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 details the various architecture optimizations we used in FlexGraph, section 5 describes FlexGraph's software-hardware codesign using Cash [15] framework, section 6 describes the experimental setup, section 7 describes our results analysis, section 8 describes the related work, section 9 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

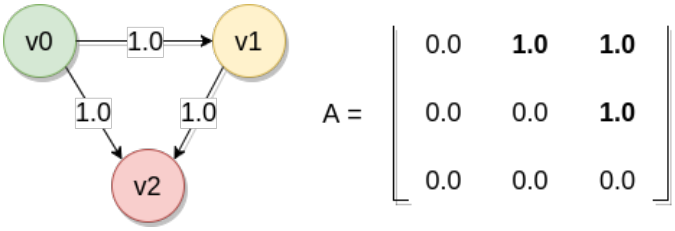

Figure 1: Sample Graph Representation

$$\#Edges = A^T \times Identity \tag{1}$$

$$\#Edges = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \tag{2}$$

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 (SpM-SpV) 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

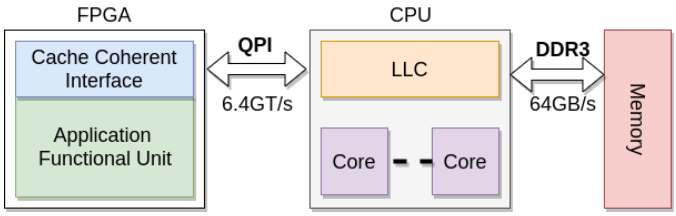

Figure 3: Intel HARP Architecture

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 accel-

(a) Offload model

(b) Collaborative model

Figure 4: Shared Memory Compute Models

erator 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

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 a sample matrix A = {(0,5,a),

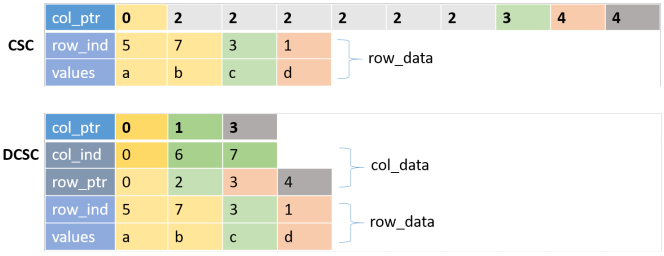

Figure 5: DCSC hyperspace matrix format

(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 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

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 1 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.

```
1  def SpMSpV_kernel(a, x):
2    y_values[] = {0}
3    y_bitmask[] = {false}
4    for i in (a.col_start, a.col_end):
5      (col, row_start, row_end) = a.coldata[i]
6      x_active = x.bitmask[col]
7      if (x_active) :
8        x_value  = x.values[col]
9        for j in (row_start, row_end):
10         (row, a_value) = a.rowdata[j]
11         y_values[row] += a_value * x_value
12         y_bitmask[row] = true
13   return (y_values, y_bitmask)
```

Listing 2: Pseudo-code for SpMSpV kernel

3.3 FlexGraph Microarchitecture

Externally, FlexGraph input and output signals 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 software running on the host processor. Listing 2 shows AAL device interface implementation using Cocoh's API. Our FlexGraph Accelerator's class in Cocoh simply derives from this interface and override the *initialize()* function to provide its implementation and Cocoh takes care of generating the corresponding verilog module. It implements three input signals *(start, qpi_in, ctx)* and two output signals *(qpi_out, done)*. The AFU starts execution when the *start* signal is asserted and communicate completion by asserting the *done* signal. The *ctx* signal provides application's specific data like constants and buffers address in the case of FlexGraph. The *qpi* in/out signals implement Intel Quick Path Interconnect (QPI) interface [23], providing single channel read/write ports for accessing external shared memory. The FPGA socket hosts a 64-byte cache coherent interface (CCI) [22] that connects to the host processor's last level cache (LLC) via QPI.

```
1  class aal_device {
2  public:
3    virtual out_t initialize()(
4      const ch_logic& start,
5      const qpi::in_t& qpi_in,
6      const afu_ctx_t& ctx,
7      qpi::out_t& qpi_out,
8      ch_logic& done
9    ) const = 0;
10 };
```

