tends to be well encapsulated in few per-node variables used in the computation. We have found large numbers of simultaneously used graph properties to be rare throughout our research.

3.5 Related Work

There has been abundant work in the general area of storage systems that employ data replication for various uses. Successful use cases include data replication for throughput [16] and redundancy and durability [60]. LO-arrays differ fundamentally from these approaches, because data replication is performed in a fine-grained manner as opposed to across disk drives or data centers.

X-Stream [52] is another graph processing system that uses a form of data replication in its design. We fully compare our work to X-Stream later in Section 4.5.4.

Chapter 4

GraphOps Architecture

This chapter formally presents GraphOps and describes its design details. GraphOps is a library of parameterizable dataflow hardware components that serve as building blocks for creating graph analytics accelerators. Each block serves a different function and can be instantiated to operate on input data streams and produce output data streams. A useful abstraction with which to understand GraphOps is to think of them as functional operators which execute on data streams between other blocks and to/from system memory.

The chapter begins with a holistic view of our design philosophy. After this overview, we will better motivate the idea of a dataflow hardware library by walking the reader through the design process for a single case study application. In the process, the reader will understand how an accelerator goes from being an idea to hardware using the GraphOps library. This will be followed by categorization, enumeration, and description for all the blocks in the library, based on their use in a dataflow program. We will then return to the initial motivating case study and use the GraphOps blocks from that application to dissect in detail the internal hardware components for a representative subset of the GraphOps blocks. Finally, we will present an analysis of the GraphOps library by evaluating applications accelerated using GraphOps and comparing the library to related work in the high-performance graph analytics space. Source code for selected components is provided in Appendix A.

We have open-sourced the entire GraphOps dataflow library under the MIT License as a Github repository [48]. The repository contains additional documentation that describes each of the blocks, parameterization suggestions, and composition instructions.

4.1 Design Philosophy

The GraphOps library design was heavily guided by lessons learned from our previous implementations of graph-specific hardware accelerators, as described in Chapter 2. With these experiences as guidelines, we were able to clearly define the design goals for a scalable and verifiable system on our target hardware:

- The GraphOps library needed to optimally exploit hardware re-use. Our experience with the state machine based accelerator showed us that, given the communication-heavy nature of graph algorithms, complex designs would incur significant place-and-route wiring difficulties, even if logic and memory resources were plentiful.

- The library needed to be embedded in a framework that would allow for feasible verification opportunities. We had learned that our target hardware and compilation framework (Maxeler) lacked infrastructure for sufficient Verilog verification and debug.

- The GraphOps library required low-level support for handling memory requests. Such functionality needed to be hidden from the user. In doing so, the library kept the focus on the algorithms and computations themselves and simplified the programming model.

- The GraphOps library needed to target a higher level of abstraction for its components. Similar to memory operations, previous experience taught us that construction of non-trivial hardware accelerators using lower level hardware abstractions still required significant hardware expertise from the user. Higher

order abstractions would limit flexibility in the functionality of the components– we decided to mitigate this using extensive block-level parameterization.

User Design Tradeoff: Higher Level Abstraction

With these goals in mind, we designed the accelerator programming model of GraphOps as one in which high-level graph-specific functional blocks are exposed to the end user. This high-level approach was a conscious trade-off, effectively changing the hardware generation operation from being part of a software compiler to instead being a standalone hardware design environment for graph analytics. The high-level nature of the blocks makes it easier for the user to reason about which blocks are needed to achieve desired functionality. In addition, using the functional dataflow paradigm ensures maximal re-use of the hardware. For graph analytics acceleration, the dataflow description instantiates a set amount of hardware and data is streamed rapidly through the blocks and to/from memory.

Heuristic Selection of Blocks

Once the design goal of a high-level functional hardware library was defined, we began the selection of blocks which would compose the GraphOps library. We chose the included blocks heuristically by reviewing a diverse set of graph algorithms and documenting frequently occurring paradigms in the computation. These paradigms were chosen for encapsulation as GraphOps blocks. Each blocks includes the core logic, control, memories, and wiring for the high-level graph operation. A catalog of the GraphOps library follows later in Section 4.3.

Parameterization

GraphOps blocks are designed to allow parameterization and flexibility. Parameters are set dynamically by the host software application during instantiation and calling of the accelerator hardware. Examples of possible parameters for each block include: size of BRAM memory arrays used for buffering, computation primitives (e.g. summation, in a reduction), and addresses of the arrays being read/updated. To further clarify how one would use the library to design hardware for a particular application, we will present a case study in Section 4.2.

User Interface

Hardware accelerators in the GraphOps model present themselves as co-processors to

a host application. The user's larger application may vary in general complexity, but inevitably features a well-defined computation-heavy algorithm that needs to be run on a graph data structure. This algorithm is implemented and embedded in hardware using the GraphOps tools described. The user issues calls to send graph data to the accelerator memory space. Finally, the hardware accelerator is then initiated via another API call from the user application. The program structure is similar to that of a GPU-accelerated application. Section 4.2 will present a detailed case study that includes application and hardware code.

Embedding Framework

The GraphOps library is implemented using the MaxJ dataflow programming language [51]. MaxJ is a dataflow hardware programming language that is embedded in Java. GraphOps is targeting end users that have basic familiarity with MaxJ and MaxCompiler. Advanced users with interest would be easily able to extend the GraphOps library with their own functional blocks.

Designing the GraphOps library on top of the MaxCompiler and MaxJ frameworks provided some key benefits. MaxJ code was extremely well supported from a verification standpoint. For example, a robust simulation environment was available.

In addition, using Maxeler tools at this granularity provided ample support for handling of memory commands. Memory request and data streams could be concisely declared and passed down to the compiler which would generate the buffers and arbiters/muxes necessary for the number of memory interfaces requested by the GraphOps blocks.

4.2 Case Study: Pagerank

4.2.1 Coarse-Grained Dataflow

Section 1.2.3 introduced the concept of dataflow computer architectures and their use in parallel computation. The example in that section performed simple arithmetic operations on primitive data types. In accordance with our design goal of higher level abstraction, we aimed to design the GraphOps blocks (kernels) with more built-in

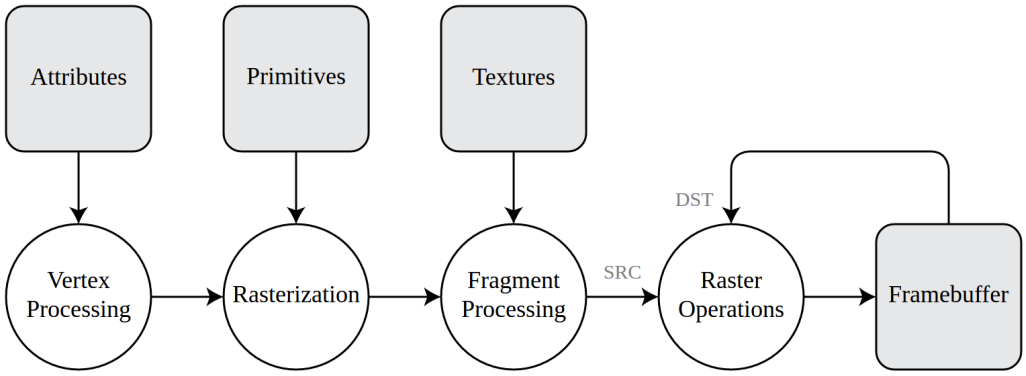

Figure 4.1: High-level dataflow operation in the graphics pipeline. GraphOps blocks use a similar model to express graph analytics computation.

functionality than simple arithmetic, reducing design time for the end user. To that end, our use of the dataflow paradigm is one of *coarse-grained* kernels. A well-known example of high-level dataflow operators from another domain is the graphics pipeline, shown in Figure 4.1.

4.2.2 Pagerank

We now illustrate in this section how a user would construct a hardware accelerator for a well-known algorithm, *PageRank* [49]. The reader will recall that the hardware accelerator is run in tandem with the main software application on the host system. A Green-Marl [36] implementation of the PageRank algorithm is shown in Listing 4.1.

4.2.3 Block Selection

The first step for the user is to analyze the structure of the algorithm and determine which sections will be accelerated in hardware. Through profiling or code analysis of the PageRank algorithm (Listing 4.1), the user could determine that calculation and update of the *val* variable (demarcated in bold in the figure) dominate the runtime of this algorithm. A coarse-grained DFG for the PageRank loop is shown in Figure 4.2.

Given the functionality described in the DFG, the user would then consult the GraphOps library to select blocks for parameterization:

Listing 4.1: PageRank algorithm implemented in Green-Marl.

```
1  Procedure pagerank(G: Graph, e,d: Double, max: Int;
2                     pg_rank: Node_Prop<Double>)
3  {
4    Double diff;
5    Int cnt = 0;
6    Double N = G.NumNodes();
7    G.pg_rank = 1 / N;
8    Do {
9      diff = 0.0;
10     Foreach (t: G.Nodes) {
11       Double val = (1-d) / N + d*
12         Sum(w: t.InNbrs) {
13           w.pg_rank / w.OutDegree()} ;
14       diff += | val - t.pg_rank |;
15       t.pg_rank <= val @ t;
16     }
17     cnt++;
18   } While ((diff > e) && (cnt < max));
19 }
```

The data fetch kernel implements a functional pattern in which computation is performed using neighbors of a vertex (lines 10-12). A kernel is needed to generate memory requests for these neighbor sets. From the library components enumerated in Section 4.3, the user first selects the *ForAllPropRdr* block to fetch the necessary neighbor sets from memory.

The arithmetic computation kernel implements the actual arithmetic or reduction upon the neighbor sets. To perform the reduction, the user selects the *NbrPropRed* block. The set of parameters used to customize this computation is exposed to the user.

The update kernel is self-explanatory. The user is left with a choice between which GraphOps block best fits. For PageRank, we note that the updates can be performed in-order, due to the independent nature of the update and the bulk synchronous iterative nature of the application (line 15). As a result, the *ElemUpdate* block is the superior choice. This block can be driven by the more optimized *UpdQueueSM*

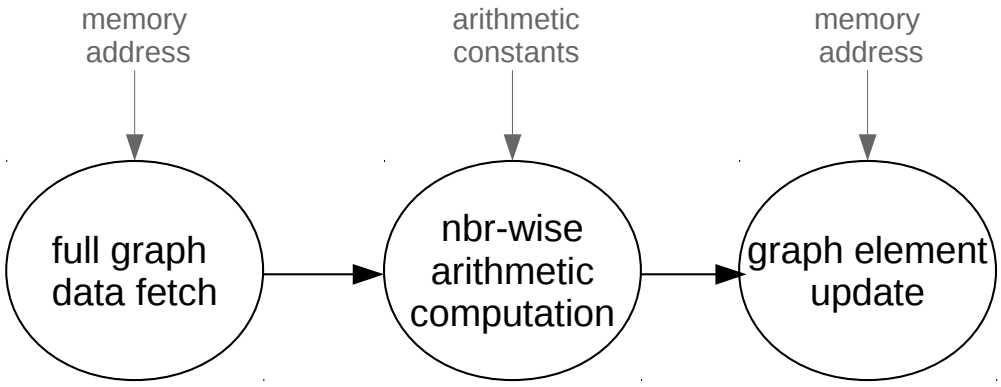

Figure 4.2: Coarse-grained dataflow graph for the PageRank core computation. The parameters that are fed into each kernel are exposed to users of the GraphOps library as customizable inputs.

control block instead of *CoalesceSM*. Utility blocks are added as needed: an *EndSignal* to track completion and one *MemUnit* block for each memory request interface.

The GraphOps library aims to be a toolbox for widely-used computational patterns. Although the blocks are parameterizable and offer flexibility, they cannot express every possible computational pattern. An advanced user may supplement chosen blocks with that user's own custom blocks to fully implement a custom design. Alternatively, a faster and simpler route to solution for a custom design could be to use the included GraphOps blocks and perform additional pre-processing and/or post-processing on the host.

4.2.4 Block Parameterization

Once a set of blocks has been selected, they must be properly configured to perform the specific computation on the correct graph in memory. In the GraphOps library, block parameters are implemented as static inputs that are driven over the PCIe bus by the host system. We detail the physical implementation of the system in Section 4.5. At this point, the designer would modify the logic to fit the exact computation. For example, a summation reduction could be replaced with a product.

One common parameter that is used by several different blocks is that of a memory

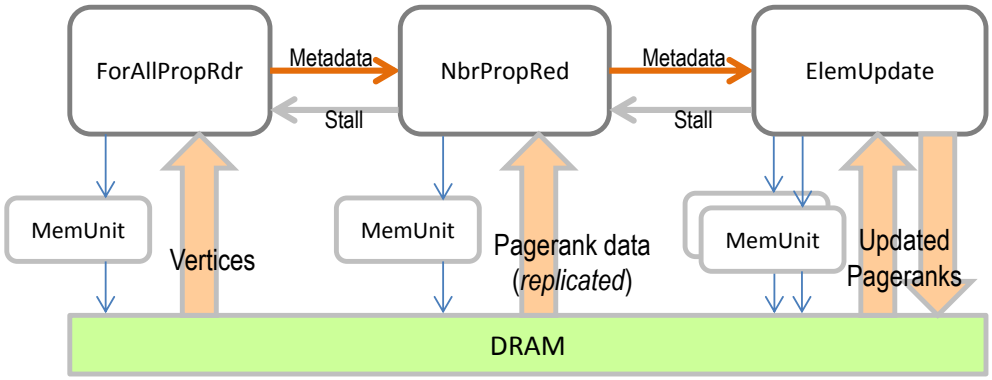

Figure 4.3: Detailed composition of GraphOps blocks to form the PageRank accelerator.

address. Any block that issues a memory request must have this base address from which to determine the memory location of specific graph elements. ForAllPropRdr, for instance, reads all pointers in the node array, then issues memory requests for property sets. It thus requires two address parameters, one each for the base addresses of the node array and property array. ForAllPropRdr also takes as an additional parameter, N, the number of vertices in the graph.

4.2.5 Block Composition

The final step in constructing the accelerator is to compose all blocks together to form a functioning system. Metadata outputs from each block are routed to the accompanying inputs on downstream blocks. Memory request signals are routed through MemUnits. *Done* signals are routed to the EndSignal block. Figure 4.3 shows a detailed block diagram of the blocks used in the PageRank algorithm. The Done signals and EndSignal blocks are omitted for clarity. Note that this figure closely mirrors the DFG of the PageRank computation, shown in Figure 4.2. This figure additionally highlights how the blocks interact with the memory system. For expressing block composition, a dataflow software compiler platform [51] is used to instantiate the blocks and wire the inputs and outputs.

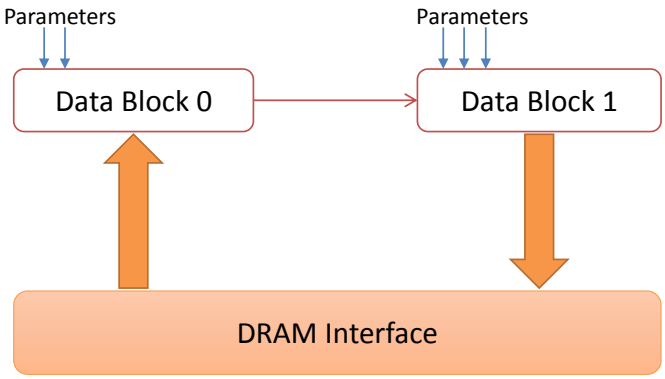

Figure 4.4: Two standard data blocks depicting their communication and memory interfaces. Note that each data block has a different set of parameters.

4.3 Enumeration of GraphOps

4.3.1 Categories of GraphOps

We divide the GraphOps library into three categories of functionality: *data-handling* blocks, *control* blocks, and *utility* blocks. In this section, we describe the three types of blocks and their general use cases. For each of the three categories of GraphOps blocks, we include at the end of its section a table which summarizes the blocks in that category.

Data blocks are the primary GraphOps components. They compose the main datapaths for the hardware accelerator. Most data blocks have at least one interface to the memory subsystem, either for reading or writing. Data blocks also contain the arithmetic LUTs defined by the algorithm, e.g. adders for a summation reduction. Finally, their interfaces include inputs and outputs for communicating metadata with other GraphOps blocks that precede or follow. A common example of this metadata are index pointers for the data arrays, used to generate data masks for incoming data streams. Data blocks operate on graph information expressed using the locality-optimized storage representation that we presented in Chapter 3. They are implemented using the MaxJ dataflow language. The GraphOps library includes 11 data blocks.

