GraphOps: A Dataflow Library for Graph Analytics Acceleration

Tayo Oguntebi
Pervasive Parallelism Laboratory
Stanford University
tayo@stanford.edu

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University
kunle@stanford.edu

ABSTRACT

Analytics and knowledge extraction on graph data structures have become areas of great interest. For frequently executed algorithms, dedicated hardware accelerators are an energy-efficient avenue to high performance. Unfortunately, they are notoriously labor-intensive to design and verify while meeting stringent time-to-market goals.

In this paper, we present GraphOps, a modular hardware library for quickly and easily constructing energy-efficient accelerators for graph analytics algorithms. GraphOps provide a hardware designer with a set of composable graph-specific building blocks, broad enough to target a wide array of graph analytics algorithms. The system is built upon a dataflow execution platform and targets FPGAs, allowing a vendor to use the same hardware to accelerate different types of analytics computation. Low-level 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. As an enabling contribution, we also present a novel locality-optimized graph data structure that improves spatial locality and memory efficiency when accessing the graph in main memory.

Using the GraphOps system, we construct six different hardware accelerators. Results show that the GraphOps-based accelerators are able to operate close to the bandwidth limit of the hardware platform, the limiting constraint in graph analytics computation.

Keywords

FPGA, Graph analysis, Analytics, Dataflow, Accelerator

1. INTRODUCTION

Graph analytics problems have recently attracted significant interest from the research and commercial communities. A large number of important data sets can be usefully expressed as graphs, which efficiently encode connections between data elements. Analytics algorithms executed on these data sets can yield valuable insight otherwise difficult to extract from traditional data stores, e.g. relational databases.

The ability of graphs to capture information about relationships makes them easily amenable to a wide variety of data analytics algorithms. The usefulness of these algorithms has been amplified by the prevalence of large data sets, now becoming commonplace in data centers operated by large corporations and research labs. As graphs become a more integral tool in processing unstructured network-based data, the energy efficiency of the operations executed on them begins to assume prime importance.

It is commonly known that the vast majority of large scale efforts in data analytics utilize commodity hardware, for a variety of reasons [6, 9]. Dedicated hardware accelerators usually provide significant performance-per-energy benefits, but are considered unwieldy and difficult to program [10].

In this context, we present *GraphOps*, a hardware library for quickly and easily constructing energy-efficient accelerators for graph analytics algorithms. GraphOps provide a hardware designer with a set of composable graph-specific building blocks necessary to implement a wide array of graph analytics algorithms. The target user is a hardware designer who can design her own logic but would benefit from a library of modular building blocks that are tailored to the domain of graph analytics. The system is built upon a *dataflow* execution platform [17]. GraphOps-based accelerators are defined as sets of *blocks* in which graph data are streamed to/from memory and computation metadata are streamed through the GraphOps blocks as inputs and outputs.

Most graph data structures are inherently sparse – indeed, they are usually represented on disk as sparse matrices. This sparseness manifests itself as a dearth of spatial locality when attempting to traverse a vertex's edges to visit its neighbors. We address this issue by using a modified storage representation to increase spatial locality and enable element-wise parallelism within the GraphOps blocks. GraphOps-based accelerators pre-process the graph on the host machine and store it in the FPGA memory space using our modified storage format. Stubborn hardware implementation details such as flow control, input buffering, rate throttling, and host/interrupt interaction are built into the design of the GraphOps, greatly reducing design time.

The major contributions of this paper are:

- We present a library of flexible, modular hardware modules that can be used to construct accelerators for high-performance streaming graph analytics. We enumerate a subset of the components in Section 5.

FPGA'16, February 21 - 23, 2016, Monterey, CA, USA

ISBN 978-1-4503-3856-1/16/02...$15.00

DOI: http://dx.doi.org/10.1145/2847263.2847337

- We describe a novel graph representation that is optimized for coalesced access to the properties of the graph elements. This data structure is used in all accelerators and is described in Section 3.

- We illustrate how the GraphOps library can be used to construct new accelerators. As a case study, we construct a PageRank accelerator in Section 4.