Listing 3: AAL Device Interface in Cocoh C++

Figure 6 illustrates FlexGraph accelerator microarchitecture. It is comprised of four main module types, a controller,

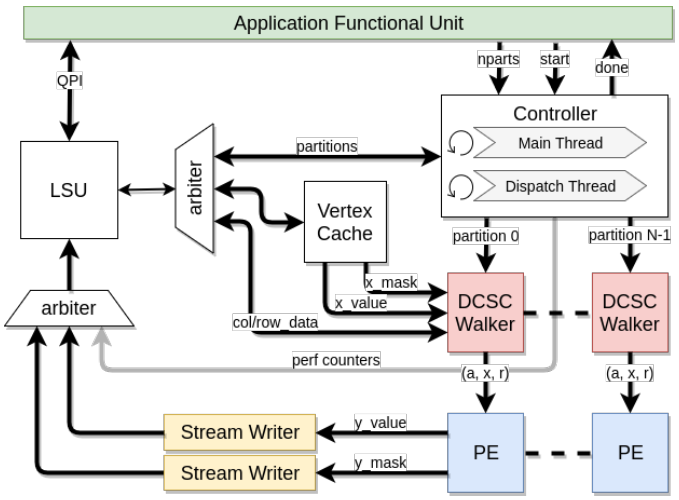

Figure 6: FlexGraph Microarchitecture

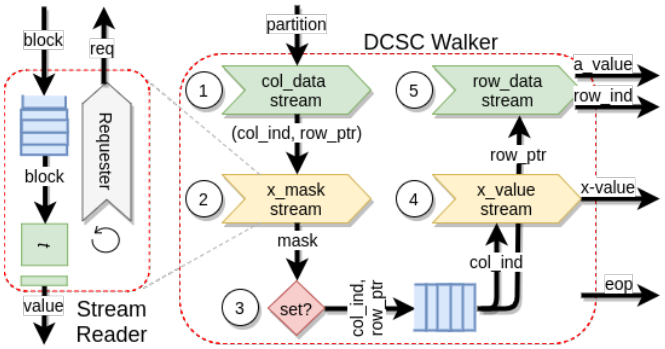

Figure 7: DCSC Walker

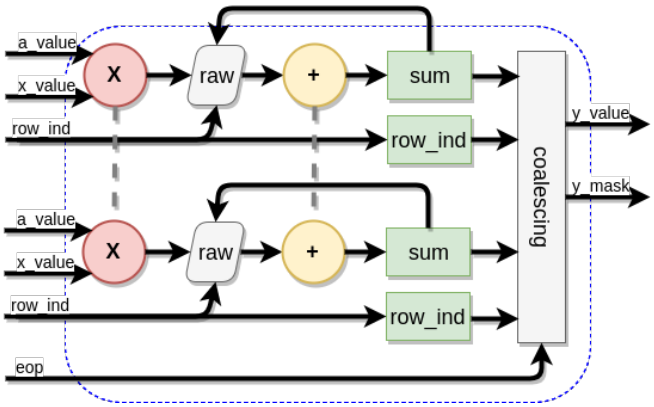

Figure 8: Processing Element

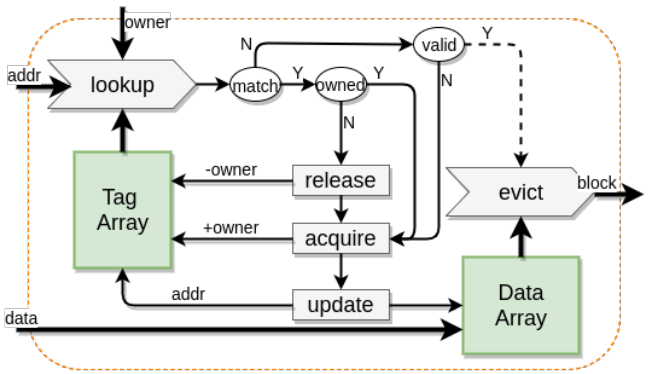

Figure 9: Stream Writer

processing elements (PEs), a load store unit (LSU) and vertex caches. The main controller is responsible for starting the accelerator, scheduling tasks for execution, reporting hardware counters and terminating the execution. The processing elements execute partition tasks assigned to them by the controller and communicate with the LSU to access the matrix and vertex data for their partition. They are also responsible for sending their final output result back to memory. The LSU is the module responsible for managing external communication between the accelerator and memory via the QPI interface. It directly binds to the QPI ports defined in listing 2. The vertex caches store intermediate vertex data and active masks for sharing between the processing elements. FlexGraph processing pipeline sightly resembles the SPMV execution steps illustrated in listing 1. We made several important modifications to it to improve performance.