Figure 4.4 depicts a simple notional accelerator composed of two data blocks. The

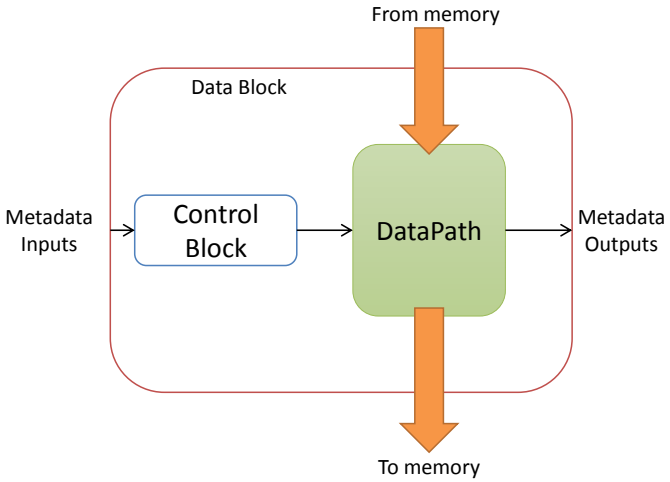

Figure 4.5: Control block state machines handle complex control logic in inside the data block.

wide arrows represent high bandwidth data streams to/from memory.

Control blocks handle intricate control operations that are not easily expressed in the dataflow paradigm. A common example of these operations is a control circuit that involves feedback paths. Such operations are traditionally implemented using the state machine abstraction, common in mainstream HDLs. We therefore implement control blocks as state machines using the MaxJ state machine language. These state machines are wrapped in their own sub-blocks and are themselves instantiated as controllers in data blocks. Five control blocks are included in the GraphOps library.

Figure 4.5 depicts a diagram of the described embedded control block architecture.

Utility blocks are additional logic blocks that do not operate on graph data directly, but rather handle necessary "bookkeeping" tasks like generating host interrupts and properly formatting memory requests before passing them on to the DRAM interface. These blocks, though often small and composed of fairly simple logic, are necessary for proper operation of the accelerator. Two utility blocks are included in the GraphOps library.

4.3.2 Data-Handling Blocks

We now enumerate the data blocks and describe their functions. For some of the blocks, we delve more deeply into details of their operation and diagram their internal hardware structures. Table 4.1 summarizes the set.

(i) **ForAllPropRdr** issues memory requests for all neighbor property sets across the entire graph. In order to do this, it first issues requests and reads in the entire node array. As the node array is being streamed in, the incoming row the node array data are used to issue memory requests for all sets of neighbor properties. Metadata about the requested neighbor properties are emitted to be processed by the subsequent block.

Many graph algorithms feature computation that is a function of a vertex's neighbor's properties. For example, any neighbor reduction algorithm fits naturally here, e.g. PageRank calculation [49].

Listing 4.2: Example algorithm snippet that is expressible using the ForAllPropRdr block.

```
1 Foreach (t : G.Nodes) {
2     Int nbrSum = Sum (w : t.Nbrs) {
3         w.nodeProp;
4     }
5 }
```

The ForAllPropRdr block assumes that neighbors for *every vertex* in the entire graph need to be read and processed (lines 1-2 in Listing A.1). This in itself is insufficient to warrant use of this block. The computation must also use a graph property of those neighbors (line 3), in addition to just reading them.

Figure 4.6 diagrams the hardware used to implement the block. Note the interface signals: two memory request outputs, one each for the node and property arrays and one data array input for the node array data. Control flow in the block is driven by the counters, which track the progress of node array requests.

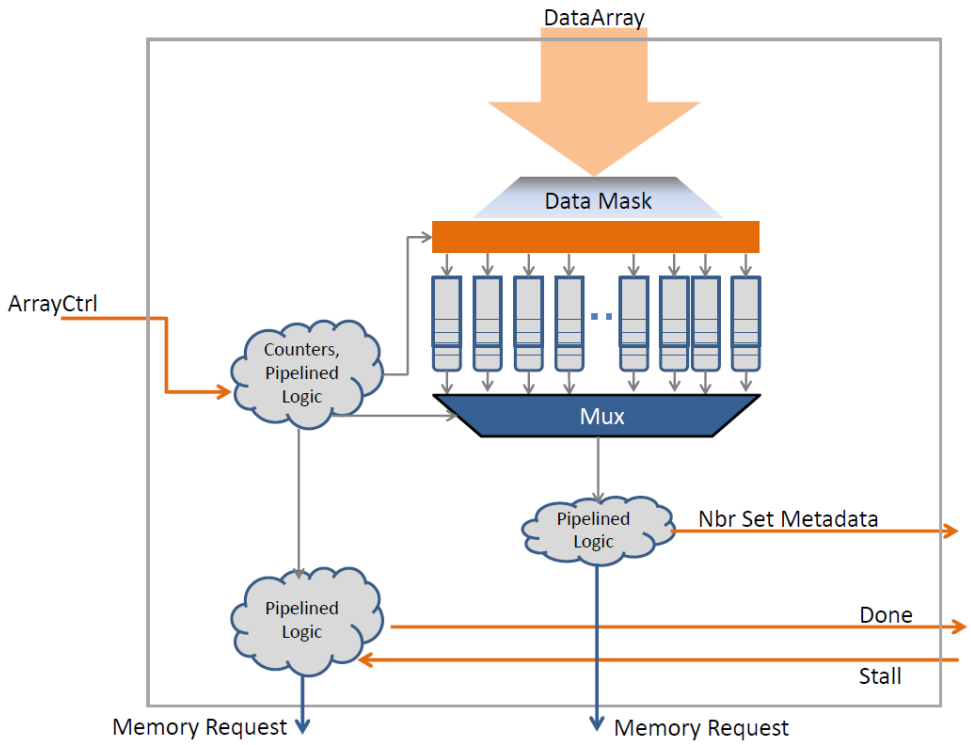

Figure 4.6: ForAllPropRdr: This data block takes in a graph property address and issues memory requests for all neighbor property sets.

Node array pointers enter through the DataArray input and are stored in an array of FIFO buffers. All node array elements are valid, except for elements which exceed the graph size N. These elements appear at the end of the node array, and the data mask enforces which elements proceed to be stored in the buffers.

Another counter generates control for a mux. Whenever an element is passed through the mux, special control logic sequentially "pops" that element from the FIFO buffer and uses it to generate a memory request for its neighbor sets. The full block operates continuously unless FIFOs reach an almost-full threshold. At this point, node array requests halt while the block continues to empty out the FIFOs by issuing requests for the property arrays.

(ii) **NbrPropRed** performs a reduction on the properties of a vertex's neighbor set. The block receives the neighbor property data as a data stream input from memory. Each set of neighbor properties is accompanied by a metadata packet as an input to the kernel from a preceding block (e.g. ForAllPropRdr). The metadata is used to mask the correct data elements from the incoming neighbor property data packets for inclusion in the reduction operation. For each neighbor property set, an accumulating reduction is performed on the data and the result is emitted as an output along with accompanying metadata.

The code example in Listing A.1 (line 2) demonstrates a summation reduction over neighbor property sets. The NbrPropRed block is used to execute the reduction for arbitrarily large sets of properties.

Figure 4.7 diagrams the operation of the block. The incoming metadata corresponds to the "Edgelist Ptrs" input required to mask out the correct elements from the incoming properties. Upon arrival, the metadata elements are buffered in FIFOs.

From the standpoint of robust execution, the input FIFOs serve a critical role because input rates of metadata and data packets may differ. Because of DRAM access latency, it is often the case that large numbers of metadata packets arrive before the data packets begin to arrive. As a result, the fullness of the FIFOs is used to generate a feedback /textitstall that is used to halt input from a previous block.

The buffered metadata pointers, along with a data packet control signal, are processed together to generate a dynamic mask for the incoming data elements. For every set of properties, the correct elements must be allowed into the reduction tree, and the accumulating result must be cleared before the next property set begins computation.

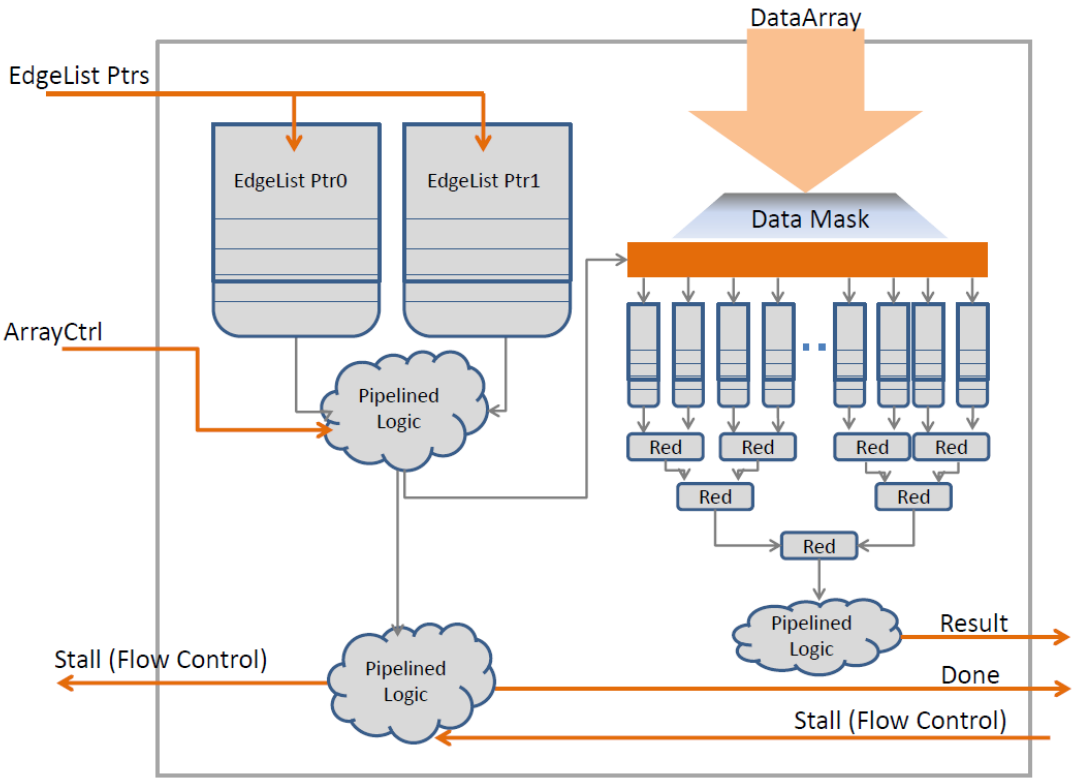

Figure 4.7: NbrPropRed: This data block performs reduction on an incoming set of values. This set of values is constrained by the metadata specified in the EdgeList Ptrs.

For data that are masked out of the calculation, an "identity" value is placed in the reduction tree in their place. The identity value is a value that does not change the result of the operation. This value varies based upon the type of reduction. For example, the identity value for a summation would be zero and for a multiplication would be one.

The data packet control signal is combined with a counter to determine when a given property set has completed computation. At this point, the result is passed out of the block as an output.

(iii) **ElemUpdate** updates graph property values. The unit receives a vertex reference and an updated value as input. It issues memory read requests for the element locations, performs the merging writes, and issues memory write requests for the newly updated values.

The ElemUpdate block can be used to update a single graph element. It is more common, however, for graph algorithms to iteratively update large portions of the entire graph. In such sitiuations, naive system-level implementation of the graph mutation protocol could easily lead to prohibitively poor performance.

We added a write-merging buffer into the ElemUpdate block to ensure performance when executing sequential updates to large data arrays. Although a simple optimization, this functionality, is not provided by low-level memory interfaces such as the memory controller used in our target hardware system.

The control logic necessary to implement the write-merging logic is a poor fit for the dataflow paradigm used in the ElemUpdate block. A more flexible control block is necessary to handle these details. In this case, we use the UpdQueueSM control block to handle data packet merging for multiple pending write requests. We describe the UpdQueueSM block in more detail later in this section.

Figure 4.8 diagrams the operation of the block.

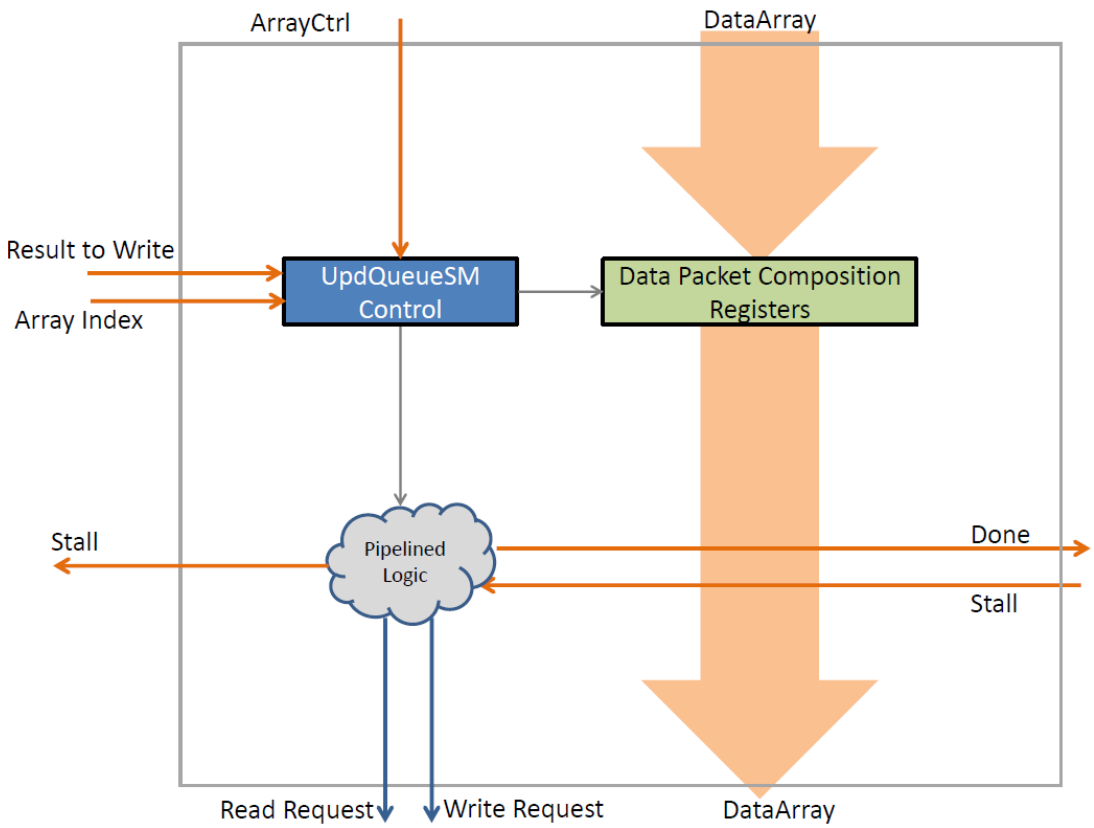

Figure 4.8: ElemUpdate: This data block updates a property value in the graph structure. The UpdQueueSM control block is used performs write-merging in order to reduce the number of update requests.

(iv) **AllNodePropRed** reads property values for the entire graph and performs a graph-wide reduction, conditional on a boolean filter. This block is useful in algorithms designed to collect a measurement that reflects all vertices in the graph.

For example, our implementation of graph *conductance* [9] needs to calculate a graph-wide summation of properties, conditioned on vertices that belong to a given membership group. Listing 4.3 shows a snippet of Green Marl code implementing part of the conductance computation that expresses this computation.

Listing 4.3: Sample of a code snippet that could be implemented using the AllNodePropRed block.

```
1 class = 2;
2 SumAlpha = Sum(u:G.Nodes) (u.class_member == class) {
3       u.alpha
4 };
```

The AllNodePropRed data block operates on the entire graph, so the summation domain (line 2) needs to be global. In this example, the addresses of the property arrays *class_member* (line 2) and *alpha* (line 3) are supplied to the block as parameters. The block hardware fetches both arrays.

User input in this block is required to complete the specification. For the example in Listing 4.3, the user would supply: (a) the reduction operation, (b) the conditional filter, and (c) the reduction argument (in this case, just the property itself).

(v) **NbrPropRdr** requests the properties of the neighbor set for a single vertex, which is supplied as input. The block uses an input FIFO buffer to queue the vertices that need to access neighbors. From this queue, vertices are used to generate memory requests for the node array which are streamed out to DRAM.

As node array data enters the block, the requested vertex is used as an index (into the node array) to generate locations into the neighbor property array. These locations are processed and used to generate a second memory request

for the neighbor properties. The neighbor property data stream is processed by a subsequent data block. The NbrPropRdr block emits the vertex processed and the property array pointers as metadata. This block is effectively the single-vertex version of ForAllPropRdr.

(vi) **SetReader** reads a set of graph elements from memory and emits them as a stream for processing in subsequent blocks. Many graph analytics algorithms are implemented in an iterative manner. In these algorithms, algorithmic state is often carried from iteration to iteration using intermediate working sets. Listing 4.4 shows pseudocode for a basic breadth-first search (BFS) or traversal. In the listing, each iteration begins by reading the to-be-explored *frontier set* for the breadth-first exploration (line 9).