- We prototype the hardware library on an FPGA platform by implementing six different hardware accelerators. We present evaluation results of the hardware by comparing against two different types of software systems in Section 6.

2. BACKGROUND

In this section, we provide the terms used in this paper as well as give a short overview of key technologies used in this work.

2.1 Graph Terminology

Research on graphs is extensive, spanning from mathematics to computational science and beyond, and standard terminology is not well established. Our terms are briefly described here.

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. Properties can be arbitrary data members of any type.

2.2 Dataflow Architectures

Dataflow architectures are a special type of computer architecture in which there is no traditional program counter. Instead, the execution of processing is determined by the flow and availability of data inputs to instructions which are manifested as kernel processors. Data flows from memory to the kernels and also flows between the kernel processors. The 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, network routing, and graphics processing.

3. LOCALITY-OPTIMIZED GRAPH REPRESENTATION

It has long been known that computational performance of graph analytics codes is usually bound by memory bandwidth [5]. The memory constraint is compounded by a dearth of spatial locality due to the inherently sparse nature of graph data structures.

Unfortunately, the memory controllers of FPGAs and other accelerator systems are often optimized for throughput – wide memory channels with coarse fetch granularities. Because the GraphOps library relies heavily on data parallelism, naive executions of GraphOps on standard graph data structures suffer heavily from this memory bottleneck. To address this issue when using the FPGA memory system, we propose a novel *locality-optimized graph representation* that uses redundancy to trade off compactness for locality.

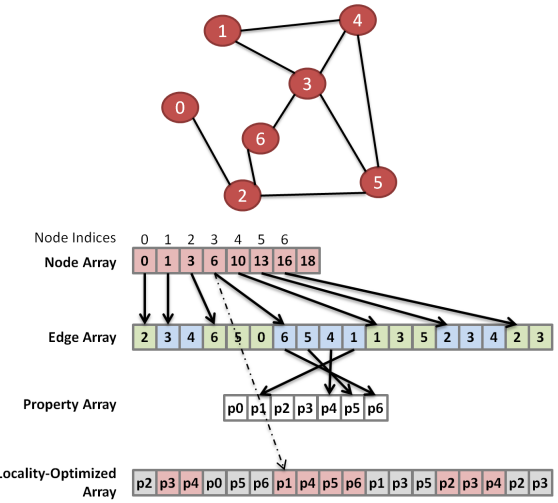

Figure 1: A simple graph and its associated data structures. The locality-optimized array redundantly coalesces properties, making the accessing of a vertex's neighbors' properties be possible with only level of indirection. In particular, the expensive second (random) level of indirection is avoided.

3.1 Traditional Graph Representation

There is abundant prior research on data structures for representing graphs in computer memory [4, 5]. 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. Common formats include compressed sparse row (CSR), coordinates list (COO), and ELLPACK (ELL).

The compressed sparse row format is composed of a vertex array (or node array) and an edge array. These arrays are shown, along with their associated graph, in Figure 1. The vertex array is indexed by the vertex ID (an integer). Data elements in the vertex array act as indices into the edge array, which stores the destination vertex of each edge.[1]

3.2 Locality-Optimized Graph Representation

The edge array is an efficient data structure for reading lists of neighbors. However, when accessing associated properties for a neighbor set, the system must perform random access into a property array. This scattering effect is visualized by the accesses to the property array in Figure 1. 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.

We propose adding a new data structure called the locality-optimized array, also shown in the figure. Instead of an edge list, this array stores the properties associated with those neighbors. Reading neighbor properties now is achieved without random access, significantly increasing spatial locality. Updates are still performed to the original property array.

The new data structure achieves locality by replicating

[1]Compressed sparse row (CSR) also features heavily in the domain of sparse matrix computation [1].

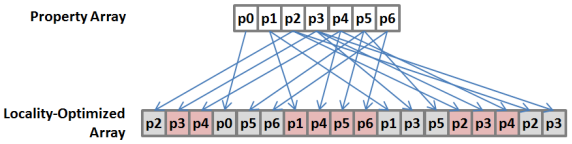

Figure 2: The locality-optimized array is generated and updated offline on the host.

properties of vertices which serve as neighbors to multiple other vertices. This array needs to be prepared and updated offline. Figure 2 displays the scattering maintenance operation that is performed to prepare the array. This operation is performed on the host machine in-between computation iterations.

