FlexGraph: A Reconfigurable Graph Analytics Accelerator for Heterogeneous CPU-FPGA Architectures

Blaise-Pascal Tine
Georgia Tech
btine3@gatech.edu

David Sheffield
Intel Labs
david.b.sheffield@intel.com

Sudhakar Yalamanchili
Georgia Tech
sudha@gatech.edu

ABSTRACT

In recent years, The end of Dennard Scaling is pushing the computer architecture community towards designing more specialized and energy efficient systems. This move has led to a new application domain for FPGAs and the emergence of heterogeneous CPU-FPGA computing platforms, enabling the design of new energy efficient FPGA accelerators for domain specific applications. Graph Analytics, one the largest applications in production data centers today, faces scaling challenges with its ever increasing workload size, inherent sparsity and memory-bound characteristic. We present FlexGraph, a flexible and energy-efficient Graph Analytics Accelerator for heterogeneous CPU-FPGA architectures. FlexGraph uses a Doubly Compressed Sparse Matrix format to eliminate unnecessary data transfers and reduce storage requirements. Our architecture allows support for other compressed format by decoupling the matrix data structure traversal from the compute and memory units. We introduce memory access hardware primitives for unstructured fine-grained accesses on cache-coherent shared memory. Flexgraph uses a C++ Hardware Generation Language to extend its vertex programming model with domain centric reconfigurability in an integrated single source development environment, providing a design efficient alternative to High Level Synthesis. Flexgraph achieves X GFlop/s, performing Nx faster than HLS implementation.

1. INTRODUCTION

The end of Dennard Scaling [1] has moved the focus of computer architecture designers towards power and energy efficient architectures. However, energy efficient solutions such as ASIC designs present a serious limitation in flexibility and production cycle. These constraints have pushed production data centers towards using FPGA-based accelerators [2] for their reconfigurability and energy savings when compared to general-purpose graphics processing units (GPUs). This trend has led to emergence of heterogeneous CPU-FPGA computing platforms [3] [4], enabling fast development of new energy efficient accelerators [5] for domain specific applications. Graph Analytics is one the largest applications in production data centers today [6], spanning domains such as bio-informatics, social network, mining, cyber-security, etc. Graph Analytics performance suffers from workload imbalance, frequent updates, limited data locality and low compute communication ratio making it memory-bound and energy inefficient. This problem is further exacerbated with the ever increasing size of its dataset, presenting a scalability challenge for the industry. Several solutions have been proposed to address this problem both at the software level, with better algorithms and programming abstraction [7] [8] [9] [10] [11], and at the hardware level with custom accelerators [12] [13] [14]. Most Graph Analytics accelerators [12] [13] [14] define a proprietary graph data structure to exploit efficient computation on their hardware, limiting flexibility on the host processor for efficient software processing. FlexGraph uses sparse matrices as underlying data structure to enable efficient computation in a shared CPU-FPGA environment where both the host processor and the accelerator are modifying the same graph. However, accessing unstructured fine-grained data on a cache coherent FPGA fabric pauses challenges because of the restricted coarse-grained cache line access granularity and longer miss penalty. Additionally, generalized sparse matrix storage encoding COO, ELL, CSR, CSC, lack the same level of compaction as directly using non-zero edges list [12], pausing a problem for energy efficiency and memory bandwidth for hyper-sparse Graph Analytics application. FlexGraph uses a Doubly Compressed Column Based Sparse matrix encoding similar to GraphMath [9], however introducing additional indirections to the matrix traversal path. We decoupled FlexGraph's matrix traversal unit from the rest of the compute and memory fabric to scaling by increasing memory bandwidth and provide support for other formats without altering the rest of the system.
Our contributions are as follows:
1. A specialized Graph Analytics Accelerator for heterogeneous CPU-FPGA fabric that provide efficient collaborative computation while maximizing energy efficiency.
2. A decoupled data structure traversal and compute architecture for sparse matrices enabling maximum bandwidth utilization and compute scaling.
3. We introduce hardware primitives for unstructured fine-grained accesses on coherent shared memory architectures.
4. A Doubly-Compressed Sparse Matrix-Sparse Vector Multiplication Accelerator implementation on FPGA. 5. A domain specific accelerator with single source programming and reconfiguration environment providing a design-efficient alternative to High-Level Synthesis.

The remainder of this paper is organized as follows: Section 2 provides a background and motivation for the FlexGrah accelerator, section 3 describes FlexGraph's architecture, section 4 describes FlexGraph's software-hardware codesign using Cash [15] framework, section 5 describes the experimental setup, section 6 describes our results analysis, section 7 describes the related work, section 8 summarizes our main contribution and results.

2. BACKGROUND AND MOTIVATION

In this section, we provide the background and motivation for this work.