4. FLEXGRAPH OPTIMIZATIONS

The first optimization we performed in FlexGraph early during the design phase was the support of multiple processing units. The reasoning behind it was to extract the maximum bandwidth out of the LSU such that the QPi is always busy processing a request when some processing element are stalled waiting for their data to return. the other advantage of parallelization for FlexGraph is to increase the overall accelerator throughput by processing multiple a larger chunk of the workload per unit of time. The scheduling of the partitions for execution on the processing elements is controlled by a dispatch unit inside the main controller module. The dispatch unit fetches partitions data from the LSU and pass down each partition in a first come first serve fashion to the processing elements.

4.1 DCSC Matrix partitioning

To enable the parallelization of FlexGraph tasks, the DCSC sparse matrix is first partitioned into aligned partitions of 32 consecutive rows containing non-zero edges. we choose a partition size of 32 mainly to match the 32-bit size of the bitmasks encoding the vertices that are active. These masks are used by the software on the host processor to determine the regions of the acceleration's output buffer that have been updated. Figure 10 illustrates sample matrix partitioning in which the non-zero horizontal regions have been broken into

	c0	c1		c2		c3
P0	x			x		
	x x					
P1	xxx	x		xxx		
	x xx	xxx				xx

Figure 10: DCSC Matrix Partitioning

two partitions P0 and P1. The non-zero column ranges *(c0, c1, c2, c3)* represent the selected chunks covered by each partition. Partition P0 will only contain ranges *c0* and *c2* while P2 has non-zeros in all four ranges. The partitioning scheme is not ideal because of the workload in-balance that might exist when the number of non-zero varies disproportionally between partitions.

4.2 Synchronising Memory Accesses

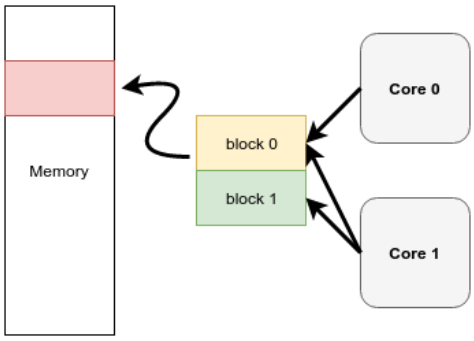

Figure 11: Active Masks Write Synchronization

An implementation challenge we faced when supporting multiple processing elements in FlexGraph was the synchronization of write accesses for the output active masks. Because the LSU data transfer granularity is 64 byte (matching CCI [22]), FlexGraph need to manage shared write accesses to the same block to avoid having a processing element override the content of another one. Luckily this doesn't pose any problem for writing out partition output values because the 32 rows within a partition occupy 32 * 4 bytes = 2 x 64 bytes blocks. Because the partitioning reinforces 32 rows alignment, each processing element can safely write into their assigned blocks. However, writing out the active masks poses a challenge because a single 4-byte mask encodes the active states of all 32 rows in a partition. This causes the processing elements to potentially share a single 64-byte block when writing the masks out to memory. To alleviate the contention, FlexGraph keep N active 64-byte blocks locally assigned to either one of the N processing elements. It assigns an ownership bit mask to each block to track their reference count and the processing element assigned to them. It also tracks the current address value assigned to each block. When a processing element is ready to write its 4-byte active mask, it passes the mask address and value to the LSU. The LSU goes to last active block the processing element previ-