4. A MOTIVATING EXAMPLE

Before going into the details of the GraphOps blocks and their architectures, let us first motivate the library and explain how these components are used to compose a practical application. We will focus on a well-known algorithm, *PageRank* [15]. We assume that the accelerator is being run in tandem with the main application on the host system. We present the construction of the application in three categories: block selection, block parameterization, and block composition.

Block Selection

We refer the reader to the reference [15] for a detailed description of the algorithm. Through profiling or code analysis, the designer would determine that the calculation of new pagerank scores for each vertex dominates the runtime of this algorithm. This is the core computation that a GraphOps-based accelerator most naturally accelerates. For every vertex, a reduction is performed using the neighbors of that vertex. Fundamentally, a PageRank accelerator needs to perform three functions:

1. Fetch the necessary sets of data properties (PageRank scores of neighboring vertices, in this case) for each vertex.
2. Perform the arithmetic reduction operation described by the core algorithm.
3. Store the updated values by updating the data structure after each iteration.

The components in the GraphOps library were borne out of necessity, as the authors implemented several prominent graph analytics algorithms within the dataflow model. Patterns emerged across these applications, and we encoded these patterns in the library as GraphOps blocks. The final set of GraphOps blocks result in: (i) natural coverage of a wide, interesting set of graph analytics algorithms and (ii) ease of use when compared with other flows, e.g. HDL and HLS.

For PageRank, the high-level blocks used are *ForAllPropRdr*, *NbrPropRed*, and *ElemUpdate*. The next section goes into more detail about these blocks. The full library has been open-sourced, and further details about the blocks are available online [14].

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. 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 6.

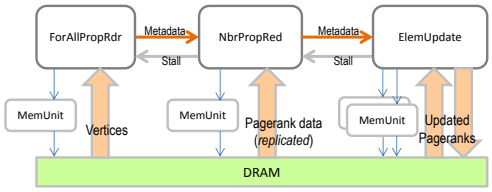

Figure 3: Composition of GraphOps blocks to form the PageRank accelerator.

Every block requires a few local parameters, needed for customization. There are also global accelerator-level parameters that can be modified. One common parameter that is used by several different blocks is that of memory addresses for property arrays. Any block that issues a memory request must have this base address from which to determine the memory location of specific graph elements. Further details regarding parameterization are available in the library documentation at the source repository [14].

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 memory interface units. Figure 3 shows a detailed block diagram of the blocks used in the PageRank algorithm.

5. HARDWARE DESIGN

This section continues the PageRank example of the previous section by presenting the design details of the GraphOps library components used in PageRank. We begin with brief descriptions of each of the library components. Because of space constraints, we will not fully characterize all of the details and parameters involved in all components. We will instead focus on one interesting component in detail, describing its architecture, internal structures, and operation. Using the components presented, we explore other aspects of the hardware design of GraphOps. We finish by providing some of the graph-specific optimizations used to maximize throughput.

We have open-sourced the entire GraphOps dataflow library under the MIT License as a Github repository [14]. The repository documentation contains additional documentation that describes each of the blocks, parameterization suggestions, and composition instructions.

5.1 Enumeration of PageRank GraphOps

The GraphOps library can be broken down into three categories: *data* blocks, *control* blocks, and *utility* blocks. We enumerate blocks of each category used in PageRank.

5.1.1 Data-Handling Blocks

Data blocks are the primary GraphOps components. They handle incoming data streams, perform arithmetic operations, and route outputs to memory or subsequent blocks:

(i) **ForAllPropRdr** issues memory requests for all neigh-

bor property sets in the graph. In order to do this, it first reads all the row pointers in the graph. The incoming row pointer data are used to issue individual memory requests for each set of neighbor properties.

(ii) **NbrPropRed** performs a reduction on a vertex's neighbor set. The unit receives the neighbor property data as a data stream 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).

(iii) **ElemUpdate** is used to update property values in the graph data structure. The unit receives a vertex reference and an updated value as input. It issues memory read requests for the requisite memory locations and memory update requests for the updated values.