2.1 Graph Vertex Processing

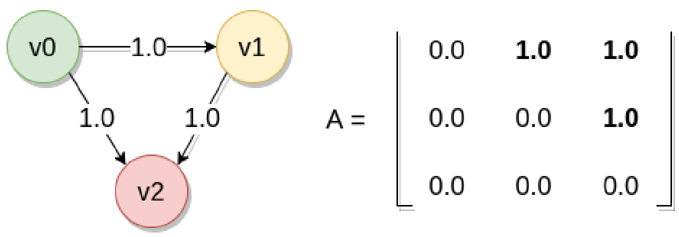


Figure 1: Sample Graph Representation

$$\#Edges = A^T \times Identity \qquad (1)$$

$$\#Edges = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \qquad (2)$$

Figure 2: Incoming Edges Algorithm

```
GraphProgram(G, P, N) :
  for_each N :
    x := Assign(P)
    y := Process(G,x)
    P := Apply(P,y)

e.g. FindIncomingEdges(G=A, P=I, N=1) :
          Assign(P)       := P
    where Process(G,x)  := G * x
          Apply(P,y)      := y
```

Listing 1: Graph Vertex Processing Model

A large variety of programming models have been proposed for describing Graph algorithms, expressing computation using vertex operations on matrices [9] [11], [10], [16] [7] task-based models [8] or domain-specific languages [17]. The vertex programming model has show great adoption for its ease of abstraction for describing algorithms using Linear Algebra and its efficient computation on commodity multi-core processors.

Figure 1 shows a simple 3 vertices graph example with its corresponding 3x3 matrix representation. For every edge between a source and destination vertex, the corresponding row and column entry in the matrix is activate. For instance, the matrix entry in row 0 and column 1 represents the edge between source *v0* and destination *v1*. using the matrix representation, we can apply an algorithm on the graph to calculate the total number of incoming edges at each vertex. This algorithm can be expressed using a simple matrix-vector operation as shown in equations (1) and (2) in Figure 2. Several graph algorithms, including Breadth First Search (BFS) [18], PageRank [19], Single Source Shortest-Path (SSSP) [20], can be described similarly using the vertex programming model [9].

Listing 1 shows the three stages of a generalized graph vertex processing model - *Assign*, *Process* and *Apply*, given a graph *G*, some vertex properties *P* and an iteration count *N*. The *Assign* stage generates an input vector *x* using the vertex properties *P*. The *Process* stage applies the input vector *x* to the graph *G* and generates an output vector *y*. The *Apply* gather the resulting output vector to update the vertex properties for the next iteration. The graph program executes the three stages for several iterations until it converges. A direct mapping of our incoming edges algorithm to this processing model is also provided in Listing 1, where the identity vector is used for the vector properties. The *Assign* and *Apply* stages of this program are simple identity operators, while the *Process* stage perform a matrix-vector multiplication. The input graph can be represented as a compressed sparse matrix to only store the non-zero entries of the graph. This data structure enables accelerated computation, often using a Sparse Matrix-Sparse Vector Multiplication (SpMSpV) kernel to target multi-core CPUs [9] or GPUs [16]. The computation that cannot be accelerated is simply executed on the general-purpose host CPU. FlexGraph's objective is to provide a more energy efficient graph processing backend using a FPGA for hardware acceleration [2].

2.2 Intel Heterogeneous CPU-FPGA Platform

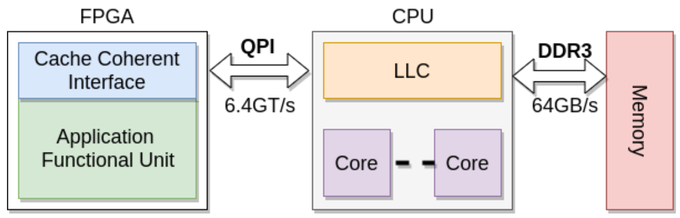
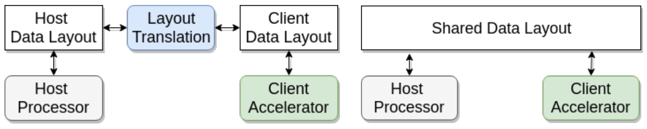


Figure 3: Intel HARP Architecture

With the rising adoption of FPGAs in production data centers [2], there is a need to increase its ecosystem by making them more energy efficient as well as accessible to both the users and designers. Modern CPU-FPGA platforms [5] provide tightly-coupled shared-memory CPU-FPGA integration [21] [22], making then easier to program and efficient for hardware acceleration. Figure 3 shows an overview architecture of the Intel Xeon-FPGA platform prototype that we used in our evaluation. It has a dual-socket system, one containing a 10-core Intel Xeon E5-2680 CPU and the other containing an Altera Stratix V 5SGXEA FPGA operating at 200 MHz. The two sockets are connected via a 6.4 GT/s QPI [23] link for data transfer. The FPGA's area is partitioned into two main blocks - the Application Function Unit (AFU) or 'Green' bitstream allocated for the custom accelerator and the 'Blue' bitstream implementing the Cache Coherent Interface (CCI) for the accelerator. The CCI runtime implements a 64 KB direct-mapped cache to support address translation (1024 page table entries) with request re-ordering for a total of 2 GB of addressable memory. The AFU memory interface is 64-byte cache-line addressable and it is up to the accelerator to extend the interface if fine grained (e.g. single byte) access is desired.