ous wrote to and check if the block address matches. If so, it simply adds the new content to the existing block. If there is no match, it clears its ownership bit and flushes the block to memory if the ownership mask goes to zero. Then it looks up the other active block if anyone already has the address to use it, otherwise if it acquires the last unused block. Figure 11 shows a simplified illustration of the scheme with two processing elements. We don't need to keep more than N blocks in local storage for this scheme to work because the block address references are always incremental and never regress, meaning that there is always going to be free block available to use. It is important to also point out that for this scheme to work, the operation has to be atomic. FlexGraph has a single communication channel between all processing elements and the LSU which does some round robin arbitration and blocks the write mask request until it is committed. We also investigated an alternative solution to avoid this synchronization, which is simply increasing the partition size to 16 * 32 rows, allowing the aggregated active masks to occupy a full 64-byte block. They were two major issues with this approach. Firstly, the total size of all resident partitions will be too large if we support multiple processing elements. Secondly, it will worsen the workload imbalance that is already present with 32 rows.

This section describes some optimizations we added to FlexGraph to improve its performance.

4.3 Optimizing Memory Accesses via Stream Buffers

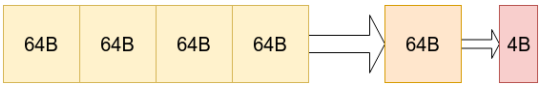

Figure 12: Stream Buffers

As stated before, FlexGraph's LSU transfer granularity is 64 bytes, which is much larger than the 4 bytes of elements accessed by the processing elements. To save on bandwidth, the LSU uses stream buffers that fetch a 64-byte blocks of data but extracts 4-byte elements at the time. It is implemented using a fifo structure in the back-end where 64-byte responses from QPI go into. In the front-end, there is a temporary 64-byte block that is extracted from the fifo on demand to deliver 4-byte element at the time. We employ a shift register to extract the 4-byte element to pass it to the processing element. Figure 12 shows an illustration of the stream buffer concept.

4.4 Caching Vertex Values and Masks

Another optimization we implemented in FlexGraph is the caching of vertex values and masks. Looking at the SPMV speudo-code in listing 1, we can observe in lines 5 and 6 that there is a random memory access to buffers *x_values* and *x_activemask*. FlexGraph employs a stream buffer like concept to consume 64-byte vertex data once they arrive inside the processing element to each iteration loop. However, the same block can be referenced by another processing element and sharing them inside the cache structure can save unnecessary memory traffic. We implemented two small fully associative caches with first-in-First-out replacement to hold active vertex data during execution. The caches are implemented inside the LSU which is responsible for managing them. When a vertex request arrives from a processing element, the LSU first looks up the cache if the block is already present and return it. If the block is not there, it sends the request to QPI. QPI output interface support a 14-bit metadata field that is used to identify the block once it is returned. We use that field to store the index of the block such that we can compute the cache tag when it arrives. Because the cache is small, we implemented a one cycle tag lookup for the LSU to known if the block is present and accessible.

4.5 Executing Non-blocking Memory writes

FlexGraph memory writes are all non-blocking, this applies to both the output values and active masks. This allows the processing elements to push their write request and resume execution while the LSU is processing them. Upon write responses from the QPI channel, the LSU internally keeps track of the count of outstanding requested to ensure that all writes have completed. The QPI interface exposes two write response ports by which the memory replies could be sent. This allows servicing two responses simultaneously. The LSU implements a mechanism to process both channels simultaneously when they are active. At completion time when all processing elements are done executing the current run, the controller waits for all outstanding write requests to complete before asserting the *done* signal.

5. FLEXGRAPH SOFTWARE-HARDWARE CODESIGN

6. EXPERIMENTAL SETUP

In this work, we simulated our evaluation environment entirely in software using the Cocoh [29] C++ framework. Figure 13 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.

6.1 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 14 shows some configuration parameters we use for both the simulation and accelerator modelling.

6.2 Graph Datasets

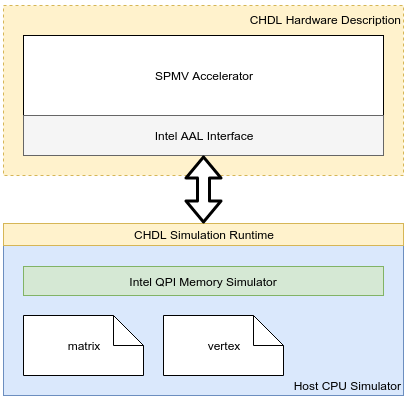

Figure 13: FlexGraph Simulation Environment

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 14: FlexGraph Simulation Parameters

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 15 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 15: Graph500 dataset

7. RESULTS ANALYSIS

In this section, we discuss our evaluation results.