5.1.2 *Control Blocks*

In GraphOps, the majority of the logic is amenable to dataflow. One key reason for this is that feedback control is rare. There are situations, however, that call for more intricate control difficult to express without state machines. Key control blocks for the PageRank application are:

(i) **QRdrPktCntSM** handles control logic for input buffers in the data blocks. A common use case occurs in the following situation: A metadata input datum dictates how many memory packets belong to a given neighbor set. QRdrPktCntSM handles the counting of packets on the memory data input and instructs the data block when to move on to the next neighbor set.

(ii) **UpdQueueSM** handles control logic for updating a graph property for all nodes. This unit assumes that the properties are being updated sequentially and makes use of heavy coalescing to minimize the number of update requests sent to memory.

5.1.3 *Utility Blocks*

Additional logic is needed to properly interface with the memory system and the host platform. These are realized via the utility blocks:

(i) **EndSignal** monitors *done* signals for all data blocks and issues a special interrupt request to halt execution when all units are finished.

(ii) **MemUnit** provides a simplified memory interface to the data blocks. It compiles memory profiling information, watches for end-of-execution interrupt requests, and includes control logic for handling very large memory requests.

5.2 NbrPropRed

We now specify the details of one representative block to give the reader an illustration of the type of logic common in data blocks. NbrPropRed is interesting because it interacts with memory streams while performing computation that is fundamental to an algorithm.

Figure 4 shows a detailed diagram of the NbrPropRed block. As described in Section 5.1, this block is used to perform a reduction on the properties of a vertex's neighbor set, a common operation for many analytics algorithms. In the dataflow paradigm, this block would be placed after a

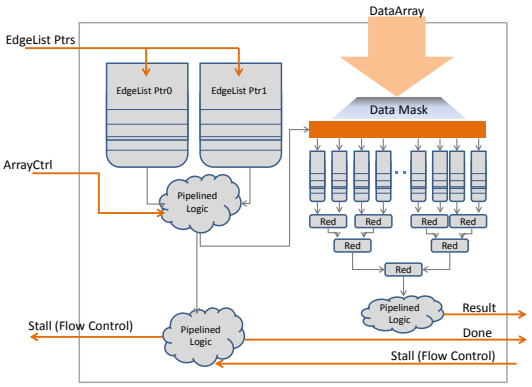

Figure 4: Detailed architecture of the NbrPropRed data block.

block which has already issued memory requests for neighbor properties.

The NbrPropRed has three primary inputs: the property data stream from memory (*DataArray* and *ArrayCtrl*), metadata from the previous requesting block (*EdgeList Ptrs*), and a flow control signal to halt execution when buffers are nearing capacity (*Stall*). There are three primary outputs: the metadata output for this block is the reduction results (*Result*). The other two outputs are common to all data blocks. The *Stall* output is issued when this block has halted, telling upstream blocks to also halt. The *Done* signal is asserted when this block has finished its work.

For each memory request, the corresponding edge list pointers define a data mask that dictates which properties in the data stream are a part of the neighbor set. Simple logic operations are used to generate this mask. The data mask filters the incoming data stream and allows only valid neighbor properties to participate in the reduction tree. For very large neighbor sets, many data packets are required to be reduced. The reduction tree therefore uses accumulation registers to handle these large sets. When the neighbor set is finished processing (as dictated by the edge list pointers), the accumulation registers are cleared and the result is emitted by the block (*Result* output).

6. EVALUATION

We prototyped the GraphOps architecture using a system from Maxeler Technologies. The system 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 of the relatively large width (384 bits) of the memory channel.