2.3 Collaborative CPU-FPGA Computation



(a) Offload model (b) Collaborative model

Figure 4: Shared Memory Compute Models

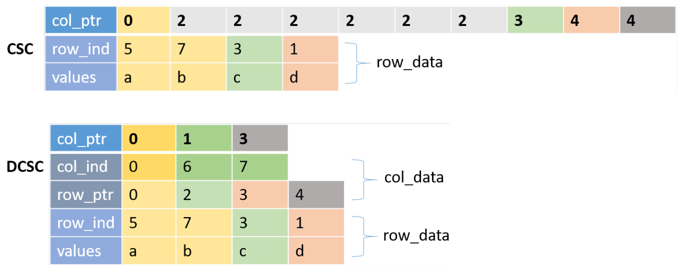
The prevalent execution model for accelerator design is the offload model [24] where computation takes place on discrete copies of a shared resource that are optimized for efficient access by the target device. In this model of computation, the host processor applies some layout translation on the data before making it available to the client accelerator (see Figure 4a). The additional latency of the translation process is generally mitigated using optimization techniques such as batching and double-buffering. The offload model is well suited for discrete memory systems where the necessary data transfer can be coupled with layout translation. One attractive application of shared-memory systems is the enabling of efficient collaborative computation (see Figure 4b) in which all attached compute elements can access the same shared resource and modifying it during the course of the program execution. It is particularly ideal if the data layout provides efficient access by the compute devices. Contrarily to existing Graph Analytics Accelerators [12] [14] that employ a custom optimized graph data structure for computation, Flexgraph uses a sparse matrix format for computation. This decision provides several advantages - first, it is storage efficient because the graph's physical memory is shared - second, it enables collaborative computation with the host CPU, leveraging the large ecosystem of matrix-based frameworks [25] [26] - third, it decouples the software and hardware design, enabling them to scale independently.

3. FLEXGRAPH ARCHITECTURE

In this section, we describes the overall architecture of the FlexGraph Accelerator.

3.1 The DCSC Matrix Format

FlexGraph uses the Doubly Compressed Sparse Column (DCSC) [27] format as underlying graph data structure. The format allows minimal traversal into the sparse matrix structure, saving necessary memory bandwidth when fetching empty columns. Figure 5 shows the memory layout for a sample matrix A = {(0,5,a), (0,7,b), (6,3,c), (7,1,d)} with four non-zero edges (src, dst, weight) encoded using DCSC versus the conventional CSC [28] format. The traversal over CSC requires accessing all consecutive column ranges *(cols[i+2] - cols[i])* even though most of the distances are empty. DCSC



CSC

col_ptr	0	2	2	2	2	2	2	3	4	4
row_ind	5	7	3	1						
values	a	b	c	d						

DCSC

col_ptr	0	1	3	
col_ind	0	6	7	
row_ptr	0	2	3	4
row_ind	5	7	3	1
values	a	b	c	d

Figure 5: Matrix formats comparison

saves on bandwidth by using an additional indirection buffer inside its structure to only encode non-zero columns. It is important to note that this indirection can introduce additional storage overhead and access latency if the matrix is too small or not sparse enough. However, Graph Analytics datasets are generally very large and hyper-sparse which eliminate the DCSC format overhead. In addition to a sparse matrix, FlexGraph also uses a sparse vector to provide property data as well as edge selection during computation. A bitmask is used to encode the non-zero entries in the input vector.

3.2 The Sparse Matrix-Sparse Vector Multiplication Kernel

```
1  def SpMSpV_kernel(a, x):
2    y_values[] = {0}
3    y_bitmask[] = {false}
4    for_each i in a.col_ptr
5      (col_ind, row_ptr) := a.col_data[i]
6      x_active = x.bitmask[col_ind]
7      if (x_active):
8        x_val = x.values[col_ind]
9        for_each j in row_ptr
10         (row_ind, a_val) := a.row_data[j]
11         y_values[row_ind] += a_val * x_val
12         y_bitmask[row_ind] = true
13   return (y_values, y_bitmask)
```

Listing 2: Pseudo-code for SpMSpV kernel

FlexGraph micro-architecture implements a Sparse Matrix-Sparse Vertex Multiplication (SpMSpV) kernel in FPGA. This kernel diverges from conventional Sparse Matrix-Vector Multiplication (SpMV) implementations in two ways - firstly, it uses a sparse data structure as input and output vector during computation - secondly, it uses a doubly-compressed sparse matrix (DCSC) format with additional memory indirection buffer when accessing the matrix columns. These two properties present unique performance challenges when designing the accelerator. Listing 2 shows the pseudo-code of the SpMSpV kernel. Given the start/end addresses (col_start, col_end) into the matrix columns buffer (coldata), for The program iterates through each column entry and fetches the corresponding column index (col) and rows buffer address (row_start, row_end). It then checks if the current column index (col) is active using the vertex bitmask, only then it proceeds with rows traversal loop where the matrix row_data is accessed to retrieve corresponding row index and matrix value for computation. A Multiply-Accumulate (MAC) operation is then performed on the matrix and vertex data and output bitmask is updated. Lines 6 and 8 show the code region where external memory is randomly accessed to obtain the vertex data and active state. Lines 5 and 10 show the code region where external memory is accessed semi-random because only the first access cannot be predicted and after that the address is simply incremented. FlexGraph's architecture attempts to address some of those performance hogs using several optimizations detailed in section 4.