Listing 4.4: Pseudocode of a basic BFS algorithm.

```
1  array nodes, edges
2  bool finished
3  set v_set, v_set_next = {root}
4  cur_level = -1
5
6  do {
7      finished = true
8      cur_level++
9      v_set = v_set_next
10     v_set_next = {}
11     foreach vertex v in v_set:
12         num_nbrs = nodes[v+1] - nodes[v];
13         nbrs = & edges [ nodes[v] ]
14         foreach nbr in nbrs:
15             if nbr.level == INF
16                 finished = false
17                 nbr.level = cur_level+1
18                 v_set_next.add(nbr)
19 } while (!finished)
```

Hardware in the SetReader block is composed primarily of counters which generate the read requests and FIFO buffers which hold the working set being read. The counters feature throttling logic to prevent buffer overflow in the FIFOs. This helps to limit the FIFO size required and conserve on chip memory resources. The main input is the data array from memory, and the outputs are the memory request channel and the stream of active graph elements to be processed by subsequent blocks.

(vii) **SetWriter** is the complementary block to SetReader. It accumulates a set of graph elements and writes them out to memory, recording the intermediate working set for the next iteration. In Listing 4.4, , each loop iteration adds a neighboring vertex to v_set_next if a specific condition is met (line 18), building up the frontier set for the next iteration.

SetWriter hardware is composed of data packet composition registers. Simple counters are used to determine where in a data packet an incoming element should be stored. Other counters track how many packets to send are outstanding and when new memory write requests should be issued. The main input is the graph element to be stored, and the outputs are a memory request and a data array representing the new set.

(viii) **NbrPropFilter** applies a filtering operation to sets of properties and emits a stream of vertices whose properties pass the filter. Such an operation is useful in a variety of contexts. For example, a social network algorithm that aims to enumerate users or neighbors that fit a given criterion (e.g. friends that live in New York) is fundamentally performing the computation described by this data block. In the BFS algorithm shown in Listing 4.4, vertices which pass the frontier condition (line 15) are captured and passed out of the block for further processing.

From a hardware standpoint, the NbrPropFilter block interface has inputs for a data array (of graph properties) and the accompanying edge pointers used to mask the data array. Edge pointer inputs enter the block and immediately generate memory requests for the property array. The reader should recall

from our discussion of the graph representation in Chapter 3 that the locality-optimized property arrays are the same length as the edge array. LO-arrays contain property data for properties residing on the destination of edges in the edge arrray.

We exploit this structure by using the same set of edge pointers to generate masks for the LO property arrays. As the property data enters the data block, the data is masked and then the filter operation is applied in a data-parallel manner. Elements which pass the filter have their corresponding edge values (neighboring vertices) written to an array of FIFO buffers. These FIFO buffers then use a round-robin approach to sequentially stream out the valid neigbhor vertices to the subsequent block.

(ix) **GlobNbrRed** performs an accumulating reduction across a number of neighbor property sets. The notion of a set of neighbors is a natural one in network analysis, and some graph analytics algorithms organize their computation along neighbor sets.

Listing 4.5 illustrates a simple example. The domain of operation is the entire graph–this could also be a subset of the graph. Within the domain, each vertex is exploring its neighbors conditionally, subject to constraints on line 4. For each vertex-neighbor pairing, a max reduction is performed on a user-specified metric–in this case, a quotient.

Listing 4.5: Sample of a code snippet that could be implemented using the GlobNbrRed block.

```
1 // bProp1, bProp2: boolean properties
2 // fP1, fP2: float properties
3 Foreach(s: G.Nodes) {
4     Foreach(t: s.Nbrs) (s.bProp1 && t.bProp2) {
5         <maxVal; from, to>  max=  <s.fP1/t.fP2; s, t>;
6     }
7 }
```

The GlobNbrRed data block takes in a stream of property values, along with accompanying metadata. The metadata is composed of edge pointers that constrain which property values are valid. Within the block, the user describes the computation necessary to process the individual sets. The output of this computation produces a value, and that value is fed into the reduction operation. The reduction operation is part of the template logic for the data block, and the result is passed out of the data block as an output.

(x) **VertexReader** issues memory requests for the node array pointers for a single vertex and emits edge pointers to downstreams blocks. This simple block can be used more generically to read other graph elements from any type of array.

(xi) **NbrSetReader** uses edge pointers to issue memory requests to a locality-optimized data structure, such as the edge array or a property array. Together, the VertexReader and NbrSetReader can be used to read arbitrary neighbor sets (e.g. list of neighbors, neighbor properties, etc) for any single vertex in the graph.

4.3.3 Control Blocks

Control blocks are implemented using the MaxJ state machine language, instead of the general dataflow language, and are therefore named with a trailing 'SM'. The reader should recall that control blocks are embedded inside data blocks. We now enumerate the control blocks and describe their functions. Table 4.2 summarizes the set.

(i) **QRdrPktCntSM** handles control logic for input buffers in the data blocks. A common use case occurs in the following situation: A metadata input dictates how many data packets belong to a given neighbor set. The data block needs to know how many data packets to process on behalf of the neighbor set before moving on to the next neighbor set. There is inherent feedback in this operation because the number of packets to process depends on the incidence of new

Block Name	Function	Complexity
ForAllPropRdr	Issue memory request for every node's neighbor set for a given graph property.	High
NbrPropRed	Perform reduction on the properties of a neighbor set. Stream out result.	High
ElemUpdate	Update property values in the graph. Works in tandem with a control block.	High
AllNodePropRed	Read propertie values for the entire graph and perform reduction.	High
NbrPropRdr	Request neighbor set properties for a single vertex.	Medium
SetReader	Reads a set of graph elements from memory and streams them out to following blocks.	High
SetWriter	Accumulates graph properties from kernel inputs, coalesces them, and writes to memory.	Medium
NbrPropFilter	Filters an incoming property stream and streams those vertices whose properties pass the filter.	Medium
GlobNbrRed	Performs an accumulating reduction on each neighbor set in a graph.	Medium
VertexReader	Issues memory requests for the vertex ID for a single vertex and streams out its edge pointers.	Low
NbrSetReader	Issue memory request for a locality-optimized set of data, either properties or an edge list.	Low

Table 4.1: Summary of the data-handling blocks.

packets in the data block. In this case, the QRdrPktCntSM block handles the counting of packets on the memory data input and instructs the data block when to move on to the next neighbor set. This unit is also used for flow control, emitting a *stall* output signal when metadata input buffers are nearing capacity.

(ii) **FifoKernelSM** is a wrapper for a standard FIFO block which we augmented to include an additional control signal – *dataReady*. *dataReady* states that the FIFO has data available to be read in the following cycle and should be a candidate for reading. This information is necessary to construct a sophisticated feedback-based parallel queue structure in the data blocks. Section 4.4 will detail the functionality of the parallel queue. Figure 4.9.(a) displays a diagram of the

interface for the control block.

(iii) **MemUnitSM** handles control logic for memory requests involving very large data sizes. The hardware platform underlying the GraphOps system may have constraints about size of request, so control logic is needed to transparently break up a large request and issue multiple requests, for example. The unit includes input buffering to prevent subsequent requests from being dropped while a large request is being issued. MemUnitSM blocks are designed to be embedded inside the MemUnit utility block, presented later in Section 4.3.4.

The GraphOps library provides the following two control blocks for optimally updating graph elements in memory:

(iv) **UpdQueueSM** handles control logic for *sequential* updating of a graph property when the entire graph is being updated. This control block exploits the sequential nature of the updates by composing maximally-sized data blocks, minimizing the number of update memory requests necessary. The hardware logic uses a write-merging coalescing scheme implemented using bit vectors. The size of the buffer is parameterizable.

(v) **CoalesceSM** also handles logic for updating graph properties. However, this version does not assume in-order vertex updates and thus does not coalesce writes as efficiently. CoalesceSM is appropriate for random updates – however, large numbers of random updates will likely be the primary source of performance degradation. Best-effort coalescing buffer logic is built into the control unit by using a least-recently used write buffer. The oldest element in the buffer is evicted and written to memory, generating a request, while recently-updated vectors are maintained in the buffer. Every buffer is appended with a timestamp that generates eviction upon expiration. The size of the buffer is parameterizable.

Block Name	Function	Complexity
QRdrPktCntSM	Handles control logic related to tracking incoming data packets counts.	Medium
FifoKernelSM	Wraps a standard FIFO block, augmenting the output with additional metadata.	Low
MemUnitSM	Handles control logic involved with breaking up large memory requests into smaller chunks.	Medium
UpdQueueSM	Handles control logic for sequentially updating a property for the entire graph.	Medium
CoalesceSM	Handles control logic for updating property values. Does not assume sequential updates. Highly complex.	High

Table 4.2: Summary of the control blocks.

4.3.4 Utility Blocks

Utility blocks provide logic to properly interface with the memory system and the host platform. They are typically implemented using the general dataflow language and function as standalone units that are not embedded in other data blocks, but are used to handle some type of simple bookkeeping task. We now enumerate the two included utility blocks. Table 4.3 summarizes the utility blocks.

(i) **EndSignal** is an optional block that monitors end-of-operation signals, called *done*, from all data blocks and tracks completion. The block issues a special interrupt request when all units are finished to declare end of execution on the host system.

(ii) **MemUnit** provides a simplified memory interface to the data blocks, primarily acting as a wrapper for the Maxeler memory API. This abstraction takes three simple inputs: address, size, and control. Other functionality is added within the data block, such as additional memory profiling information and monitoring logic for when to generate end-of-execution interrupts. Finally, this block includes the embedded MemUnitSM control block for handling very large memory requests.

Block Name	Function	Complexity
EndSignal	Monitors end-of-operation signals from all data blocks and issues interrupt to host when done.	Medium
MemUnit	Provides a simplified memory interface to the data blocks.	Medium

Table 4.3: Summary of the utility blocks.

4.4 GraphOps Implementation Details

Buffering and Rate Matching

Like most computing systems based on streaming, proper matching of throughput rates and adequate buffering of data are critical to achieving correct execution in the GraphOps library. The PageRank case study in Section 4.2 provides a good example. This accelerator makes use of the NbrPropRed data block, diagrammed in Figure 4.7. During execution, each memory request from the previous block (ForAllPropRdr) corresponds to a metadata input in NbrPropRed. Because of the difference in throughputs between the memory controller and the data blocks, metadata inputs would quickly overwhelm moderately sized input buffers without any intervention.

One method used to mitigate this effect is to *throttle* the memory-requesting data blocks, reducing their effective output rate. In the ForAllPropRdr block (Figure 4.6), for instance, the memory request and metadata emission rates are reduced by a factor of two. The throttling factor is easily modified using a single user-accessible parameter. This design allows NbrPropRed to better match the memory throughput and prevent overflow in its input buffers.

In addition to throughput throttling, the relatively large DRAM latency also causes large numbers of requests to be emitted before the first data packet is received. To address this hazard, the user should size the input buffers adequately to account for this. For the example in the PageRank case study, we used a value of 8K for the input buffers in addition to a system-wide flow control scheme which we now describe.

Flow Control

Despite best efforts to match throughputs across different blocks, it is difficult to

account for all corner cases of execution. For example, in a highly irregular graph, a few vertices have a very large number of neighbors, while most vertices have very few. Returning to the PageRank case study, the NbrPropRed block may spend a disproportionate amount of time performing a reduction of one particularly large neighbor set. During this computation, the metadata inputs (edge array pointers) would not advance, since the head of the queue is actively being processed. As a result, the buffers would overflow. This is one example. In a dataflow computation in which block latencies are irregular, there are many unpredictable ways for one data block to "get ahead" of another, causing incorrect execution.

Our flow control scheme involves the use of *stall* signals which are received and propagated by every data block. A downstream data block emits a signal "upstream" towards the preceding block. A high stall input in an upstream block indicates that the downstream block is approaching a state in which it can no longer accept new inputs. Upon reception of a stall, the upstream block halts execution at the next convenient point and propagates the stall further upstream. In addition to stalls received from downstream blocks, all data blocks generate their own stall signals. The generated stall signals are often derived from input buffers nearing capacity. These values are or'd with the downstream stall input in order to generate the overall stall output to send upstream (*Stall* output).

Figure 4.3 illustrates the stall flow control signals in the PageRank accelerator. The data block diagrams in Section 4.3 show the generation and propagation schematics for those examples.

FifoKernelSM Parallel Queue Figure 4.9 shows a diagram of a parallel queue constructed using the FifoKernelSM control block. This structure addresses a situation commonly encountered in the GraphOps parallel processing framework. As described in the NbrPropRed architecture, incoming memory data packets carry multiple potential elements with arbitrary masks defining which of them are buffered and which are discarded. In this scenario, a control block is needed which allows for round robin selection from among only those buffers which have data. When a buffer is selected, a dequeue signal needs to be automatically sent to only that buffer, leaving other

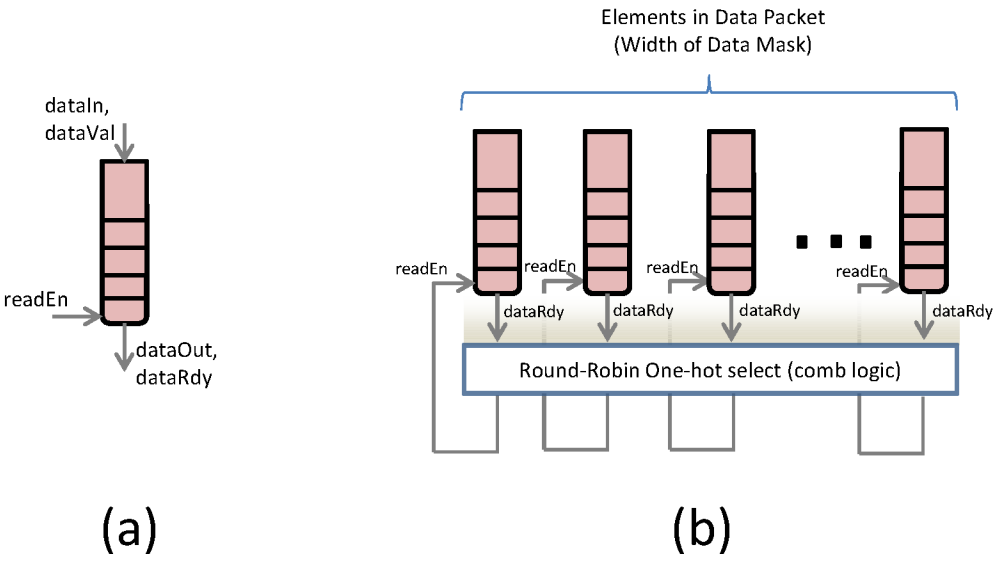

Figure 4.9: (a) Diagram for the FifoKernelSM control block. This block augments the standard FIFO primitive with an additional control signal called *dataReady*. This signal is necessary to efficiently construct the parallel queue. (b) Architecture of the parallel queue constructed using the FifoKernelSM control block. An array of *dataReady* signals are used as input to a combinational logic circuit that generates a one-hot *readEnable* output based on the available inputs.

queues intact. This tight feedback operation necessitates the use of special control logic.

The FifoKernelSM enables this automated parallel queue by providing a *dataReady* control output, in addition to standard FIFO interface signals. The dataReady vector goes through a combinational logic block which selects one entry from among the buffers with data available. This one-hot *readEnable* vector is fed back into the buffers as the dequeue input.

4.5 Experiments and Analyses

This section describes the experiments and analyses undertaken to qualify the GraphOps library. We experimented by implementing a variety of accelerators using the GraphOps blocks. We used the accelerators in tandem with host applications written in C. We used these accelerators to perform computations on a variety of data sets. Results were compared with pure software applications and another CPU-based streaming system.

We chose the C language because it allows for low-level control of the memory layout, necessary to efficiently construct the locality-optimized storage representation (Chapter 3) used by the GraphOps blocks. In addition, the software API for the FPGA co-processing routines were well-supported in C.

4.5.1 Target Hardware System

The target hardware system for our hardware accelerators is a platform developed by Maxeler Technologies [57]. Figure 4.10 shows a diagram of the prototyping system.