The FPGA is connected via PCIe x8 to the host processor system. The host system has two 2.67 GHz Xeon 5650 multi-processors, each having six multi-threaded cores mak-

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

Algorithm	Resource Usage (%)			GraphOps Blocks	
	FF	LUT	BRAM[2]	# MemUnits	Blocks Used
pagerank [15]	33.2	21.3	24.5	5	ForAllPropRed, NbrPropRed, ElemUpdate
bfs	32.1	18.8	36.6	6	NbrPropRdr, NbrPropFilter, ElemUpdate, SetWriter, SetReader
conduct [3]	25.7	16.0	20.6	4	AllNodePropRed, NbrPropRdr, NbrPropRed
spmv	33.0	20.6	24.5	5	ForAllPropRed, NbrPropRed, ElemUpdate
sssp	30.7	18.8	37.0	6	NbrPropRdr, NbrPropFilter, ElemUpdate, SetWriter, SetReader
vcover [16]	23.4	14.7	19.4	3	ForAllPropRed, GlobNbrRed

Table 1: Resource usage and enumeration of GraphOps blocks for each accelerator. The EndSignal block, used in all accelerators, is not included.

ing 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 a software framework built by Maxeler [17]. The tools provide an HDL and interfaces for accelerating development of dataflow and streaming accelerators. The higher level language is compiled to generate VHDL.

6.1 Applications

We use the GraphOps blocks to implement accelerators for six analytics algorithms. Table 1 breaks down the resource usage of each accelerator and lists the GraphOps components used in their implementations. None of the accelerators are bound by on-chip resources on our Xilinx Virtex-6 FPGA. However, BRAM resources can impose undue pressure on the place-and-route tool if the designer is too liberal with use of FIFO buffers. The throttling and flow control schemes described in Section 5.2 naturally reduce buffer pressure and allow for more conservative sizing. Typical buffer sizes in the GraphOps blocks range from about 2K to 8K elements, with typical element widths being about 32 to 64 bits.

6.2 Software Comparison

We present the performance of the GraphOps-based accelerators by comparing computation done on the accelerators with computation done via an optimized software implementation. The software implementations are C++ multi-threaded (OpenMP) versions generated using the Green-Marl graph compilation framework [11]. Software versions are run using the Intel Xeon processors on the host system, described at the beginning of this section. The GraphOps accelerated versions use the run-times of the same applications in a C program with the "inner loop" computation accelerated using the FPGA, as described for PageRank in Section 4. The time to transfer the graph data to/from the host and the FPGA is not included.

Figure 5 compares performance throughput for selected accelerators. The x-axis is the number of vertices in the graph. The degree of the vertices of the graph follows a uniform distribution with the average being eight. This means that the number of edges in each workload is 8*N. The y-axis is millions of edges per second, or MEPS, 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.

Referring to the legend, SW1 through SW8 are the number of threads used by each software version. GraphOps (150 MHz) is the standard FPGA-accelerated version. HW+Scatter is the accelerated version which also takes into account the time to pre-process the graph data structure on the host system.

All of the graph analytics algorithms displayed are bound by memory bandwidth. This is evident in Figure 5, as all of the lines begin to roughly approach a steady state 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. Note that both of these effects apply partially to the HW+Scatter version, because the graph pre-processing workload is a normal software function and therefore depends on the cache.

In contrast to the software versions, the GraphOps-based implementations show no caching effects, as there is no significant re-use happening on the FPGA. Therefore, we see that the throughput is roughly constant for all three accelerators. Comparing the performance of the software and hardware versions, we see that the SpMV and Vertex Cover eight-threaded software versions perform better than the FPGA implementation, even for the largest graph size. As stated earlier, the FPGA peak bandwidth is about 38 GB/s while the CPU bandwidth is about 32 GB/s per socket. The eight threads are co-located on the same socket. The reason for the superior software performance is its access to three memory channels as compared with one channel on the FPGA. Using calculations based on the number of data words requested during FPGA execution runs, we determine that the single memory channel causes some memory request queues (and therefore the entire streaming system) to occasionally stall. An ideal prototyping system for GraphOps-based accelerators would have several memory channels, with the most heavily-used memory interfaces having access to a dedicated memory channel.

6.3 Streaming Comparison

We continue the evaluation of the GraphOps library by comparing against a graph processing framework called X-Stream [18]. The X-Stream framework is an apt choice because, similarly to the locality-optimized storage representation used in GraphOps (Section 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. We refer the reader to the X-Stream reference [18] for a more thorough description of the X-Stream system.