3.3 FlexGraph Microarchitecture

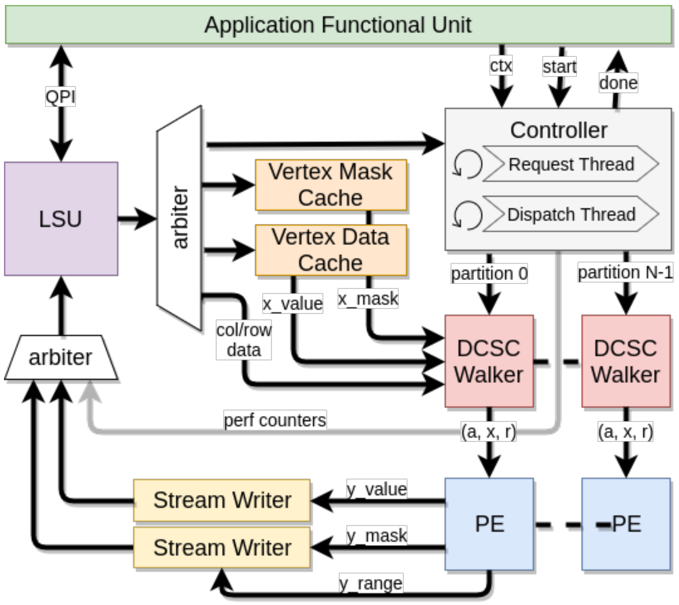


Figure 6: FlexGraph Microarchitecture

Figure 6 illustrates the microarchitecture of FlexGraph accelerator. Externally, FlexGraph input and output ports implement an Accelerator Functional Unit (AFU) interface defined by Intel's Accelerator Abstraction Layer (AAL) [3]. AFUs implementing the interface are able to bind with the FPGA's board support package (BSP) and seamlessly communicate with the AAL runtime on the host processor. The interface exposes four signals *qpi, ctx, start, done*, where the *qpi* bus handles shared-memory communications, *ctx* is used for passing custom input context to the accelerator, in our case we pass down all data structure layout information needed to access memory, *start* and *done* are the control knobs for starting and ending execution. At a high-level, Flexgraph's microarchitecture consists for a six building blocks: The main controller, the DCSC matrix walkers, the processing elements, the stream writers, the vertex caches and the load/store unit (LSU).

3.4 Main Controller

Upon reception of the *start* signal from the host processor, the main controller is responsible for scheduling matrix partitions for execution on each processing element. It is comprised of two execution threads: A request thread that fetches partition data from the load/store unit into a local buffer, a dispatch thread that submit received partition data to individual matrix walker engines for processing. After all matrix partitions have been processed, the controller sends performance counters back to the host processor and activates the *done* signal.

3.5 DCSC Matrix Walkers

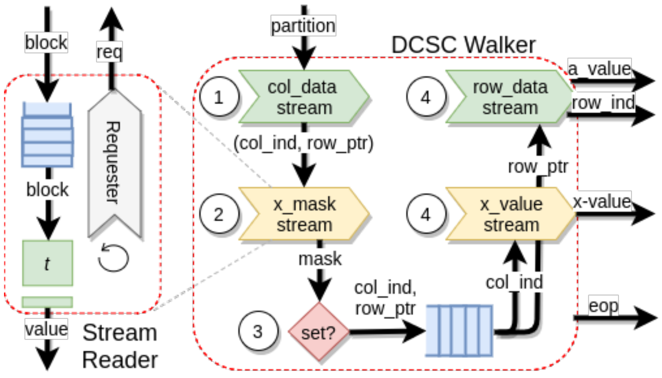


Figure 7: DCSC Walker

The DCSC Matrix Walker is a custom data structure traversal engine for the DCSC Matrix. Figure 7 shows an overview of the engine microarchitecture. It is implemented as a four-stage state machine where each stage accesses a portion of the matrix layout as reflected in the pseudo-code in Listing 2. In the first stage, the engine fetches column data (*col_ind, row_ptr*) from memory given the partition address. In the second stage, the engine fetches the vertex active mask from memory given the *col_ind* value obtained in the previous stage. In the third stage, the engine checks if the current property is active using the mask obtained in the previous stage, if so passes the associated column data to the next stage. In the fourth stage, the engine fetches the vertex data and the matrix row data (*row_ind, a_val*) from memory for each active column data and send those values to the connected processing element where the actual computation takes place. Each fetch unit is implemented as a Stream Reader engine to handle coarse-grain memory access for the associated data. It consists of a request module for sending memory read access command, a fifo buffer for receiving requested blocks and a temporary block buffer for accessing sub-block elements. The vertex data (values, masks) are fetched from dedicated vertex caches to enable reuse by the walker engines. We use a crossbar to multiplex outgoing requests from the walker engines to the LSU and vertex caches.