7.1 Throughput Performance

Figure 16 shows the throughput performance of the accelerator in GFlops. We estimated the performance using the following equation:

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

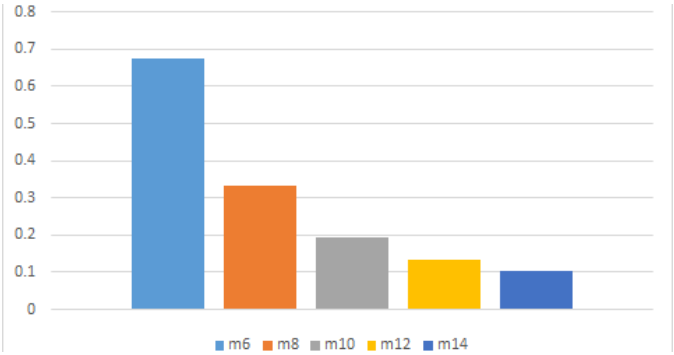

Figure 16: FlexGraph Throughput Performance

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.

7.2 The Impact of Parallelism

Figure 17 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.

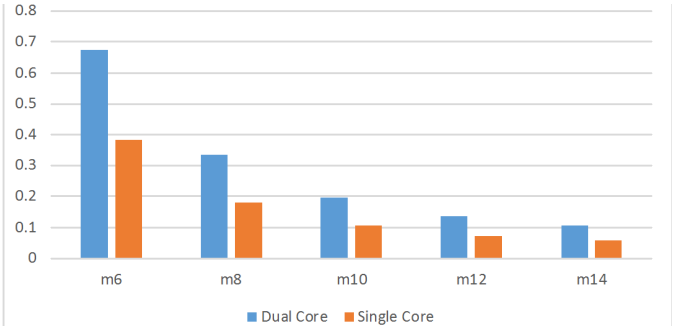

Figure 17: FlexGraph Single Core Performance

7.3 The Impact of Memory Optimizations

Figure 18 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.

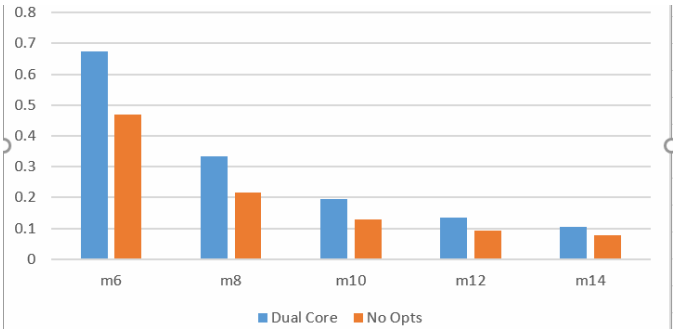

Figure 18: FlexGraph Performance with memory optimizations

7.4 Summary Discussion

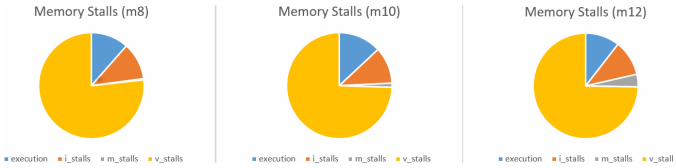

Figure 19: FlexGraph Memory Stalls

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

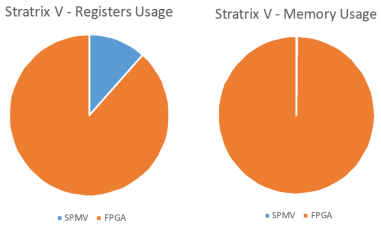

Figure 20: Cocoh's hardware cost evaluation

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 implementation 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 19 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 17 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 20) 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.

8. RELATED WORK

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

8.1 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.

8.2 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.

8.3 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.

9. 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.

10. 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.

11. 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.

12. 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.