Our locality-optimized storage representation is similar to

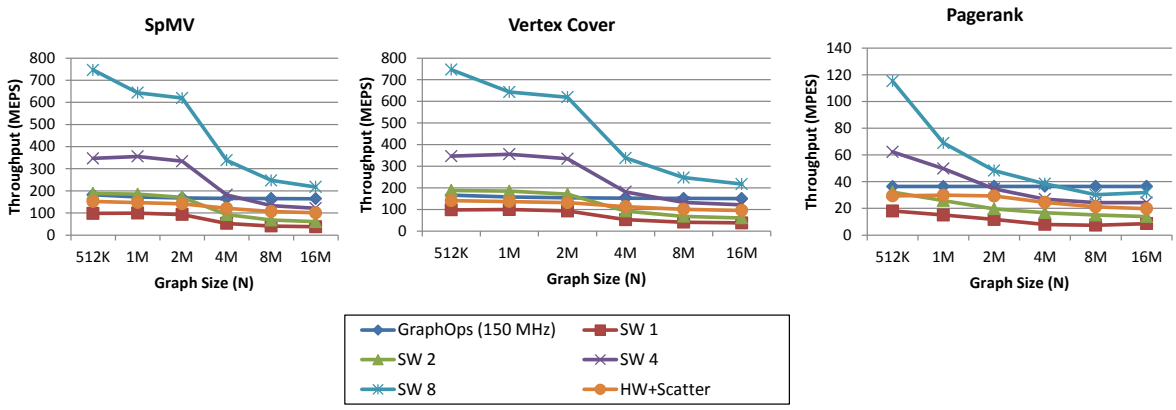

Figure 5: Performance throughput for selected accelerators.

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 from the Stanford SNAP project [12] for use in this comparison study. The data sets used are: *amazon0601*, *cit-Patents*, *wiki-Talk*, *web-BerkStan*, and *soc-Pokec*. We refer the reader to the reference for detailed characteristics about the data sets.

Figure 6 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 the beginning of this section. 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 runtime 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 that memory accesses are constrained to rather large data blocks. Indeed, we designed the locality-optimizing storage representation presented in Section 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.

6.4 Bandwidth Utilization Calculations

We performed an experiment to determine how well the single memory interface was being utilized by the various memory interfaces. Details are omitted for brevity. The steady state throughput for the PageRank 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 memory channel must switch among the three other interfaces also issuing requests–this is the primary cause for performance limitation. The secondary cause is that issuing one memory request per neighbor set limits the size of the requests and prevents optimizations at the memory controller level.

7. RELATED WORK

Several approaches have used FPGAs as the vehicle for accelerating graph analytics. Betkaoui et al [2] accelerated the graphlet counting algorithm on an FPGA using an optimized crossbar and custom memory banks. Their work differs from GraphOps, because their framework requires an end user to express his algorithm as a vertex-centric kernel, similar to Pregel [9]. They do the work of mapping it to a Convey hardware system. Nurvitadhi, Weisz, et al [13] created an FPGA backend for a graph algorithm compiler called GraphGen. They also focus only on vertex-centric graph descriptions, an important difference from GraphOps, which is designed to be more general. DeLorimier et al [7] presented GraphStep, a system architecture for sparse graphs that fit in the block memory of the FPGA. While general, their architecture is severely constrained in the size of dataset possible, particularly in an era of rapidly expanding dataset sizes and DRAM banks.

There has also been some work in the area of using a streaming paradigm to process graphs. Ediger et al [8] used a dynamic graph representation called STINGER to extract parallelism and enable streaming processing. Roy et al [18] proposed X-stream, which we presented in Section 6. GraphOps differs from these approaches, as they are both software-focused efforts. Another key difference is that these approaches attempt to better utilize the memory hierarchy, whereas GraphOps relies on main memory bandwidth.