3.6 Processing Elements

Figure 8 shows an overview of a processing element microarchitecture. It main objective is to perform a floating-point multiplication on the incoming vertex and matrix values from the connected walker engine and accumulate the result until the whole partition has been processed. It is comprised of eight parallel pipelined Multiply-Accumulate (MAC) units attached to a coalescing output module. Eight is the maximum number of matrix row data that can be stored in a single cache block. Each MAC lane consists of pipelined floating-point multiply module, a pipelined floating-point Add unit, a Read-After-Write (RAW) control unit that stalls the

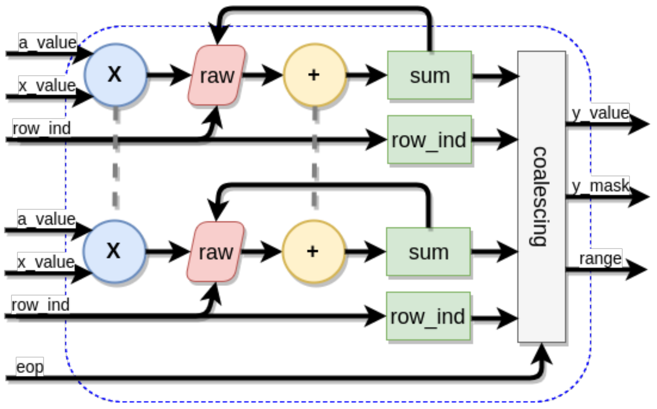


Figure 8: Processing Element

pipeline if the current *row_ind* is pending addition. We build a bitmask given the *row_ind* to track outstanding operations in addition units. The accumulated result is stored into a local buffer indexed by the *row_ind* value. Each processing element also computes the aggregate active rows mask for the given partition. At the end of the partition processing, the coalescing unit merges output values and masks according to their associated block address and sends the result to the Stream Writers.

3.7 Stream Writers

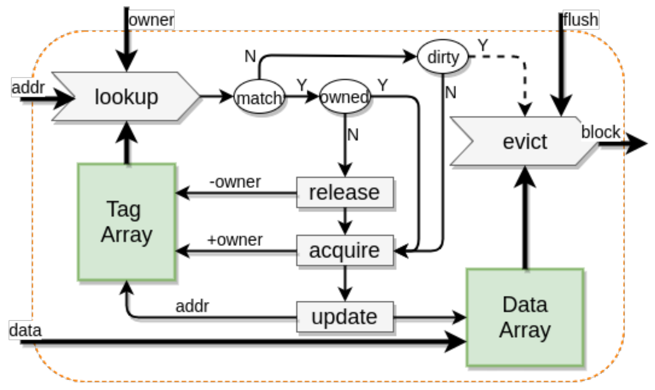


Figure 9: Stream Writer

The Stream Writers are coalescing write buffers that batches write requests to memory until entire blocks are updated before they are submitted. We used two stream writers, one for the storing vertex output and one for storing the output mask. Figure 9 shows an overview of a stream writer microarchitecture. Internally, a stream writer has an architecture similar to a cache but has been augmented with block ownership information and the stored value is merged, not overwritten. It consists of four main modules: A data store, A tag store, and lookup unit and an eviction unit. The data store holds all the blocks sent to the device. The tag store keeps the meta-data associated with each block in the data store. A tag consists of the block address and an owners list. The owners list is implemented as a bitmask that tracks the processing elements currently using a given block. In a typical operation where the writer receives a block write request, the lookup unit is first invoked to find an existing entry for the corresponding block in the tag store. If there is a match and the block is already owned, the incoming value is simply merged with existing one. If a match was found, but the block is not owned, the previosly owned block is released and this new one is acquired before data update. If no match is found, the device allocates a new block for the request and evicts existing data to memory if dirty. The writer also support an explicit flush command to the eviction of all dirty blocks to memory when all processing is completed.

3.8 Vertex Caches

FlexGraph uses two direct-mapped vertex caches to keep the vertex values and mask in the accelerator for reuse by the matrix walker engines. Their source port is connected to the load/store unit via an arbiter that multiplexes outgoing read requests with the controller and the matrix walkers. It has a typical cache architecture with an input buffer for tracking outstanding requests.

4. FLEXGRAPH SOFTWARE-HARDWARE CODESIGN

```
class aal_device {
public:
  virtual out_t initialize()(
    const ch_logic& start,
    const qpi::in_t& qpi_in,
    const afu_ctx_t& ctx,
    qpi::out_t& qpi_out,
    ch_logic& done
  ) const = 0;
};
```

Listing 3: AAL Device Interface in Cocoh C++

5. EXPERIMENTAL SETUP

In this work, we simulated our evaluation environment entirely in software using the Cocoh [29] C++ framework. Figure 10 shows an overview of what the simulation environment contains. At the top we have the accelerator implementation with its AAL interface, both modelled using Cocoh's API. At the bottom, we have the Cocoh's simulation runtime that bind the accelerator and the back-end host CPU simulator. Inside our host CPU simulator we modelled a QPI memory simulator which managed the shared memory space and served all the memory requests coming from the accelerator. In the current design we used a 1 GB shared memory space for holding the matrix and vertex data, including the output result from the accelerator. On the actual Xeon platform [3], the shared memory space limit is 2 GB.