The target FPGA is a Xilinx Virtex-6 (XC6VSX475T). The chip has 475k logic cells and 1,064 36 Kb RAM blocks for a total of 4.67 MB of block memory. For all graph accelerators, we clocked the FPGA at 150 MHz. The FPGA is connected to 24 GB of DRAM via a single 384-bit memory channel with a max frequency of 400 MHz DDR. This means the peak line bandwidth is 38.4 GB/s. Note that the peak bandwidth is achievable largely because the memory controller is optimized for bursty

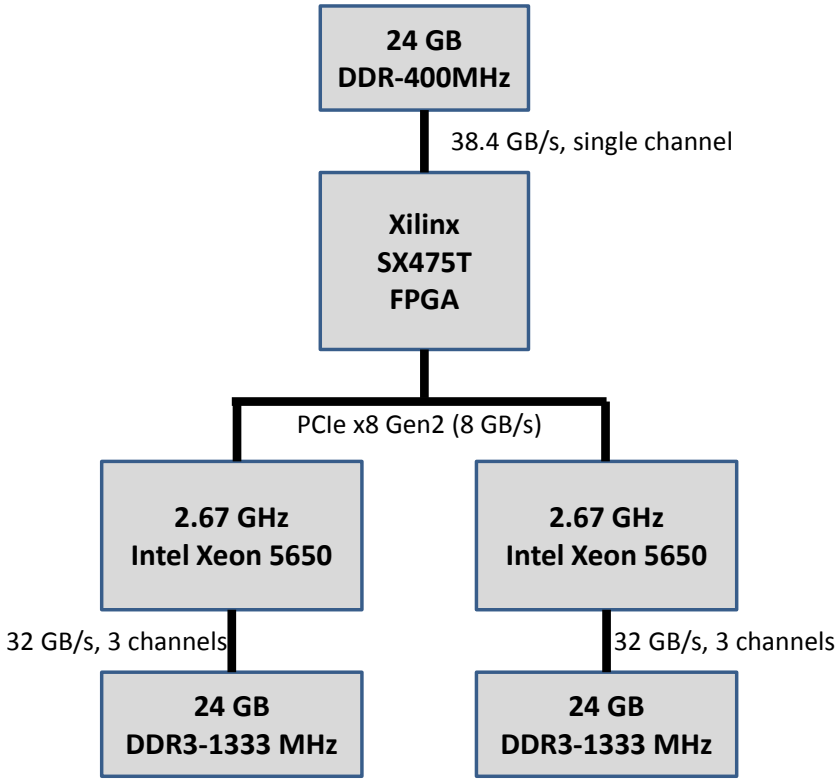

Figure 4.10: Diagram of the Maxeler prototyping system.

memory access. Memory accesses are limited to a minimum of four quadwords of a 384 bit memory channel. This effectively means that data accesses are performed at a granularity of 1.5 Kbits.

The FPGA is connected via PCIe x8 to the host processor system. All applications were implemented in multi-threaded C (OpenMP) and executed on the Xeon host processors. The host system has two 2.67 GHz Xeon 5650 multi-processors, each having six multi-threaded cores making a total hardware thread count of 24. Each of the two processors has a peak line bandwidth of 32 GB/s and three 64-bit channels to memory.

The GraphOps blocks are implemented on top of the MaxCompiler framework, also developed by Maxeler [51]. The tools provide an HDL and interfaces for accelerating development of dataflow and streaming accelerators. The higher level language

is compiled to generate VHDL.

We return the reader's attention to the memory interface of the FPGA, emphasizing two key characteristics:

First, our target hardware system features only a single memory channel. This channel is shared by all request queues via multiplexing. There are no ordering guarantees among the request queues. In bandwidth-constrained accelerators with multiple requesters, the single-channel constraint with lack of request prioritization is a fundamental bottleneck. We will quantitatively explore this effect in Section 4.5.5.

Second, the memory controller is constrained to accesses of at least 1.5 Kbits. This relatively coarse granularity is sub-optimal for individual graph element accesses and updates – most graph elements are between 32 and 96 bits. The GraphOps library tolerates this constraint through the use of the locality-optimized data structure described in Chapter 3. In addition, the graph update control blocks described in Section 4.3.3 use best-effort write merging logic to mitigate this effect and maximize locality in updates.

4.5.2 Applications

We use the GraphOps blocks to implement accelerators for six analytics applications:

- PageRank [49]: Computes the PageRank score of each vertex in the graph
- Breadth-First Search (BFS): Computes the number of hops to every vertex in the graph from a given root vertex
- Conductance [8]: Computes the conductance of a given subset of vertices. The conductance metric describes how quickly a random walk from within the subset has a non-zero probability of arriving outside that subset.
- SpMV: Sparse Matrix-Vector Multiplication
- Single-Source Shortest Path (SSSP): Computes the shortest distance to every vertex in the graph from a given root vertex.

Algorithm	Resource Usage (%)		
	FF	LUT	BRAM
pagerank	33.2	21.3	24.5
bfs	32.1	18.8	36.6
conduct	25.7	16.0	20.6
spmv	33.0	20.6	24.5
sssp	30.7	18.8	37.0
vcover	23.4	14.7	19.4

Table 4.4: Resource usage for each accelerator.

- Vertex Cover [50]: Finds an approximation of the minimum vertex cover. The vertex cover is the set of edges that covers as many vertices as possible. This implementation uses a greedy algorithm.

Table 4.4 breaks down the on-chip resource usage of each accelerator. The data confirms that none of the accelerators are bound by chip resources on the Xilinx Virtex-6 FPGA. The user should note that overly-liberal use of FIFO buffers in the data blocks can quickly impose undue pressure on the BRAM resources [1]. Input buffers need to be large enough to allow for temporarily bursty throughput behaviour, such as that experienced in the initial phase of computation before the first DRAM response. Beyond the temporary bursts, the throttling and flow control schemes described in Section 4.4 prevent overflow and ensure system stability. On the other hand, buffers should be responsibly bounded in order to constrain resource usage and reduce place-and-route effort. Typical buffer sizes in these accelerators range from 2K to 12K elements elements. Typical element widths are either 32 or 64 bits.

Table 4.5 lists the GraphOps blocks and the number of MemUnits used in each accelerator. The high-level nature of the GraphOps blocks helps to constrain the number required to implement complex operations. The most complex applications that we implemented were the BFS and SSSP graph traversals, and each use five blocks. The number of MemUnits corresponds exactly to the number of unique memory request interfaces required by the accelerator. One of the interfaces is trivial, as

[1] BRAM usage is heavily dependent on the sizing of the numerous FIFO buffers in the design. These FIFOs are often larger than required.

Algorithm	GraphOps Blocks	
	MemUnits	Blocks Used
pagerank	5	ForAllPropRed, NbrPropRed, ElemUpdate
bfs	6	NbrPropRdr, NbrPropFilter, ElemUpdate, SetWriter, SetReader
conduct	4	AllNodePropRed, NbrPropRdr, NbrPropRed
spmv	5	ForAllPropRed, NbrPropRed, ElemUpdate
sssp	6	NbrPropRdr, NbrPropFilter, ElemUpdate, SetWriter, SetReader
vcover	3	ForAllPropRed, GlobNbrRed

Table 4.5: Enumeration of GraphOps blocks for each accelerator. The EndSignal block, used in all accelerators, is not included.

it only activates when used to issue the end-of-execution interrupt request.

From the lists of blocks used, it is evident that certain blocks are more commonly used than others. This supports our heuristic observation that there are common computational paradigms in an interesting set of graph analytics algorithms. The GraphOps library makes building accelerators for these types of algorithms more accessible.

4.5.3 Evaluation: Accelerator Performance

We first evaluate the performance of GraphOps-based accelerators by comparing the accelerators with baseline software implementations.

The baseline comparison software applications for this study are written in multi-threaded C++ (OpenMP). These versions were generated using the Green-Marl graph compilation framework [36]. As described in that paper, the performance of the generated C++ is competitive with hand-optimized versions of the applications. All applications are run on the Intel Xeon processors on the host system, described at the beginning of this chapter.

The GraphOps-accelerated results are provided by implementing the same applications in C. The "inner loop" computation is accelerated using the FPGA, as we described for PageRank in Section 4.2. The time to transfer the graph data to/from the host and the FPGA is not included. Any initial data transfer time can be amortized by long-running applications on large data sets in the FPGA memory.

Recall from our discussion of the locality-optimized storage representation in Section 3.3 that the LO-arrays achieve locality via replication. These arrays must be initially prepared outside the main loop then updated as part of the main loop. Our accelerator results cover two different design points. The first set of results measures only the run-time of the accelerator, which is denoted as *GraphOps* on the figures. The second version takes into account the overhead of updating the LO-arrays – these we denote as *GraphOps+Scatter*.

Figure 4.11 depicts performance throughput for GraphOps accelerators compared with software implementations. Experiments are performed using synthetic uniform graphs with average degree of eight. The x-axis is the number of vertices in the graph. This means that the number of edges in each workload is eight times the number of vertices in that workload. The y-axis is throughput, measured in millions of edges per second, or MEPS (higher is better). MEPS is a measure of the number of edges "processed" per second. Because the notion of processed work for each algorithm is different, MEPS should not be compared across accelerators, but rather used as a relative scale for different systems within one accelerator.

Software versions do not uniformly show improved performance beyond eight threads. This is likely due to the Xeon processors exhausting their off-chip memory bandwidths. We therefore show multi-threaded results up to a maximum of eight threads.

All of the graph analytics algorithms displayed are bound by memory bandwidth. This is evident in Figure 4.11, as all of the lines begin to roughly approach a horizontal asymptote with increasing graph size. For smaller graph sizes, we see strong caching effects in the software versions. The data sets fit partially or fully in the CPU cache, greatly improving throughput, until graph sizes of around 2M vertices. Note that both of these effects apply partially to the GraphOps+Scatter version, because the LO-array maintenance routine is a normal software function and therefore susceptible to cache effects. In contrast to the software versions, the pure GraphOps implementation shows no caching effects, as there is no significant re-use happening on the FPGA. As a result, the throughput for GraphOps-based accelerators is roughly constant for all graph sizes.

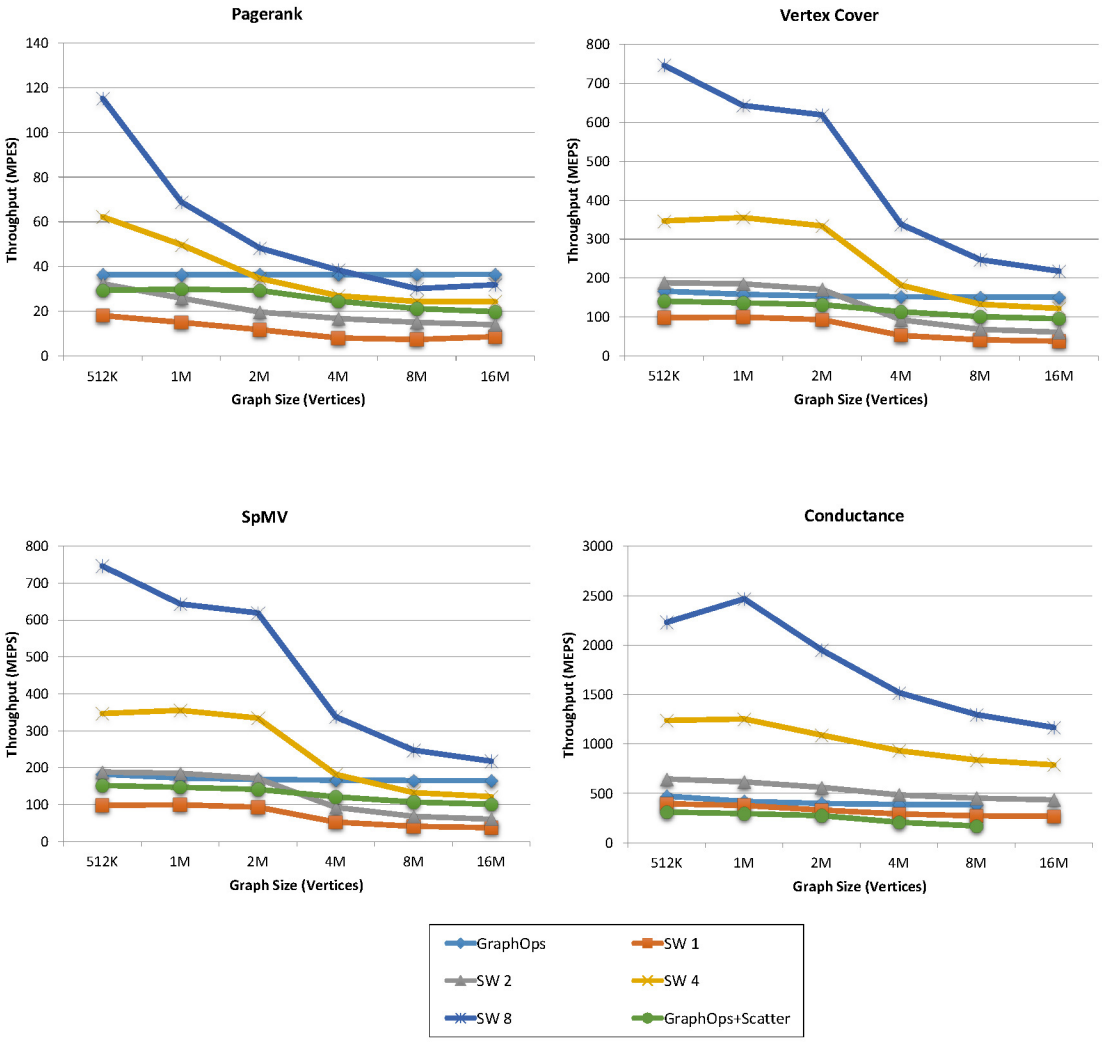

Figure 4.11: Throughput performance of GraphOps accelerators compared with software implementations.

From a performance standpoint, the GraphOps-accelerated applications deliver more performance than the single-threaded software versions but are generally outperformed by the eight-threaded versions. This performance gap is explained simply by the difference in available peak memory bandwidth. The FPGA peak bandwidth is about 38 GB/s while the CPU bandwidth is about 32 GB/s per socket (total 64 GB/s). The FPGA memory controller is also limited to one wide DRAM channel, as discussed. Superior software performance at eight threads also results from access to three memory channels in addition to the greater overall bandwidth. Additional memory channels increase the capacity of the memory system to serve data requests and hold them outstanding. This increased ability to serve additional requests holds true, even if the number of wires dedicated to data transfer is equal to that of a wide single-channel architecture, such as the one used in our target platform. Using calculations based on the number of data words requested during FPGA execution runs (excluded due to space constraints), we determine that the single memory channel causes heavy queuing and arbitration delay for the multiple memory requesters. We will detail these experiments later in this chapter.

Having acknowledged the constraints imposed by the memory subsystem, we now study the energy efficiency of the GraphOps-based accelerators. Figure 4.12 depicts a comparison of the energy efficiency of the accelerators versus single- and eight-threaded versions of the software implementations. The efficiency metric is throughput per Watt. When normalized for power consumption, the FPGA-based GraphOps accelerators are at least three times more efficient than the corresponding eight-threaded version. The reader should again note in this figure that the GraphOps+Scatter design point is susceptible to caching effects with increasing working set sizes.

4.5.4 Evaluation: Comparison with Software Streaming Framework

We continue the evaluation of the GraphOps library by comparing against a graph processing framework called X-Stream [52]. The X-Stream framework is an apt choice

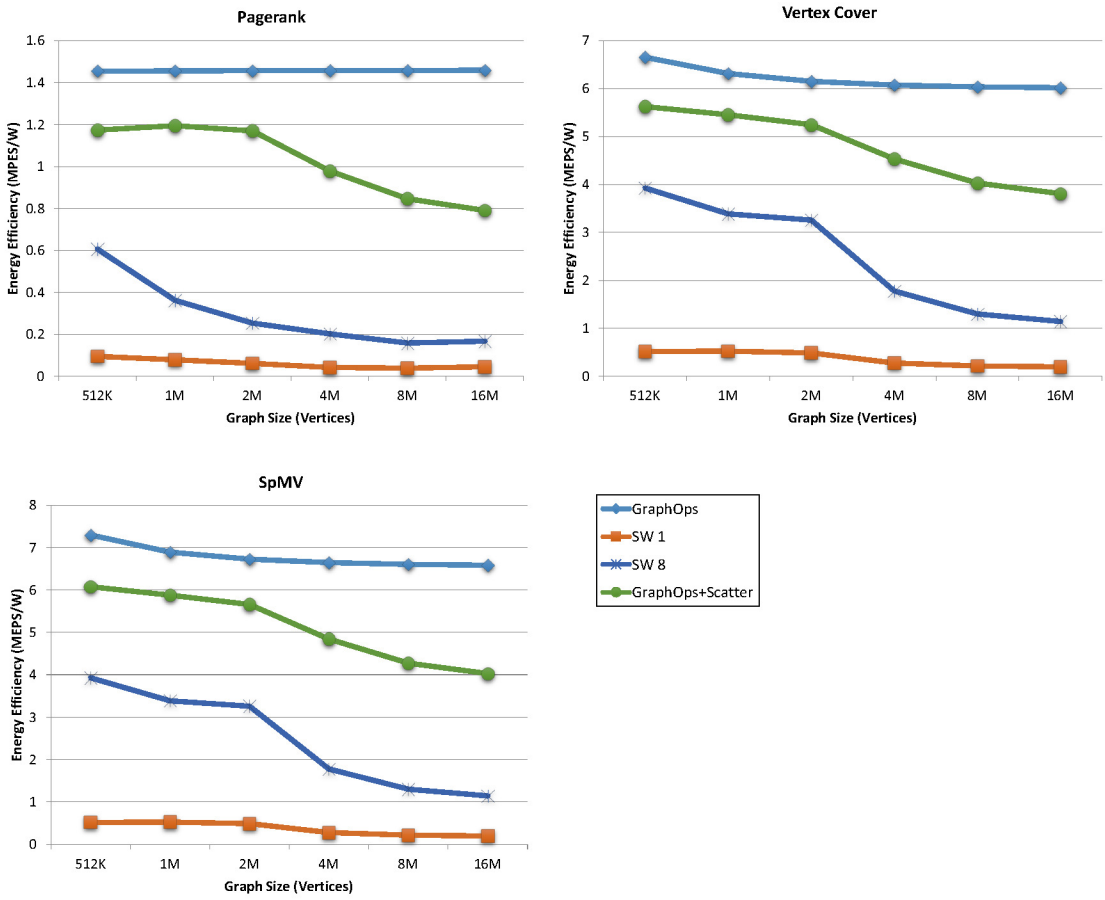

Figure 4.12: Efficiency of GraphOps accelerators measured in throughput/W, compapred with software implementations.

because, similarly to the locality-optimized storage representation used in GraphOps (Chapter 3), X-Stream is built around maximizing sequential streaming of graph data while minimizing random access. The underlying observation is that sequential memory bandwidth is usually much higher than random access bandwidth, particularly for graphs that do not fit in main memory and require disk access.