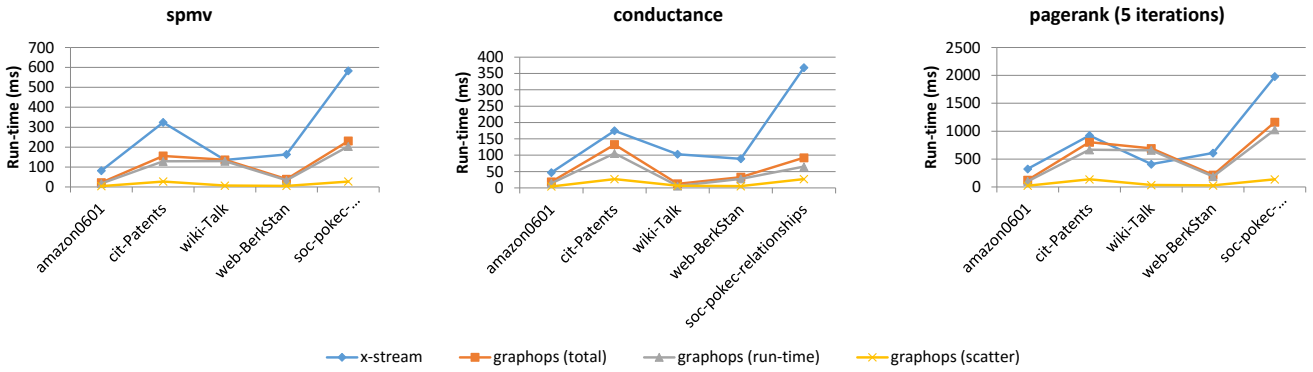

Figure 6: Run-time comparison of GraphOps and X-Stream.

8. CONCLUSION

In this paper, we present the GraphOps hardware library, a composable set of hardware blocks that allow a designer to quickly create an energy-efficient graph analytics accelerator. Using the well-known computation PageRank as a driving example, we explain how GraphOps addresses issues that a designer would otherwise spend valuable time solving and verifying. We evaluate the GraphOps library by composing six different accelerators and using them to process a variety of workloads. We compare with pure software implementations as well as with a software streaming framework. Results show that the GraphOps-based accelerators are able to operate close to the bandwidth limit of the FPGA system. Overall, this paper demonstrates that graph-specific FPGA acceleration can be achieved with a significant reduction in design time if a hardware designer is given a useful set of building blocks.

Acknowledgments

We would like to thank the Maxeler engineering team and the Maxeler University Program for their diligent help and support. We would also like to thank reviewers and the programming chair for their reviews and attention.

This work is supported by DARPA Contract- Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF Grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; NSF Grant- EAGER- XPS: DSD: Synthesizing Domain Specific Systems-CCF-1337375; and the Stanford PPL affiliates program.

9. REFERENCES

[1] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Tech Report NVR-2008-004, Nvidia Corporation, 2008.

[2] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj. A framework for fpga acceleration of large graph problems. In *FPT 2011*, pages 1–8. IEEE, 2011.

[3] N. Biggs. *Algebraic graph theory*. Cambridge university press, 1993.

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

[5] J. Chhugani, N. Satish, et al. Fast and efficient graph traversal algorithm for cpus. In *IPDPS 2012, IEEE 26th International*, pages 378–389. IEEE, 2012.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] M. DeLorimier, N. Kapre, N. Mehta, et al. Graphstep: A system architecture for sparse-graph algorithms. In *FCCM'06.*, pages 143–151. IEEE, 2006.

[8] D. Ediger, K. Jiang, et al. Massive streaming data analytics: A case study with clustering coefficients. In *IPDPSW 2010*, pages 1–8. IEEE, 2010.

[9] M. A. e. a. G. Malewicz. Pregel: a system for large-scale graph processing. In *ACM SIGMOD 2010*, pages 135–146. ACM, 2010.

[10] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on fpgas. In *IPDPS 2004*, page 149. IEEE, 2004.

[11] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[12] J. Leskovec and A. Krevl. Snap datasets:stanford large network dataset collection. 2014.

[13] E. Nurvitadhi, G. Weisz, et al. Graphgen: An fpga framework for vertex-centric graph computation. In *FCCM 2014*, pages 25–28. IEEE, 2014.

[14] T. Oguntebi. Graphops source repository. https://github.com/tayo/GraphOps.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking. 1999.

[16] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.

[17] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk. Maximum performance computing with dataflow engines. In *High-Performance Computing Using FPGAs*, pages 747–774. Springer, 2013.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP 2013*, pages 472–488. ACM, 2013.