5.1 Intel QPI Simulation

We simulated the Intel QPI memory transaction using an analytic model capturing the latency of memory transfers when the request hits local CCI [22] cache or the host processor LLC. All outgoing memory requests from the accelerator to CCI take two cycles and the responses from CCI hits take another two cyles, giving a minimum round trip latency of four cycles for read requests. Figure 11 shows some

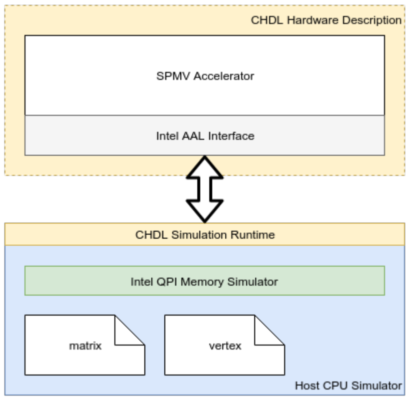


Figure 10: FlexGraph Simulation Environment

Simulation Parameters		Accelerator Parameters	
Shared Memory	1 GB	Clock Frequency	200 Mhz
QPI Max Requests	90	Vertex Data Cache	8x64 Bytes
CCI Hit Rate	80%	Vertex Mask Cache	8x64 Bytes
LLC Hit Rate	50%	Cols Data Buffers	2x2x64 Bytes
CCI Latency	4 cycles	Rows Data Buffers	2x5x64 Bytes
LLC Latency	32 cycles	Output Buffer per PE	2x64 Bytes
Memory Latency	100 cycles	Number of PEs	2

Figure 11: FlexGraph Simulation Parameters

configuration parameters we use for both the simulation and accelerator modelling.

5.2 Graph Datasets

We used the Graph500 [30] synthetic datasets for our evaluation. We configured graph generator similar to GraphMat [9] setting the default RMAT paramters (A=0.57, B=C=0.15 and D=1-A-B-C). We used five scaling factors for our graphs (6, 8, 10, 12, 14) with corresponding number of non-zero ranging from 1024 to 262,144 accordingly. These datasets are relatively small mainly due to simulation time constraints, but the setup could have supported up to a scaling factor of 20 (about 16,777,216 non-zeros). Due to time constraints we did not test any real world dataset, that will be discussed in our future work section. Figure 12 lists down the number of vertices and non-zeros for our dataset.

dataset	m6	m8	m10	m12	m14
scale factor	6	8	10	12	14
num vertices	64	256	1024	4096	16384
Non-zeros	1024	4096	16384	65536	262144

Figure 12: Graph500 dataset

6. RESULTS ANALYSIS

In this section, we discuss our evaluation results.

6.1 Throughput Performance

Figure 13 shows the throughput performance of the accelerator in GFlops. We estimated the performance using the following equation:

$$Perf = \frac{2 * nonzeros}{latency}$$

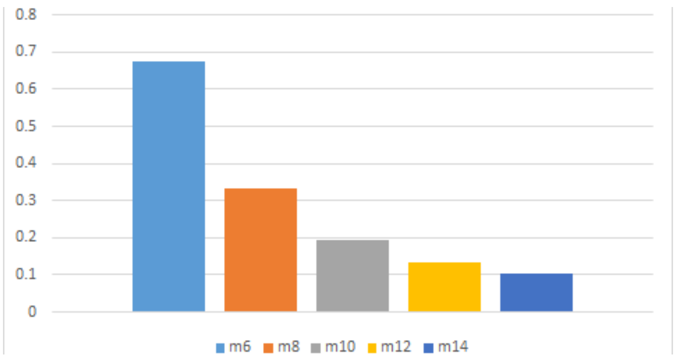


Figure 13: FlexGraph Throughput Performance

The performance varies from 0.67 Gflops to 0.1 Gflops as the size of the dataset increases. This result is comparable to our implementation of a similar kernel using the Altera OpenCL HLS, also showing similar performance degradation as the workload size increases. In prior work, GraphOps [14] peak performance for their FPGA implementation was 0.18 GFlops with a graph size of 512K vertices, about x3 our throughput. The target platform has peak memory bandwidth is 6.0 GB/s, this shows that there is room for improvement. than We identified several optimization opportunities that we will discuss later in this section.

6.2 The Impact of Parallelism

Figure 14 shows the throughput performance of the accelerator in GFlops for a single-core system versus the dual-core implementation. We can see a considerable drop in performance (by almost half) across all datasets when a single processing element is executing. This is happening because there is much less memory level parallelism that can be exploited when a single core is running. In a dual-core system, the LSU receives a lot more memory requests from the processing element and has much less idle time when QPI is taking more cycles to return the blocks. When a processing element is stalled on a request, another may be able to make progress. We believe that increase the number of processing element beyond two will have a possible impact in performance for FlexGraph, possibly doubling the current throughput.

6.3 The Impact of Memory Optimizations

Figure 15 shows the accelerator performance with all memory optimizations disabled in GFlops. Once again the graph shows a similar performance degradation as the dataset size decreases. The memory optimizations give the accelerator an average performance speed-up of 45%, which is significant considering the fact the the system is memory bound.