X-Stream retains the scatter-gather programming model used in the well-known distributed graph framework Pregel [42]. Like Pregel, state is also stored in the vertices. X-Stream differs from Pregel in that its implementation is edge-centric, rather than vertex-centric, and it streams entire edge lists, rather than performing random access into the edge list.

Listing 4.6: Pseudocode for the edge-centric scatter-gather used in X-Stream. This system is implemented in software.

```
1  edge_scatter(edge e)
2    send update over e
3
4  update_gather(update u)
5    apply update u to u.destination
6
7  while not done
8    for all edges e
9      edge_scatter(e)
10   for all updates u
11     update_gather(u)
```

Listing 4.6 depicts some high-level pseudocode modeling the X-Stream operation. The scatter phase iterates sequentially over edges and generates updates (lines 1-2). After this phase, a "shuffle" intermediate phase broadcasts updates to their destination. Then, the gather phase iterates over updates and applies them (lines 4-5). Execution proceeds iteratively until no new updates are generated. Sequential access to the edges comes at the cost of random access to the vertices – updates can be generated for any vertex in the entire graph. X-Stream mitigates this randomness by partitioning the vertex array and caching the partitions (or keeping them in memory

Name	Vertices	Edges	Description
amazon0601	475K	3.4M	Amazon product co-purchasing network
cit-Patents	3.8M	16.5M	Citation network among US Patents
wiki-Talk	2.4M	5M	Wikipedia talk (communication) network
web-BerkStan	685K	7.6M	Web graph of Berkeley and Stanford
soc-Pokec	1.6M	30.6M	Pokec online social network

Table 4.6: Data sets used in X-Stream comparison study.

for very large graphs). This effectively allows X-Stream to only perform random access to closer (faster) memory stores. We refer the reader to the X-Stream paper [52] for a more thorough description of the X-Stream system.

Our locality-optimized storage representation is similar to X-Stream primarily because of the shared emphasis on maximizing use of high-bandwidth architectures. LO-arrays generate locality through data replication, similarly avoiding random access into the large edge/property data structure.

We chose a variety of different data sets for this comparison study. Table 4.6 enumerates these data sets. All data sets are used courtesy of the Stanford SNAP project. [40]. The readers should note that both "real world" power-law graphs (e.g. soc-Pokec) as well as more uniform graphs (e.g. cit-Patents) are represented in the data.

Figure 4.13 compares execution time of the GraphOps and X-Stream frameworks for the workloads discussed. The metric is execution run time, so lower is better. Both frameworks were executed on the same machine, described in Section 4.5.1. X-Stream is a software-only framework and used the host system CPU and memory resources, ignoring the FPGA.

The figure provides a breakdown of total execution time for three GraphOps-based accelerators. GraphOps systems compare favorably with the X-Stream applications, despite the slightly inferior total bandwidth available to the GraphOps accelerators. The reader should first note that the preparation and maintenance overhead for the LO-arrays, denoted as *graphops (scatter)*, is a small fraction of the overall run-time for each implementation.

The most problematic data sets for the spmv and PageRank accelerators are wiki-Talk and cit-Patents. Both of these accelerators are dependent upon efficient access to vertex neighbors' properties. Wiki-Talk and cit-Patents are sub-optimal because they have relatively small average degrees, 2.1 and 4.3 for wiki-Talk and cit-Patents respectively. Recall from our discussion of our target hardware system in Section 4.5.1 that memory accesses are constrained to rather large data blocks. Indeed, we designed the locality-optimizing storage representation presented in Chapter 3 with this consideration in mind. Because our system is fundamentally designed around fetching neighbor property sets using large data blocks, small-degree data sets such as wiki-Talk and cit-Patents waste much of the bandwidth designated for that purpose. This observation is also borne out by the superior performance of GraphOps on the soc-pokec-relationships data set, given its average degree of 19.1. The conductance accelerator is built around streaming the entire graph, as opposed to a neighbor traversal, and is thus not subject to this small-degree effect.

4.5.5 Bandwidth Utilization Experiment

Our throughput results show a performance discrepancy – highly multi-threaded software applications outperform FPGA-based accelerators. Our earlier analyses speculated that a key reason for this performance gap was the availability of multiple memory channels in the software applications and only a single wide channel for the FPGA-based accelerators.

In this section, we perform an experiment to quantify the effects of memory request queuing and switching delays. The switching delays are due to the throughput discrepancy between the faster GraphOps blocks and the slower memory controller. The switching delays are imposed by multiple memory interfaces sharing the single memory channel in our target hardware system. We build upon the case study we described in Section 4.2 by using PageRank as a model to closely observing these effects. Figure 4.14 diagrams the memory interfaces, denoted by arrows, in the PageRank accelerator and their uses in the algorithm.

To collect data about how heavily each memory interface is using the memory

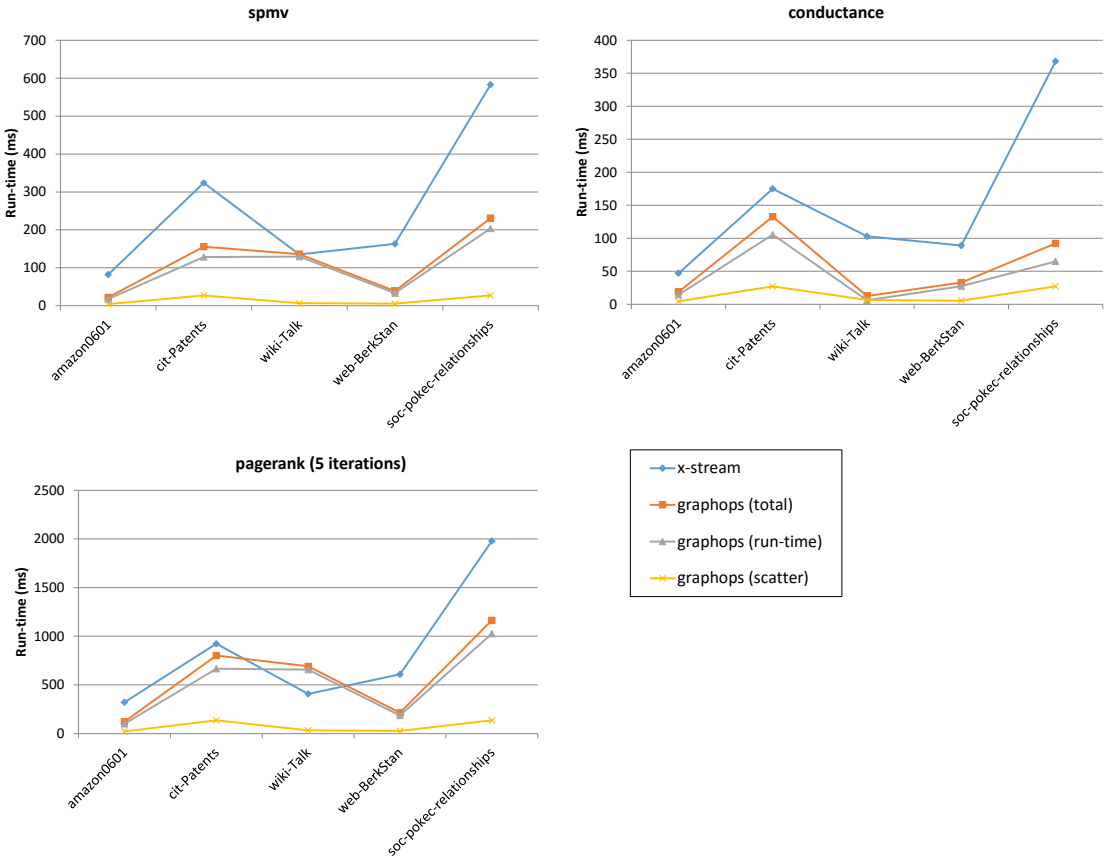

Figure 4.13: Run-time comparison of GraphOps and X-Stream.

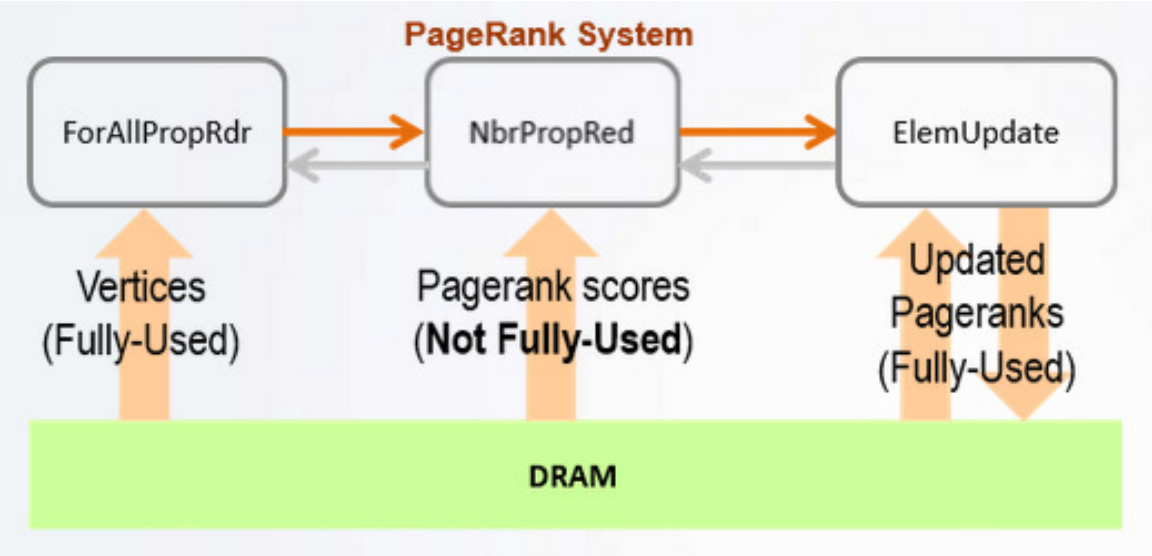

Figure 4.14: Memory interfaces used in the PageRank accelerator. The EndSignal memory interface is omitted, as it issues only one request.

subsystem, we instrument the MemUnit utility blocks with hardware counters. These counters track incoming data requests sent by a GraphOps block and accumulate a running total of the number of data packets requested by the memory interface. Note that this instrumentation is concerned with how many *physical* data packets are accessed in memory, as opposed to the number of graph elements. For example, if two graph elements are requested and they happen to be located on a block boundary spanning two blocks, the counters would record a size of two for the two physical data blocks requested. This is despite the fact that most requests of that size would be entirely located within one physical data block.

After instrumentation, we then execute the application using the accelerator, ensuring that the runtime is long enough to capture the steady state behaviour of the memory interfaces. Figure 4.15 displays the breakdown of the total memory interface usage, both reads and writes, for PageRank. For this application, it is clear that the locality-optimized property array, which is denoted "L-O Array" and populated with PageRank scores, dominates use of the memory bandwidth. This data structure corresponds to memory accesses by the NbrPropRed ("Nbr Reducer") block in Figure 4.14. The observed memory request data in this accelerator raises some key

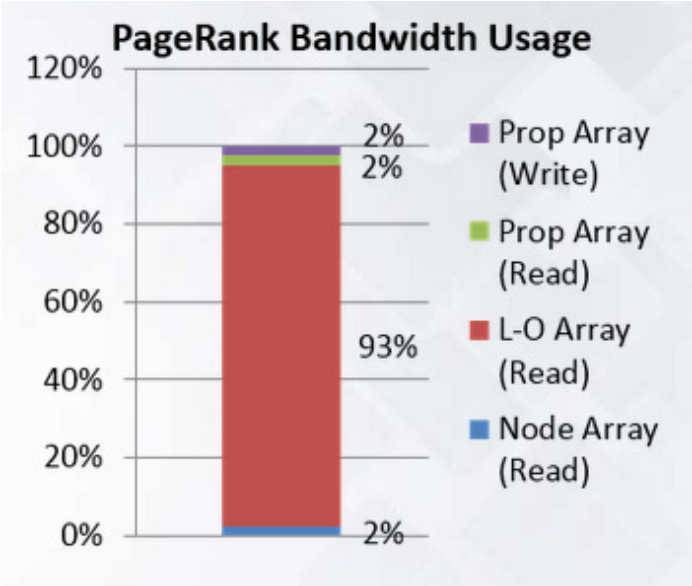

Figure 4.15: Breakdown of total bandwidth by memory interface in the PageRank accelerator.

research questions:

- How well are the memory interfaces which issue the majority of the memory requests, such as the LO-array, served by the single memory channel on our target hardware platform?
- How much of the available bandwidth is used by the accelerator?
- What are the bandwidth-related factors constraining performance as measured by throughput?

Calculations

The results collected show a line bandwidth of about **6.4 GB/s** of data incoming to the FPGA.

We run the accelerator on a synthetic uniform graph with an average degree of eight. Given the proportion of memory accesses allocated to the replicated LO-array, we focus this study on access bandwidth to that data structure by calcuating expected throughput performance if total bandwidth were efficiently allocated to only this data

structure. This allows us to quantify the effect of multiplexing the controller across different MemUnits.

The storage layout of our graph elements is configured to have 32 edges/vertices per data block, meaning an average of four neighbor sets fits into one data block. The pattern for the number of data blocks requested is therefore approximately the following: 1, 1, 1, 2, 1, 1, 1, 2,... From this pattern, we expect the average number of data blocks requested per neighbor set to be about 1.25.

Because an average of four neighbor sets (8 neighbors) fits into one data block, the average usage rate for each request is 0.25 data blocks. Given 0.25 data blocks per request, the average percentage of each request that is used by the GraphOps block is:

$$\frac{0.25 \quad \text{data blocks used per block requested}}{1.25 \quad \text{data blocks requested per neighbor set}} = 20\%\text{usage rate per request}$$

Given the line bandwidth of 6.4 GB/s, the expected throughput utilized by the GraphOps block for LO-array access is:

$$6.4 \text{ GB/s} * 20\% \text{ usage rate} = 1.28 \text{ GB/s expected throughput}$$

This is the approximate theoretical peak throughput for the PageRank accelerator on our target hardware. We call it approximate, because this figure accounts only for the LO-array, and not the other minor data structures. Looking closely at the throughput results in Figure 4.11, however, the actual PageRank throughput falls well short of this mark. The steady state throughput for the GraphOps accelerator is about 37 MEPS, which corresponds to a throughput of about 220 MB/s. This represents about 1/6 of the available theoretical throughput.

The primary cause for the difference in performance is the switching penalty we have previously described. Although the vast majority of memory requests are issued by one of the memory interfaces, the memory channel must switch among the three other interfaces also issuing requests. This causes the queue associated with the NbrPropReducer to stall unnecessarily.

The secondary cause is inherent to the graph being studied and it effects overall bandwidth degradation: Issuing one memory request per neighbor set limits the size of the requests and prevents optimizations at the memory controller level. This factor is largely responsible for the FPGA being unable to approach the theoretical peak bandwidth.

An ideal architecture for graph analytics would be one that optimizes for the number of memory channels at the expense of high-performing sequential bursty access. This configuration better matches the memory access behavior most-often associated with graph analytics algorithms.

4.5.6 Power Usage

Figure 4.16 illustrates the power usage for each of the accelerators. The data is collected using the Xilinx Power Analyzer (XPA) tool [19]. For all accelerators, analysis is done post placeandroute. Physical constraints are also supplied via a PCF file. Default XPA switching rates are used, instead of a simulation vector input.