6.4 Summary Discussion

In summary, FlexGraph current average performance is 0.1 Gflops (using the lower bound of the largest dataset in our experiment). This is in part with our HLS implementa-

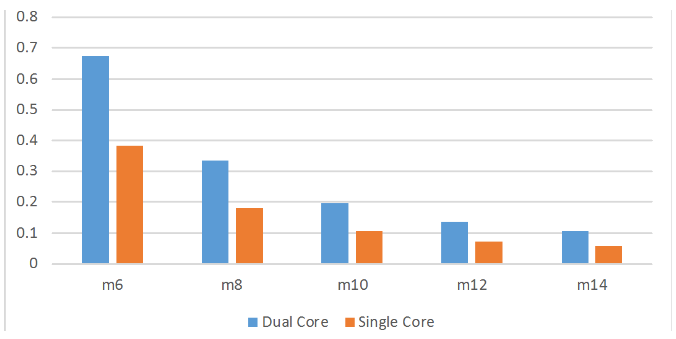


Figure 14: FlexGraph Single Core Performance

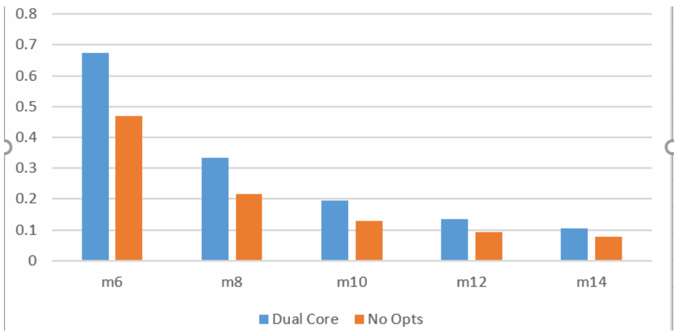


Figure 15: FlexGraph Performance with memory optimizations

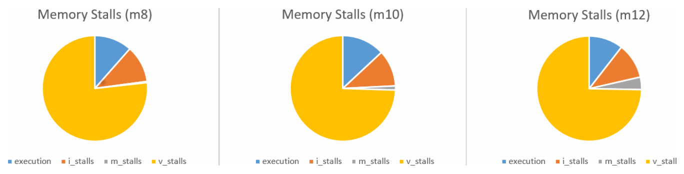


Figure 16: FlexGraph Memory Stalls

Memory objects	21
Registers	143
Muxes	785
ALU ops	1914
Literals	93

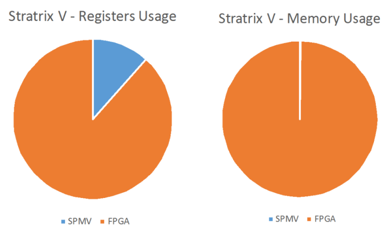


Figure 17: Cocoh's hardware cost evaluation

tion of the same kernel. We have identified several optimization opportunities to boost the accelerator performance.
We added some performance counters into FlexGraph accelerator to monitor the memory stalls in each processing elements and identify bottlenecks. Figure 16 shows the memory stalls distribution over three datasets *m8*, *m10* and *m12*. The blue region represents the execution cycles, the orange region represents the stall cycles occurring when we access the matrix column indices, this is *coldata* buffer in the SPMV speudo-code (line 4 in listing 1). The gray region represents the stall cycles when accessing the vertices active masks (line 6 in listing 1). We can significantly reduce the stall cycles by prefetching the blocks before they are needed to hide the memory latency. Using a simple next-line prefetcher will suffice since we know the access pattern on *coldata*.
Another performance optimization is in reducing the impact of the branch operation when checking if a vertex is active of not (line 7 in listing 1). We will investigate adding a preliminary test on the entire 32-bit mask to see if it is zero and skip the whole block altogether.
Lastly, another performance opportunity is increasing the number of precessing elements. Figure 14 showed the performance gain of using two cores versus a single one. Having multiple processing elements will improve the system memory level parallelism by overlapping the execution of some processing element while others are stalled on memory accesses. We ran some estimate of the current design hardware cost using Cocoh's framework (see Figure 17) and it showed that our memory usage is very small (about 1% of the target Stratix V FPGA capacity). Also, FlexGraph accelerator only currently consumes 15% of the registers capacity. With this reserve, we anticipate extending the number of processing elements to 16. A challenge that we will have to resolve when adding additional processing elements is the synchronization of write accesses for the active masks. Another challenge will be the arbitration latency in the LSU when communicating with all nodes, since the single LSU will now become a bottleneck in the system.

7. RELATED WORK

In this section, we discuss prior works related to this project.

7.1 GraphMat

GraphMat [9] is a high-performance Graph Analytics Framework for multi-core CPU platforms. It defines a C++ template-based vertex programming model for describing various graph algorithms in software and use Sparse Matrix Vector Multiplication (SPMV) as underlying compute kernel for efficient parallel processing. GraphMat also uses a doubly-compressed DCSC [27] matrix format for storage and compute efficiency. FlexGraph's programming model is an adaptation of GraphMat's model for collaborative computation with an FPGA accelerator.

7.2 Graphicionado

Graphicionado [12] is a much recent project that is similar to FlexGraph. In this work, the authors implemented a graph accelerator targeting Intel Xeon platform. Graphicionado also uses a vertex programming model of graphs computation similar to GraphMat [9]. It is a more specialized accelerator compared to FlexGraph in that it implements the entire graph algorithm in hardware, sacrificing flexibility for efficiency. Although the design allows some reconfigurability using FPGA, it still add many inconveniences compared to software. FlexGraph on the other hand only accelerate the graph edges computation in hardware, most of the algorithm is still written expressed in C++ and runs on the host processor like in GraphMat. FlexGraph reconfigurability is minimal and restricted to the SPMV module (changing the matrix datatype or the reduce operation). Graphicionado main architecture strength is their efficient optimizations of the graph access patterns to reduce stalls inside the pipeline.

7.3 GraphOps

GraphOps [14] is another graph accelerator also targeting FPGAs like FlexGraph. In this work, the authors propose an accelerator architecture that efficiently process a graph in memory encoded using a proposed locality-optimized scheme. Their main contribution is the graph storage representation which provides an efficient memory access pattern. A drawback of their proposal is that fact that the host processor has to pre-process the graph before the accelerator can consume it, which can add some considerable latency, considering that Graph Analytics algorithms tend to update the graph frequently.

8. FUTURE WORK

As stated in the results discussion, FlexGraph performance can be greatly improved once we implement the proposed optimizations. Doing this first step will be our priority for future work. The Cocoh C++ library we used in this work did not yet have support for verilog generation, we plan to add that support such that we could install the accelerator on the actual FPGA board for evaluation. We also anticipate to do some power and energy efficiency analysis ion the final design, hopefully identifying all the components in the system consuming the most energy. We also plan to add some real world datasets to our evaluation, popular benchmarks include Flicker, Facebook, Wikipedia, Twitter, Netflix and LiveJournal.

9. CONCLUSION

In this paper we presented FlexGraph, a reconfigurable Graph Analytics Accelerator targeting commodity heterogeneous CPU-FPGA platforms. FlexGraph was implemented using Cocoh C++ libary, providing a productive and efficiency development support for modifying the accelerator when needed to accommodate all graph algorithms. FlexGraph architecture implements a customized SPMV kernel adapted for graphs vertex computation. It implements some memory optimizations and tasks parallelization to improve its performance. FlexGraph current performance is not ideal and we identified several strategies to fix that, including doing some memory prefetching and adding additional processing elements.

10. ACKNOWLEDGMENT

The authors gratefully acknowledge the insightful feedback and input from Professor Heysoon Kim from Georgia Institute of Technology and David Sheffield from Intel Labs.

This research was supported by a grant from the National Science Foundation under grant CCF-1533767.

11. REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, June 2011.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” *SIGARCH Comput. Archit. News*, vol. 42, pp. 13–24, June 2014.

[3] P. Gupta, “Intel xeon+fpga platform for the data center.” `http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf`.

[4] Y. L. Fei Chen, “Fpga acceleration in a power8 cloud.” `https://openpowerfoundation.org/blogs/fpga-acceleration-in-a-power8-cloud`.

[5] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, “A quantitative analysis on microarchitectures of modern cpu-fpga platforms,” in *Proceedings of the 53rd Annual Design Automation Conference*, DAC ’16, (New York, NY, USA), pp. 109:1–109:6, ACM, 2016.

[6] D. Gage, “The new shape of big data.” `http://www.wsj.com/articles/SB10001424127887323452204578288264046780392`, 2013.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, “Apache spark’s api for graph and graph-parallel computation.” `http://spark.apache.org/graphx/`.

[8] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[9] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *Proc. VLDB Endow.*, vol. 8, pp. 1214–1225, July 2015.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: A framework for machine learning and data mining in the cloud,” *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[12] T. J. Ham, W. Lisa, S. Narayanan, S. Nadathur, and M. Margaret, “Graphicionado: A high-performance and energy efficient accelerator for graph analytics,” in *the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “Tesseract: A scalable processing-in-memory accelerator for parallel graph processing,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA ’15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[14] T. Oguntebi and K. Olukotun, “Graphops: A dataflow library for graph analytics acceleration,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’16, (New York, NY, USA), pp. 111–117, ACM, 2016.

[15] B. P. Tine, “Cash: A c++ api and simulator for hardware,” 2017.

[16] Z. Fu, M. Personick, and B. 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*, GRADES’14, (New York, NY, USA), pp. 2:1–2:6, ACM, 2014.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-marl: A dsl for easy and efficient graph analysis,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 349–362, ACM, 2012.

[18] wikipedia, “Breadth-first search.” `https://en.wikipedia.org/wiki/Breadth-first_search`.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.,” Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[20] wikipedia, “Shortest path problem.” `https://en.wikipedia.org/wiki/Shortest_path_problem`.

[21] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, “Capi: A coherent accelerator processor interface,” *IBM Journal of Research and Development*, vol. 59, pp. 7:1–7:7, Jan 2015.

[22] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, “A reconfigurable computing system based on a cache-coherent fabric,” in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 80–85, Nov 2011.

[23] wikipedia, “Intel quickpath interconnect.” `https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect`.

[24] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik, “A taxonomy of accelerator architectures and their programming models,” *IBM J. Res. Dev.*, vol. 54, pp. 473–482, Sept. 2010.

[25] A. Buluç and J. R. Gilbert, “The combinatorial blas: Design, implementation, and applications,” *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 496–509, Nov. 2011.

[26] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM ’09, (Washington, DC, USA), pp. 229–238, IEEE Computer Society, 2009.

[27] A. Buluc and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–11, April 2008.

[28] wikipedia, “Sparse matrix.” `https://en.wikipedia.org/wiki/Sparse_matrix`.

[29] B. Tine, “Cocoh, a c++ domain specific library for hardware design and simulation.” `http://casl.gatech.edu/research/cocoh`, 2017.

[30] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, “Introducing the graph 500.” `http://www.graph500.org/`, 2010.