Power usage is fairly similar for all accelerators, with I/O functions drawing roughly 60% of the power. All of the accelerators incur less than 25W of power, roughly a quarter of the TDP for the Xeon 5650 host CPU. These figures support the notion of FPGAs as energy-efficient graph analytics accelerators, especially when paired with efficient asynchronous data movement and data re-use in the accelerator memory.

4.6 Related Work

There has recently been significant interest in the area of accelerating graph analytics using co-processors.

There is prior research in the area of using FPGAs to efficiently process many types of algorithms, including those in the graph analytics space [20, 14, 21].

Betkaoui et al [6] accelerated the graphlet counting algorithm on an FPGA using an optimized crossbar and custom memory banks. This approach demonstrated high

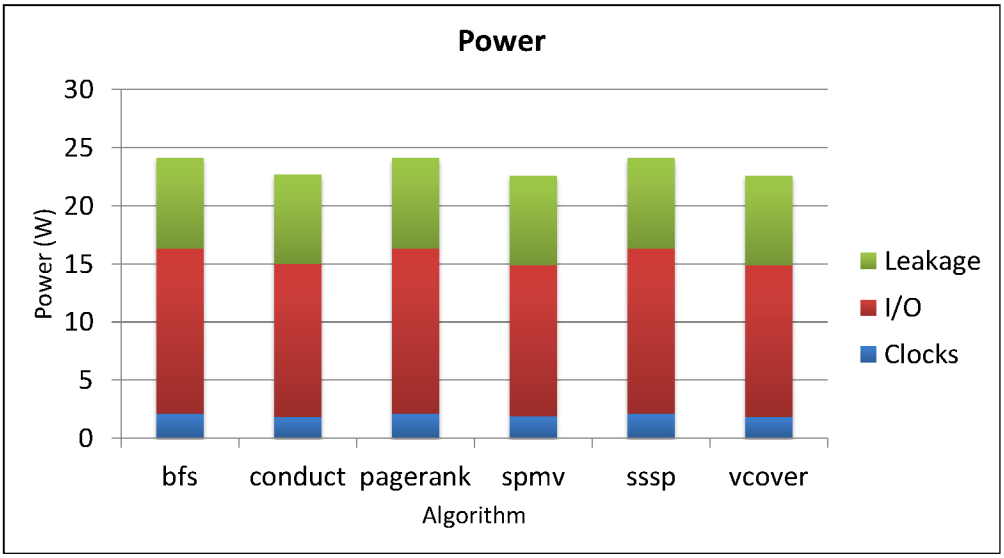

Figure 4.16: Primary power usage for each accelerator. Minor categories are omitted (under 500mW).

performance via a custom memory subsystem, but significant expertise was required on the part of the designer. Their work differs from GraphOps, because their framework requires an end user to express his algorithm as a vertex-centric kernel, similar to Pregel [42]. They do the work of mapping it to a Convey hardware system.

Nurvitadhi et al. [47, 59] proposed another FPGA-based graph processing system, called GraphGen, focusing on vertex-centric graph algorithms. In GraphGen, the user supplied the complete RTL specification of the update function which executes on all vertices in the graph. This is a radically different design philosophy than the one used in GraphOps. The GraphGen system compiled the specification onto an application-specific graph processor and memory system for the input provided. GraphGen also limits algorithms to be expressed in a vertex-centric manner and requires creation of a platform-specific hardware abstraction layer before being used.

DeLorimier et al. [21] proposed another FPGA-based framework, called GraphStep. GraphStep organized computation by placing the entire graph in the FPGA BRAMs and overlaying a lightweight network across the chip. Accessing graph elements from the BRAMs provided massive improvements in bandwidth and access

latency, increasing performance. The prominent limitation of this approach is a constraint of representable graph size to networks that only fit in on-chip BRAMs (on the order of a few MBs as of 2015).

Significant research has also been done on using GPUs to accelerate graph anlytics. Hong et al [35] used a warp-centric programming approach to more efficiently utilize the GPU's memory system for breadth-first traversal. Merril et al [44] improved on this work by using an efficient prefix sum to generate efficient fine-grained tasks for their traversal implementation. More recent work has seen the GPU used for a wider array of algorithms. MapGraph [26] uses a dynamic scheduling scheme and a custom memory pattern to ensure efficient memory behavior. GPU solutions, in general, are very programmable and provide significant performance benefits. However, energy efficiency is necessarily sacrificed, given the power requirements of high performance GPUs. For both FPGA and GPU approaches, great effort has been taken to mitigate the challenging memory behavior encountered with graph analysis.

There has also been some work in the area of using a streaming paradigm to process graphs. Ediger et al [25] used a dynamic graph representation called STINGER to extract parallelism and enable streaming processing. STINGER provides a general dynamic data structure that can be used with a variety of architectures, but still requires a knowledgeable developer to compose the full algorithms.

Roy et al [52] proposed X-Stream, an edge-centric approach whereby edges are streamed from slow storage and updates to vertices are performed efficiently to fast, partitioned storage. We fully compared the GraphOps system to X-Stream in Section 4.5.4. It is interesting to note that both GraphOps and X-Stream use a unique storage representation of the graph in order to maximize spatial locality. This enables both systems to make better use of abundant sequential memory bandwidths.

Chapter 5

Conclusions

This dissertation presents a suite of composable high-level building blocks that enable the simple design of energy-efficient and performant graph analytics accelerators.

We describe, via experiences with previous iterations of the graph acceleration hardware, potential pitfalls in using reconfigurable hardware to develop graph analytics accelerators. First, our experiences confirm the necessity of there being compatibility between the target hardware platform and the choice of hardware language used to develop the system. In our case, Verilog is shown to limit verification options in our hardware environment. Second, we describe how hardware-level considerations such as place-and-route effort and resource contention must be first-class considerations when designing a hardware synthesis system. In our case, an elegant and scalable compiler-level representation proves unworkable because it generates hardware descriptions too complex for the place-and-route tool.

Hardware accelerators are usually optimized for energy-efficiency on their specified task. One way to provide this naturally is through reliance on a high-throughput parallel design. Most data storage in the graph analytics domain were developed to be used in general purpose computers and therefore make different assumptions about various tradeoffs in memory access, such as the efficiency of fine-grained data access and the availability of high-performance hardware caches.

We address this shortcoming by proposing a novel graph representation that is optimized for the increased spatial locality crucial for hardware accelerators. Our

data representation is particularly well-suited for memory controllers in which the memory interfaces are very wide and the number of memory channels is limited. Such configurations are more prevalent in accelerator memory architectures and less so in general purpose architectures.

With this context established, we present an enumeration of the GraphOps library. We select certain representative components to highlight and present these in detail, using them to demonstrate the design principles and functionality of GraphOps-based accelerators. We further demonstrate the use case for GraphOps by presenting PageRank as a case study.

Finally, we qualify the work presented by comparing GraphOps-based hardware accelerators with two different software frameworks. The first comparison is against multi-threaded software implementations of the various algorithms. Results show that the hardware accelerators are performant against the software, especially considering limitations in the memory subsystem. We also compare against a software-based streaming system for graph analytics. GraphOps performs favorably, outperforming the software system for most of the algorithms. We conclude with an experiment delineating the source of the majority of the performance deficiencies in GraphOps-based accelerators: contention for a limited single-channel memory controller.

Final Thoughts

We believe that the original promise of FPGAs, in which the performance of hardware is provided along with the customizability of software, is still outstanding. There is much work yet to be done before significant performance in FPGAs is delivered to an end user in the absence of hardware expertise. This is especially true in domains where the computation paradigm is not ideally suited for the FPGA computation fabric, e.g. fine-grained embarassingly parallel computation on limited-size datasets with significant hardware re-use.

The work in this dissertation aimed to bridge this performance-programmability gap for the domain of graph analytics by providing pre-designed high-level blocks to the end user. We found that this approach provided many benefits, but, earnestly, is quickly constrained by the specific hardware platform that must be targeted – specifically, the memory system of that platform. We also believe, after this research,

that the high level block approach is still more limiting than would be ideal for a system which aims to execute *all* types of graph analytics codes. Additionally, an argument can easily be made that the burden placed on the hardware designer is still too significant for the GraphOps library to be widely deployed. We believe that the most evident method to bridge this gap is to choose a lower level of abstraction with a more standard interface/ISA and leverage compiler technology to provide higher level programming abstractions to software engineers and hardware designers.

Appendices

Appendix A

GraphOps-based Pagerank Implementation

We present a sample of the GraphOps library by again using Pagerank as a driving application. This appendix will be presented in sections, with different blocks of the Pagerank accelerator each represented and explained in a single section.

Pagerank is represented in GraphOps using three different data blocks, as well as a few control blocks and utility blocks. We refer readers interested in full implementations of the entire GraphOps library to our online repository at `http://github.com/tayo/graphops`.

ForAllPropRdr

The code below illustrates ForAllPropRdr.

Listing A.1: MaxJ code for the ForAllPropRdr block.

```
1  package pagerank.kernels;

   import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
   import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
5  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
   import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
   import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEArray;
   import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.
       DFEArrayType;
9  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.IO.*;
   import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count;
   import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
   import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.
       OffsetExpr;
   import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Reductions;
   import com.maxeler.maxcompiler.v2.utils.MathUtils;
   import pagerank.GC;
17
   public class ForAllPropRdr extends Kernel {

     public ForAllPropRdr(KernelParameters parameters) {
21     super(parameters);
       DFEVar cnt = control.count.simpleCounter(64);
       // Scalar Inputs
       DFEVar uDVal = io.scalarInput("uDVal", dfeUInt(32));
25     DFEVar dbgStartCnt = io.scalarInput("StartCnt", dfeUInt(64));
       DFEVar nodeAddr    = io.scalarInput("nodeAddr", dfeUInt(GC.scalWidth));
       // repAddr should be the address of the replicated property array
       DFEVar repAddr    = io.scalarInput("repAddr", dfeUInt(GC.scalWidth));
29     int numNodesWidth  = 32;
       DFEType numNodesType = dfeUInt(numNodesWidth);
       DFEVar scalN       = io.scalarInput("NumNodes", numNodesType);
       DFEVar begin    = cnt.eq(dbgStartCnt);
33     DFEVar endN     = scalN+1; //account for end pointer for last node
```

```
        // Types
        int arSize = GC.PKTWIDTH/GC.NODEWIDTH;
37      DFEType nodeType = dfeUInt(GC.NODEWIDTH);
        DFEArrayType<DFEVar> arrayType = new DFEArrayType<DFEVar>(nodeType,
            arSize);

        // Inputs from DRAM
41      NonBlockingInput<DFEArray<DFEVar>> nodeArray_In =
          io.nonBlockingInput("nodeArray", arrayType, constant.var(true), 1,
              DelimiterMode.FRAME_LENGTH, 0, NonBlockingMode.NO_TRICKLING);
        DFEArray<DFEVar> nodeArray = nodeArray_In.data;
45      DFEVar arrayCtrl           = nodeArray_In.valid;

        int endBrstWidth = numNodesWidth - GC.nPBBits;
        DFEVar endBrst = endN.slice(GC.nPBBits, endBrstWidth).cast(dfeUInt(
            endBrstWidth));
49      DFEVar endPkt  = endN.slice(GC.nPPBits, GC.pktsPBBits).cast(dfeUInt(GC.
            pktsPBBits));

        // Feedback stall input from reducer
        NonBlockingInput<DFEVar> outputStall_In =
53        io.nonBlockingInput("outputStall", dfeBool(), constant.var(true), 1,
              DelimiterMode.FRAME_LENGTH, 0, NonBlockingMode.NO_TRICKLING);
        DFEVar outputStall = Reductions.streamHold(outputStall_In.data,
            outputStall_In.valid,
                   dfeBool().encodeConstant(false));
57
        // Counter to track number of packets
        Count.Params pktCntParams = control.count.makeParams(GC.pktsPBBits)
            .withEnable(arrayCtrl);
61      Counter pktCounter = control.count.makeCounter(pktCntParams);
        DFEVar pktCnt = pktCounter.getCount();
        // Counter tracking number of bursts arrived
        int brstCntbits = endBrstWidth;
65      Count.Params brstCntParams = control.count.makeParams(brstCntbits)
            .withReset(begin)
            .withEnable(pktCounter.getWrap());
        DFEVar brstCnt = (control.count.makeCounter(brstCntParams)).getCount();
69
        // Node-array fifos
```

```
        DFEVar numNodesIdx = endN.slice(0, GC.nPPBits).cast(dfeUInt(GC.nPPBits))
            ;
        DFEVar goodBrst = brstCnt <= endBrst;
73      DFEVar goodPkt  = arrayCtrl & ((brstCnt<endBrst)|(pktCnt<=endPkt));
        DFEVar lastPkt  = arrayCtrl & brstCnt.eq(endBrst) & pktCnt.eq(endPkt);
        DFEVar notLastPkt = goodBrst & goodPkt & ~lastPkt;
        DFEVar[] lastPktVal = new DFEVar[arSize];
77      DFEVar[] pktMask = new DFEVar[arSize];

        Memory <DFEVar> [] nodeFifos = new Memory[arSize];
        DFEVar[] fifoOutputs = new DFEVar[arSize];
81
        // Counter to throttle emission of requests: every 2^(tnBits) cycles
        int tnBits = 1; // for tnBits=1, emit request every two cycles, etc
        Count.Params emitCntParams = control.count.makeParams(tnBits)
85          .withEnable(constant.var(true));
        // use wrap signal of this counter to determine when valid to emit
            request
        Counter emitCounter = control.count.makeCounter(emitCntParams);

89      //give time for data to propagate through fifos
        DFEVar arrayCtrlDD = stream.offset(arrayCtrl, -2);
        DFEVar dataArrived = Reductions.streamHold(constant.var(true),
            arrayCtrlDD,
                                                   dfeBool().encodeConstant(
                                                       false));
93      DFEVar emit = dataArrived & emitCounter.getWrap() & ~outputStall;

        OffsetExpr finishedLoopLen = stream.makeOffsetAutoLoop("finishedLoopLen"
            );
        DFEVar finished = dfeBool().newInstance(this);
97
        // Counter to choose element to select in mux
        int muxSelBits = MathUtils.bitsToAddress(arSize);
        Count.Params muxSelCntParams = control.count.makeParams(muxSelBits)
101         .withEnable(emit & ~finished);
        Counter muxSelCounter = control.count.makeCounter(muxSelCntParams);
        DFEVar nodeFifoRdEn = muxSelCounter.getWrap() & ~finished;

105     // read- and write-pointers
        int fifoWidth = 32;
```

```
        int fifoDepth = 512;
        int fifoPtrBits = MathUtils.bitsToAddress(fifoDepth);
109     Count.Params ramRdPtrParams = control.count.makeParams(fifoPtrBits) //
            RdPtr
            .withEnable(nodeFifoRdEn);
        Count.Params ramWrPtrParams = control.count.makeParams(fifoPtrBits) //
            WrPtr
            .withEnable(arrayCtrl);
113     Counter ramRdPtrCounter = control.count.makeCounter(ramRdPtrParams);
        Counter ramWrPtrCounter = control.count.makeCounter(ramWrPtrParams);
        DFEVar ramRdPtr = ramRdPtrCounter.getCount();
        DFEVar ramWrPtr = ramWrPtrCounter.getCount();
117     DFEType fifoElemType = dfeUInt(fifoWidth);

        for (int i = 0; i < arSize; i++) {
          // Generate masks
121       lastPktVal[i] = lastPkt & (i < numNodesIdx);
          pktMask[i]    = notLastPkt | lastPktVal[i];

          // Fifos to store incoming nodes
125       nodeFifos[i] = mem.alloc( fifoElemType , fifoDepth );
          nodeFifos[i].write( ramWrPtr,
              nodeArray[i].slice(0,fifoWidth).cast(fifoElemType),
              pktMask[i] );
129       fifoOutputs[i] = nodeFifos[i].read(ramRdPtr);
        }

        DFEVar muxSel   = muxSelCounter.getCount();
133     DFEVar muxSelP1 = muxSel + 1;
        DFEVar ovflow   = muxSel.eq(arSize-1);
        DFEVar procOvf  = stream.offset(nodeFifoRdEn, -1) & ~nodeFifoRdEn;
        DFEVar node0Ar  = control.mux(muxSel, fifoOutputs);
137     DFEVar node1Ar  = control.mux(muxSelP1, fifoOutputs);
        DFEVar node0    = ~procOvf ? node0Ar : stream.offset(node0Ar, -1);
        DFEVar node1P1  = ~procOvf ? node1Ar : node0Ar;  //node0Ar should be the
            zeroth entry
        DFEVar node1    = node1P1 - 1;
141     DFEVar noNbrs   = node1.eq(node0-1); //account for the case where a node
            has no nbrs
```

```
        // Count number of requests emitted, and stop when reached number of
            nodes
        Count.Params rCntP = control.count.makeParams(numNodesWidth)
145         .withEnable((emit&~ovflow) | procOvf);
        DFEVar rCnt = (control.count.makeCounter(rCntP)).getCount();
        DFEVar reqCntEn = rCnt < scalN;
        Count.Params reqCntParams = control.count.makeParams(numNodesWidth)
149         .withEnable(reqCntEn & ((emit&~ovflow) | procOvf));
        DFEVar reqCnt = (control.count.makeCounter(reqCntParams)).getCount();

        // Throttling:
153     // Use the reqCnt (requested), rec'd (elemSoFar) to determine whether to
             pause
        // requesting (prevent the nodeFifos from overflowing for large data
            sets)
        // elemsSoFar:  number of data that have entered the kernel
        // reqCnt: number of requests emitted (one per data)
157     int threshHi = GC.threshHi;  //hysteresis: if ouststanding elems are >
            threshHi, then stall
        int threshLo = GC.threshLo;  //  if num elems in fifos drop below
            threshLo, then start again

        int brstInc     = GC.brstInc;   //brst grp size to request at once
161     int brstIncBits = MathUtils.bitsToAddress(brstInc);
        DFEVar elemsSoFar   = (brstCnt << GC.nPBBits).cast(numNodesType);
          //throttling
          DFEVar numDataInFifos = elemsSoFar - reqCnt;
165       OffsetExpr stallLoopLen = stream.makeOffsetAutoLoop("stallLoopLen");
          DFEVar stall = dfeBool().newInstance(this);
          DFEVar beginStall = ~stall & (elemsSoFar>reqCnt) & (numDataInFifos >
              threshHi);
          DFEVar endStall = stall & (numDataInFifos < threshLo);
169       DFEVar stallReg = Reductions.streamHold(beginStall, beginStall|
              endStall,
                   dfeBool().encodeConstant(false));
          stall <== stream.offset(stallReg, -stallLoopLen);
          DFEVar stallPrev = stream.offset(stall, -1);
173       DFEVar doneStall = stallPrev & ~stall;
        DFEVar brstNumInGrp = brstCnt.slice(0,brstIncBits).cast(dfeUInt(
            brstIncBits));
        DFEVar almostDone   = brstNumInGrp.eq(brstInc-1); //last brst in grp
```

```
        DFEVar moreToRd     = endN > (elemsSoFar);
177     DFEVar reqMore      = (~stall & almostDone & moreToRd & (arrayCtrl &
            pktCnt.eq(0))) |
              (doneStall & moreToRd); //
        DFEVar reqStart     = begin;

181     DFEVar done   = reqCnt.eq(scalN);
        finished <== stream.offset(done, -finishedLoopLen);

        // Determine when the kernel has finished processing
185     DFEVar knlDone = done;
        DFEVar knlDonePrev = stream.offset(knlDone, -1);
        DFEVar doneVal  = knlDone &  knlDonePrev;
        Count.Params sendDoneParams = control.count.makeParams(32)
189       .withEnable(doneVal);
        DFEVar sendDoneCnt = (control.count.makeCounter(sendDoneParams)).
            getCount();
        DFEVar sendLength = uDVal*2;
        DFEVar sendDone = knlDone & (sendDoneCnt < sendLength);
193     // output to Done kernel
        io.output("idle", knlDone, dfeBool(), sendDone);

        // Output to MemUnit: read node array
197     DFEType reqAddrType = dfeUInt(GC.brstNbits);
        DFEType reqSizeType = dfeUInt(GC.sizeBits);
        DFEVar nodeAddrBase = nodeAddr.cast(reqAddrType);
        DFEVar req_brst_addr = nodeAddrBase +
201            brstCnt.cast(reqAddrType) +
               almostDone.cast(reqAddrType);
        DFEVar req_size = constant.var(reqSizeType, brstInc);
        DFEVar req_en = reqStart | reqMore;
205     // + is concat
        io.output("memReq", req_size+req\_brst\_addr+req_en, dfeRawBits(GC.
            memReqWidth));

        // Output to MemUnit: read replicated property array, for the reducer
209     DFEVar repAddrBase = repAddr.cast(reqAddrType);
        int buf = fifoWidth - GC.nPBBits;
        int prefixZeros = GC.brstNbits - buf;
        DFEVar zeros = constant.var(dfeUInt(prefixZeros), 0);
213     DFEVar propBrst0 = (zerosnode0.slice(GC.nPBBits, buf)).cast(reqAddrType);DFEVar
```

```
            propBrst1 = (zerosnode1.slice(GC.nPBBits, buf)).cast(reqAddrType);
        DFEVar propBrsts = propBrst1-propBrst0+1;
        DFEVar req_size_rep      = propBrsts.cast(reqSizeType);
        DFEVar sendPtrs          = ~done & ((emit&~ovflow) | procOvf) & ~noNbrs;
217     DFEVar req_en_rep        = sendPtrs;
        DFEVar req_brst_addr_rep = repAddrBase + propBrst0;
        // + is concat
        io.output("memReqRep", req_size_rep+req_brst_addr_rep+req_en_rep,
221       dfeRawBits(GC.memReqWidth));

        // Output to reducer
        DFEVar propPkts    = propBrsts << GC.pktsPBBits;
225     DFEVar propPkts_   = propPkts.slice(0, GC.numPktsBits);
        DFEVar nodePtr0_   = node0.slice(0, GC.ePtrWidth);
        DFEVar nodePtr1_   = node1.slice(0, GC.ePtrWidth);
        int ptrDataWidth = fifoWidth + GC.numPktsBits + GC.ePtrWidth + GC.
            ePtrWidth;
229     // + is concat
        io.output("repPtrData", reqCnt + propPkts\_ + nodePtr1\_ + nodePtr0\_,
            dfeRawBits(ptrDataWidth), sendPtrs);

        flush.onTrigger(constant.var(false));
233   }
```

NbrPropRed

The code below illustrates NbrPropRed.

Listing A.2: MaxJ code for the NbrPropRed block.

```
package pagerank.kernels;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.SMIO;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEArray;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.
    DFEArrayType;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Mem.
    RamWriteMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.WrapMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.IO.*;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.
    OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Bitops;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Reductions;
import pagerank.GC;
import pagerank.blocks.*;
import com.maxeler.maxcompiler.v2.utils.MathUtils;

public class NbrPropRed extends Kernel {

  // return value is select control of mux
  private DFEVar prArb(int width, DFEVar req, DFEVar base) {
    DFEVar doubleReq = (req+req).cast(dfeUInt(2*width)); //+ is concat
    DFEVar diff = doubleReq - base.cast(dfeUInt(2*width));
    DFEVar doubleGrant = doubleReq \& ~diff;
    DFEVar grant = doubleGrant.slice(0, width) | doubleGrant.slice(width,
        width);
    return grant; // return value can be RawBits, so no need to cast
  }
```

```
34    private DFEVar reduceOp(DFEVar x, DFEVar y) {
        return (x+y);
      }

38    public NbrPropRed(KernelParameters parameters) {
        super(parameters);
        DFEVar cnt = control.count.simpleCounter(64);
        // Scalar Inputs
42      DFEVar uDVal  = io.scalarInput("uDVal"   , dfeUInt(32));
        DFEVar prTerm = io.scalarInput("prTerm"  , dfeFloat(8,24));
        DFEVar d      = io.scalarInput("d"       , dfeFloat(8,24));

46      // Types
        int arSize = GC.PKTWIDTH/GC.PROPWIDTH;
        // propType48:  [16-bit degree | 32-bit prop value]
        DFEType propType48  = dfeRawBits(GC.PROPWIDTH); // prop entries are 48
            bits
50      DFEType propType    = dfeFloat(8,24); //old: dfeInt(GC.PROPWIDTH);
        int propWidth       = 32; // floating point values are 32 bit
        DFEArrayType<DFEVar> arrayType = new DFEArrayType<DFEVar>(propType48,
            arSize);
        DFEType ePtrType = dfeUInt(GC.ePtrWidth);
54      DFEType parentType = dfeUInt(GC.nodeWidth);

        // Inputs from DRAM:  property arrays of the nbr
        NonBlockingInput<DFEArray<DFEVar>> propArray\_In =
58        io.nonBlockingInput("propArray", arrayType, constant.var(true), 1,
              DelimiterMode.FRAME\_LENGTH, 0, NonBlockingMode.NO\_TRICKLING);
        DFEArray<DFEVar> propArray = propArray\_In.data;
        DFEVar arrayCtrl           = propArray\_In.valid;
62
        // Inputs from EdgeReader
        int parentWidth = 32;
        int ptrDataWidth = parentWidth + GC.numPktsBits + 2*GC.ePtrWidth;
66      NonBlockingInput<DFEVar> edgePtrData\_In =
          io.nonBlockingInput("repPtrData", dfeRawBits(ptrDataWidth), constant.
              var(true), 1,
              DelimiterMode.FRAME\_LENGTH, 0, NonBlockingMode.NO\_TRICKLING);
        DFEVar edgePtrData = edgePtrData\_In.data;
70      DFEVar edgePtrCtrl = edgePtrData\_In.valid;
```

```
        DFEVar edgePtr0\_In = edgePtrData.slice(0, GC.ePtrWidth).cast(ePtrType);
        DFEVar edgePtr1\_In = edgePtrData.slice(GC.ePtrWidth, GC.ePtrWidth).cast
            (ePtrType);
        DFEVar edgeNumPkts =
          edgePtrData.slice(2*GC.ePtrWidth, GC.numPktsBits).cast(dfeUInt(GC.
              numPktsBits));
        // below, length should be parentWidth? check len of parentType vs
            parentWidth
        DFEVar parent\_In =
          edgePtrData.slice(2*GC.ePtrWidth+GC.numPktsBits, GC.nodeWidth).cast(
              parentType);

        // Feedback stall input from reducer
        NonBlockingInput<DFEVar> outputStall\_In =
          io.nonBlockingInput("outputStall", dfeBool(), constant.var(true), 1,
              DelimiterMode.FRAME\_LENGTH, 0, NonBlockingMode.NO\_TRICKLING);
        DFEVar outputStall = Reductions.streamHold(outputStall\_In.data,
            outputStall\_In.valid,
                    dfeBool().encodeConstant(false));

        // Queue Reader SM
        // for each elemIdx, tracks how many packets need to be consumed before
        // incrementing the RAM read pointer
        SMIO ctrlSM = addStateMachine("PropReaderCtrl",
              new QRdrPktCntSM(this, GC.numPktsBits, GC.edgeIdxRamDepth));
        ctrlSM.connectInput("pktVal",     arrayCtrl);
        ctrlSM.connectInput("numPkts",    edgeNumPkts);
        ctrlSM.connectInput("numPktsVal", edgePtrCtrl);
        DFEVar incRamRdPtr = ctrlSM.getOutput("incRdPtr");
        DFEVar pktCnt      = ctrlSM.getOutput("pktCnt");
        //accumulate the stall backpressure with all downstream stall signals
        //rising and falling of this signal controls when backpressure is
            propagated
        DFEVar inputStall  = ctrlSM.getOutput("stall") | outputStall;

        // FIFOs
        Count.Params ramRdPtrParams = control.count.makeParams(GC.
            ePtrRamDepthBits) //RdPtr
            .withEnable(incRamRdPtr);
        Count.Params ramWrPtrParams = control.count.makeParams(GC.
            ePtrRamDepthBits) //WrPtr
```

```
            .withEnable(edgePtrCtrl);
        DFEVar ramRdPtr = (control.count.makeCounter(ramRdPtrParams)).getCount()
            ;
106     DFEVar ramWrPtr = (control.count.makeCounter(ramWrPtrParams)).getCount()
            ;
        //parent FIFO
        Memory <DFEVar> parentRam = mem.alloc( parentType, GC.ePtrRamDepth);
        parentRam.write(ramWrPtr, parent\_In, edgePtrCtrl);
110     DFEVar parent = parentRam.read(ramRdPtr);
        //edgePtr0 FIFO
        Memory <DFEVar> edgePtr0Ram = mem.alloc( ePtrType, GC.ePtrRamDepth);
        edgePtr0Ram.write(ramWrPtr, edgePtr0\_In , edgePtrCtrl);
114     DFEVar edgePtr0 = edgePtr0Ram.read(ramRdPtr);
        //edgePtr1 FIFO
        Memory <DFEVar> edgePtr1Ram = mem.alloc( ePtrType, GC.ePtrRamDepth);
        edgePtr1Ram.write(ramWrPtr, edgePtr1\_In , edgePtrCtrl);
118     DFEVar edgePtr1 = edgePtr1Ram.read(ramRdPtr);  // edgePtr-1 (subtract
            prev kernel)

        int numBits = GC.nPBBits+GC.brstNbits > GC.ePtrWidth ?
            GC.ePtrWidth-GC.nPBBits : GC.brstNbits;
122     DFEVar edgeBrst0 = edgePtr0.slice(GC.nPBBits, numBits).cast(dfeUInt(
            numBits));
        DFEVar edgeBrst1 = edgePtr1.slice(GC.nPBBits, numBits).cast(dfeUInt(
            numBits));


126     // Generate a mask of which incoming data elements are active
        DFEArrayType<DFEVar> boolArray = new DFEArrayType<DFEVar>(dfeBool(),
            arSize);
        DFEArray<DFEVar> pktMask = boolArray.newInstance(this);

130     DFEType pktCntType = dfeUInt(GC.numPktsBits);
        int buf = GC.numPktsBits - GC.pktsPBBits;
        // cannot cast small to big
        DFEVar edgePkt0Raw = constant.var(dfeUInt(buf), 0) +  //+ is concat
134                          edgePtr0.slice(GC.ePPBits, GC.pktsPBBits);
        DFEVar edgePkt0 = edgePkt0Raw.cast(pktCntType);
        // cannot cast small to big
        DFEVar edgePkt1off = constant.var(dfeUInt(buf), 0) +  //+ is concat
138                          edgePtr1.slice(GC.ePPBits, GC.pktsPBBits);
```

```
        DFEVar edgePkt1base = (edgeBrst1 - edgeBrst0) << GC.pktsPBBits;
        DFEVar edgePkt1 = edgePkt1off.cast(pktCntType) + edgePkt1base.cast(
            pktCntType);

142     DFEVar midPkt = arrayCtrl \& (pktCnt > edgePkt0) \& (pktCnt < edgePkt1);
        DFEVar onePkt = arrayCtrl \& edgePkt0.eq(edgePkt1);
        DFEVar firstPkt = arrayCtrl \& pktCnt.eq(edgePkt0) \& ~onePkt;
        DFEVar lastPkt  = arrayCtrl \& pktCnt.eq(edgePkt1) \& ~onePkt;
146     DFEVar onlyPkt = arrayCtrl \& onePkt \& pktCnt.eq(edgePkt0);
        DFEVar elem0 = (edgePtr0.slice(0, GC.ePPBits)).cast(dfeUInt(GC.ePPBits))
            ;;
        DFEVar elem1 = (edgePtr1.slice(0, GC.ePPBits)).cast(dfeUInt(GC.ePPBits))
            ;;
        DFEVar[] firstPktVal = new DFEVar[arSize];
150     DFEVar[] lastPktVal = new DFEVar[arSize];
        DFEVar[] onlyPktVal = new DFEVar[arSize];
        DFEVar[] arrayElem = new DFEVar[arSize];
        //nullValue will depend on the reduction operation
154     DFEVar nullValue = constant.var(propType, 0.0);

        for (int i = 0; i < arSize; i++) {
          // Generate masks
158       firstPktVal[i] = (firstPkt \& (i >= elem0));
          lastPktVal[i] = (lastPkt \& (i <= elem1));
          onlyPktVal[i] = (onlyPkt \& (i >= elem0) \& (i <= elem1));
          pktMask[i] <== midPkt | firstPktVal[i] | lastPktVal[i] | onlyPktVal[i
              ];
162
          //data is in lowest 32 bits
          arrayElem[i] = pktMask[i] ? propArray[i].slice(0, propWidth).cast(
              propType)
                                    : nullValue;
166       //degree is in highest 32 bits
          //deg[i] = propArray[i].slice(propWidth, 16).cast(dfeUInt(16));
        }

170     DFEVar pktsThisCycle = pktMask.pack().cast(dfeUInt(arSize));
        DFEVar thisPkt = pktsThisCycle > 0;

        //reduction result:
174     DFEVar redResult =
```

```
        (( (arrayElem[15]+arrayElem[14])+ (arrayElem[13]+arrayElem[12]))+
        (  (arrayElem[11]+arrayElem[10])+ (arrayElem[ 9]+arrayElem[ 8])) )+
        ( ((arrayElem[ 7]+arrayElem[ 6])+ (arrayElem[ 5]+arrayElem[ 4]))+
178     (  (arrayElem[ 3]+arrayElem[ 2])+ (arrayElem[ 1]+arrayElem[ 0])) );
        // clear the accumulated result the cycle after all the data
        DFEVar clrResult = stream.offset(incRamRdPtr, -1);
        // storage registers for reduction results: if large number of neighbors
             are
182     // possible, then add reduction registers as necessary
        DFEVar redReg0 = Reductions.streamHold(redResult, clrResult|pktCnt.eq(0)
            ,
                propType.encodeConstant(0.0));
        DFEVar redReg1 = Reductions.streamHold(redResult, clrResult|pktCnt.eq(1)
            ,
186             propType.encodeConstant(0.0));
        DFEVar redReg2 = Reductions.streamHold(redResult, clrResult|pktCnt.eq(2)
            ,
                propType.encodeConstant(0.0));
        DFEVar redReg3 = Reductions.streamHold(redResult, clrResult|pktCnt.eq(3)
            ,
190             propType.encodeConstant(0.0));
        DFEVar curResult = reduceOp(reduceOp(redReg0, redReg1),
            reduceOp(redReg2, redReg3));

194     /*
        // - Loop-based approach to accumulating the reduction result is tricky
        //   because of the long latency of the floating point add
        // - Using N registers limits the number of neighbors that can be
            reduced
198     OffsetExpr resultLoopLen = stream.makeOffsetAutoLoop("resultLoopLen");
        DFEVar accResult = propType.newInstance(this);
          DFEVar tmpResult = clrResult ? nullValue :
             reduceOp(accResult, redResult);
202     DFEVar curResult = Reductions.streamHold(tmpResult, arrayCtrl|
            clearResult,
                 propType.encodeConstant(0.0));
        accResult <== stream.offset(curResult, -resultLoopLen);
        */
206

        // Determine when the kernel has finished processing for this curLvl
```

```
        DFEVar edgePtrCtrl\_d = stream.offset(edgePtrCtrl, -1); //avoid blip
            with first elem
210     DFEVar begun = Reductions.streamHold(edgePtrCtrl\_d, edgePtrCtrl\_d,
             dfeBool().encodeConstant(false));
        DFEVar emptyRam = ramRdPtr.eq(ramWrPtr);
        DFEVar emptyReqs = constant.var(true);
214     DFEVar knlDone = begun \& emptyRam \& emptyReqs;
        DFEVar knlDonePrev = stream.offset(knlDone, -1);

        //DFEVar sendDone = knlDone \& ~knlDonePrev;
218     DFEVar doneVal  = knlDone \&  knlDonePrev;
        Count.Params sendDoneParams = control.count.makeParams(32)
          .withEnable(doneVal);
        DFEVar sendDoneCnt = (control.count.makeCounter(sendDoneParams)).
            getCount();
222     DFEVar sendLength = uDVal*2;
        DFEVar sendDone = knlDone \& (sendDoneCnt < sendLength);

        // Output to ElemUpdate: Emit parent and reduction result
226     DFEVar finalResult = prTerm + (d*curResult);
        DFEType parResType = dfeRawBits(GC.nodeWidth+propWidth);
        DFEVar parentResultRaw = parent + finalResult;  //+ is concat
        io.output("parentResult", parentResultRaw, parResType, incRamRdPtr);
230
        // Outputs to BVN\_Writer
        io.output("nbrRedIdle", knlDone, dfeBool(), sendDone);

234     // Stall feedback:  flow control to previous unit
        DFEVar inputStallD = stream.offset(inputStall, -1);
        DFEVar startStall  = inputStall \& ~inputStallD;
        DFEVar stopStall   = ~inputStall \& inputStallD;
238     io.output("stall", inputStall, dfeBool(), startStall|stopStall);

        flush.onTrigger(constant.var(false));
      }
```

ElemUpdate

The code below illustrates ElemUpdate.

Listing A.3: MaxJ code for the ElemUpdate block.

```
package pagerank.kernels;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.SMIO;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEArray;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.
    DFEArrayType;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.IO.*;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Reductions;
import pagerank.GC;

public class ElemUpdate extends Kernel {

  public ElemUpdate(KernelParameters parameters) {
    super(parameters);
    DFEVar cnt = control.count.simpleCounter(64);
    // Scalar Inputs
    DFEVar propAddr  = io.scalarInput("propAddr", dfeUInt(GC.scalWidth));
    DFEVar uDVal     = io.scalarInput("uDVal"   , dfeUInt(32));

    // Types
    int arSize = GC.PKTWIDTH/GC.PROPWIDTH;
    DFEType propType48 = dfeRawBits(GC.PROPWIDTH);
    DFEType propType   = dfeFloat(8,24);
    int propWidth      = 32;
    DFEType parentType = dfeUInt(GC.nodeWidth);
    DFEType nbrType    = dfeUInt(GC.edgeWidth);
    DFEArrayType<DFEVar> propArrayType =
      new DFEArrayType<DFEVar>(propType48, arSize);

```

```
        // Inputs from DRAM
        NonBlockingInput<DFEArray<DFEVar>> propArray\_In =
          io.nonBlockingInput("propArray", propArrayType, constant.var(true), 1,
39                    DelimiterMode.FRAME\_LENGTH, 0, NonBlockingMode.NO\
                          _TRICKLING);
        DFEArray<DFEVar> propArray = propArray\_In.data;
        DFEVar propArrayCtrl       = propArray\_In.valid;

43      // Inputs from LvlReader
        DFEType edgeType = dfeUInt(GC.EDGEWIDTH);
        int parResBits   = GC.nodeWidth+propWidth;
        DFEType parResType = dfeRawBits(parResBits);
47      NonBlockingInput<DFEVar> parentRes\_Input =
          io.nonBlockingInput("parentResult", parResType, constant.var(true), 1,
                    DelimiterMode.FRAME\_LENGTH, 0, NonBlockingMode.NO\
                          _TRICKLING);
        DFEVar parentRes\_In  = parentRes\_Input.data;
51      DFEVar parentResCtrl = parentRes\_Input.valid;
        DFEVar result\_InRaw  = parentRes\_In.slice(0, propWidth);
        DFEVar result\_In     = result\_InRaw.cast(propType);
        DFEVar parent\_InRaw  = parentRes\_In.slice(propWidth, GC.nodeWidth);
55      DFEVar parent\_In     = parent\_InRaw.cast(parentType);

        Count.Params pktCntParams = control.count.makeParams(GC.pktsPBBits)
            .withEnable(propArrayCtrl);
59      Counter pktCounter = control.count.makeCounter(pktCntParams);
        DFEVar pktCnt = pktCounter.getCount();

        /////////////////
63      // UpdQueueSM
        /////////////////
            // can be a small depth. streams through,
            // doesn't stay in fifo. there is also a stall
67          // signal.
        SMIO uqSM = addStateMachine("UpdQueueSM",
            new UpdQueueSM(this, GC.nbrNumRamDepth));
        uqSM.connectInput("nodeResIn", parentRes\_In.cast(dfeUInt(parResBits)));
71      uqSM.connectInput("nodeResInCtrl", parentResCtrl);
        uqSM.connectInput("arrayCtrl", propArrayCtrl);
        DFEVar brstToWrite  = uqSM.getOutput("reqBrstAddrWr");
        DFEVar req\_en\_wr    = uqSM.getOutput("reqEnWr");
```

```
75      DFEVar brstToRead   = uqSM.getOutput("reqBrstAddrRd");
        DFEVar req\_en\_rd    = uqSM.getOutput("reqEnRd");
        DFEVar emptyQ       = uqSM.getOutput("emptyQ");
        DFEVar addrToUpdate = uqSM.getOutput("addrToUpdate");
79      DFEVar bitVecUpd    = uqSM.getOutput("resultsBitVec");
        DFEVar resPkt       = uqSM.getOutput("resultsPkt");
        DFEVar resPktVal    = uqSM.getOutput("resultsPktVal");
        DFEVar inputStall   = uqSM.getOutput("stall");
83
        // currently restricted to 2 pkts per burst (not 4 etc)
        DFEVar arBitVec = pktCnt.eq(0) ? bitVecUpd.slice(0, arSize)
               : bitVecUpd.slice(arSize, arSize);
87      DFEArray<DFEVar> propArrayOut = propArrayType.newInstance(this);

        for (int i = 0; i < arSize; i++) {
          DFEVar thisRes   = resPkt.slice(i*propWidth, propWidth);
91        DFEVar newValRaw = constant.var(dfeRawBits(16),0) + thisRes; //+ is
              concat
          propArrayOut[i] <== arBitVec.slice(i) ? newValRaw : propArray[i];
        }

95      // End of Execution
        // Determine when the kernel has finished processing for this curLvl
        // For this kernel, there are 2 cases that require done packets:
        //   a. Send some done packets before any activity duing this iteration.
99      //      This is because the last iteration will not have any data (no
        //      modifications of graph properties).
        //      - Be sure to send a packet if this iteration is not empty (upon
        //      parentResCtrl)
103     //   b. Send some done packets after this iteration's activity to tell
            the
        //      receiver (BVNWriter) that this iteration has completed.
        //
        DFEVar begun = Reductions.streamHold(parentResCtrl, parentResCtrl,
107          dfeBool().encodeConstant(false));

        DFEVar noActiveData = ~propArrayCtrl \& ~parentResCtrl;
        DFEVar knlDoneLast  = ~begun;  // case a: no data during last iteration
111     DFEVar knlDoneData  = begun \& emptyQ \& noActiveData; // case b
        DFEVar knlDone      = knlDoneData | knlDoneLast;
        DFEVar knlDone\_d    = stream.offset(knlDone, -1);
```

```
        DFEVar doneOff      = (knlDone\_d \& parentResCtrl);
115
        // Counter limits the number of done packets sent.  Too many packets
        // increases buffering and latency.
        Count.Params sendDoneParams = control.count.makeParams(32)
119       .withEnable(knlDone)
          .withReset(doneOff);
        DFEVar sendDoneCnt = (control.count.makeCounter(sendDoneParams)).
            getCount();
        DFEVar sendLength = uDVal*2;
123     // doneOff signal notifies the receiver that this iteration is not empty
        DFEVar sendDone = (knlDone \& (sendDoneCnt < sendLength)) | doneOff;
        io.output("lvlUpdIdle", knlDone, dfeBool(), sendDone);

127     // Output modified propArrayA to DRAM
        DFEVar propArrayCtrlOut = resPktVal;
        io.output("propArrayMod", propArrayOut, propArrayType, propArrayCtrlOut)
            ;

131     // Memory Requests
        DFEVar req\_size     = constant.var(dfeUInt(GC.sizeBits), 1);
        DFEType reqAddrType = dfeUInt(GC.brstNbits);
        DFEVar propAddrBase = propAddr.cast(reqAddrType);
135     DFEVar req\_brst\_addr\_rd = propAddrBase + brstToRead;
        io.output("memReqRd", req\_size\
            req_brst_addr_rd#req_en_rd,dfeRawBits(GC.memReqWidth));// Write request: updated
            prop arrayDFEVar req_brst_addr_wr = propAddrBase +
            brstToWrite;io.output("memReqWr",
            req_size#req_brst_addr_wr#req_en_wr,dfeRawBits(GC.memReqWidth));// Stall feedback:
            flow control to previous unitDFEVar inputStallD = stream.offset(inputStall,
            -1);DFEVar startStall = inputStall &  inputStallD;DFEVar stopStall =  inputStall &
            inputStallD;io.output("stall", inputStall, dfeBool(),
            startStall|stopStall);flush.onTrigger(constant.var(false));
```

Bibliography

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[2] Andrey Balmin and Yannis Papakonstantinou. Storing and querying xml data using denormalized relational databases. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 14(1):30–49, 2005.

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[4] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[5] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[6] Brahim Betkaoui, David B Thomas, Wayne Luk, and Natasa Przulj. A framework for fpga acceleration of large graph problems: graphlet counting case study. In *FPT 2011*, pages 1–8. IEEE, 2011.

[7] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Signal Processing, IEEE Transactions on*, 49(10):2408–2421, 2001.

[8] Norman Biggs. *Algebraic graph theory*. Cambridge university press, 1993.

[9] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

[10] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[11] Jakob Carlström and Thomas Bodén. Synchronous dataflow architecture for network processors. *Micro, IEEE*, 24(5):10–18, 2004.

[12] Reinhardt Che, Beckmann. Belred: Constructing gpgpu graph applications with software building blocks.

[13] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS 2012, IEEE 26th International*, pages 378–389. IEEE, 2012.

[14] Seonil Choi, Ronald Scrofano, Viktor K Prasanna, and Ju-Wook Jang. Energy-efficient signal processing using fpgas. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 225–234. ACM, 2003.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[17] Altera Corp. Overview - Altera FPGAs. `https://www.altera.com/products/fpga/overview.html`.

[18] NVIDIA Corp. Tesla product literature. `http://www.nvidia.com/object/tesla_product_literature.html`.

[19] Xilinx Corp. Power Analysis Overview. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/ise_c_power_analysis_overview.htm`.

[20] D Crookes, K Benkrid, A Bouridane, K Alotaibi, and A Benkrid. Design and implementation of a high level programming environment for fpga-based image processing. *IEE Proceedings-Vision, Image and Signal Processing*, 147(4):377–384, 2000.

[21] Michael DeLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E Uribe, Thomas F Knight Jr, and Andre DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 143–151. IEEE, 2006.

[22] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[23] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.

[24] Jack Bonnell Dennis. Data flow supercomputers. *Computer*, (11):48–56, 1980.

[25] David Ediger, Karl Jiang, Jason Riedy, and David A Bader. Massive streaming data analytics: A case study with clustering coefficients. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[26] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[27] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 345–354. ACM, 2012.

[28] James Goodman and Anantha P Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *Solid-State Circuits, IEEE Journal of*, 36(11):1808–1820, 2001.

[29] R. Grimes, D. Kincaid, and D. Young. Itpack 2.0 user's guide. Technical report, University of Texas Technical Report CNA-150, Center for Numerical Analysis, 1979.

[30] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2):302–312, 2004.

[31] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, volume 94, pages 12–15. Citeseer, 1994.

[32] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, volume 4873. Springer, 2007.

[33] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[34] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.

[35] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *PACT 2011*, pages 78–88. IEEE, 2011.

[36] Sungpack et al. Hong. Green marl: Domain-specific language for graph analytics. In *PPoPP 2012*.

[37] Xilinx Inc. All Programmable FPGAs - Xilinx. `http://www.xilinx.com/products/silicon-devices/fpga.html`.

[38] Ju-wook Jang, Seonil Choi, and Viktor K Prasanna. Energy-efficient matrix multiplication on fpgas. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 534–544. Springer, 2002.

[39] David B Kirk, Matthew Papakipos, Shaun Ho, Walter Donovan, and Curtis Priem. Graphics pipeline including combiner stages, December 25 2001. US Patent 6,333,744.

[40] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2014.

[41] Kamesh Madduri, David Ediger, Karl Jiang, David Bader, Daniel Chavarria-Miranda, et al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[42] Grzegorz Malewicz, Matthew H Austern, Aart JC 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*, pages 135–146. ACM, 2010.

[43] Michael C McFarland, Alice C Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[44] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[45] Onur Mutlu. Main memory scaling: Challenges and solution directions. In *More than Moore Technologies for Next Generation Computer Design*, pages 127–153. Springer, 2015.

[46] Sofia Maria Nikolakaki. *Real-time Stream Data Processing with FPGA-Based SuperComputer*. PhD thesis, Technical University of Crete, Greece, 2015.

[47] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28. IEEE, 2014.

[48] Tayo Oguntebi. Graphops source repository. `https://github.com/tayo/GraphOps`.

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[50] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.

[51] Oliver Pell, Oskar Mencer, Kuen Hung Tsoi, and Wayne Luk. Maximum performance computing with dataflow engines. In *High-Performance Computing Using FPGAs*, pages 747–774. Springer, 2013.

[52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[53] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[54] G Lawrence Sanders and Seungkyoon Shin. Denormalization effects on performance of rdbms. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2001.

[55] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 14. IEEE Computer Society Press, 2012.

[56] Robert Schreiber, Shail Aditya, B Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of nonprogrammable hardware accelerators. In *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, pages 113–124. IEEE, 2000.

[57] Maxeler Technologies. Maxcompiler documentation. `http://www.maxeler.com`.

[58] Neo Technologies. Graph Database vs Relational Database. `http://neo4j.com/developer/graph-db-vs-rdbms/`.

[59] Hoe Weisz, Nurvitadhi. Graphgen for coram: Graph computation on fpgas. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2013.

[60] Jay J Wylie, Michael W Bigrigg, John D Strunk, Gregory R Ganger, Han Kiliççöte, and Pradeep K Khosla. Survivable information storage systems. *Computer*, 33(8):61–68, 2000.

[61] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.