Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics

Tae Jun Ham†  Lisa Wu‡  Narayanan Sundaram†  Nadathur Satish‡  Margaret Martonosi∗
‡Princeton University  †University of California, Berkeley  ∗Parallel Computing Lab, Intel Corporation

{tae, mrm}@princeton.edu  lisakwu@berkeley.edu  {narayanan.sundaram, nadathur.rajagopalan.satish}@intel.com

Abstract—Graphs are one of the key data structures for many real-world computing applications and the importance of graph analytics is ever-growing. While existing software graph processing frameworks improve programmability of graph analytics, underlying general purpose processors still limit the performance and energy efficiency of graph analytics. We architect a domain-specific accelerator, Graphicionado, for high-performance, energy-efficient processing of graph analytics workloads. For efficient graph analytics processing, Graphicionado exploits not only data structure-centric datapath specialization, but also memory subsystem specialization, all the while taking advantage of the parallelism inherent in this domain. Graphicionado augments the vertex programming paradigm, allowing different graph analytics applications to be mapped to the same accelerator framework, while maintaining flexibility through a small set of reconfigurable blocks. This paper describes Graphicionado pipeline design choices in detail and gives insights on how Graphicionado combats application execution inefficiencies on general-purpose CPUs. Our results show that Graphicionado achieves a $1.76 \times - 6.54 \times$ speedup while consuming $50 - 100 \times$ less energy compared to a state-of-the-art software graph analytics processing framework executing 32 threads on a 16-core Haswell Xeon processor.

I. INTRODUCTION

Starting out as a recreational mathematical puzzle known as the Königsberg bridge problem [9] in the early 18th century, graph theory and topology have since developed into well-known representations of many-to-many relationships, solving research problems in the areas of network communications, social informatics, natural language processing, system biology, and cyber security. In the era of producing and consuming “big data”, there has been a resurgence of interest in developing graph analytics applications to gain new solutions and insights; examples include Google’s citation ranking algorithm, PageRank [30], Facebook’s semantic search engine, Graph Search [44], Ayasdi’s topological data analysis engine that led scientists to more effective breast cancer treatments [15], and MITRE’s cyber warfare analytics management software, CyGraph [48], that correlates intrusion alerts to known vulnerabilities. This renewed interest spurred the software community to develop more efficient graph analytics processing frameworks [4, 5, 23, 25, 33, 35, 46] as well as the hardware community to create hardware that can execute graph analytics applications with more efficiency than what the off-the-shelf, general-purpose processors and systems can provide [11]. To that end, our research focuses on exploiting the structured data movements and computation patterns that graph analytics applications exhibit to improve efficiency, and on mitigating the challenges such applications face when executing on traditional CPUs.

Certain key characteristics must be accounted for when considering graph domain accelerators. First, graph analytics applications are often memory latency- or bandwidth-bound. For example, graph traversals often require many memory accesses relative to only small amounts of computation. Unfortunately, current general purpose processors are not the ideal platform for executing such applications. Their inefficiencies include 1) waste of off-chip memory bandwidth from inefficient memory access granularity—loading and storing 64-byte cacheline data while operating on only a portion of the data (e.g., 4 bytes), 2) ineffective on-chip memory usage—hardware cache is oblivious to graph-specific datatypes and does not effectively retain high locality data on-chip, and 3) mismatch in execution granularity—computation and communication of data using x86 instructions instead of utilizing domain-specific datatypes for graph analytics such as edges and vertices. To overcome these limitations, we propose a set of datatype and memory subsystem specializations in addition to exploiting the inherent parallelism of graph workloads to alleviate these performance bottlenecks.

We architect, design, and evaluate a domain-specific accelerator for processing graph analytics workloads. Graphicionado features a pipeline that is inspired by the vertex programming paradigm coupled with a few reconfigurable blocks; this specialized-while-flexible pipeline means that graph analytics applications (written as a vertex program) will execute well on Graphicionado.

This paper makes the following contributions:

• A specialized graph analytics processing hardware pipeline that employs datatype and memory subsystem specializations while offering workload-specific reconfigurable blocks called Graphicionado.

• An in-depth tour of the various microarchitecture optimizations to provide performance and energy efficiency, techniques to extract more parallelism, and tactics to support large-scale real-world graphs using slicing and partitioning.
A. Software graph processing frameworks

Software graph processing frameworks typically aim to provide three things to users—ease of programming, improved performance, and efficient scalability that allows the workloads to scale up and out. In an effort to improve programmability of graph algorithms, different programming models have been proposed. Examples include vertex programming [4, 33, 35, 46], sparse matrix operations [8], graph domain-specific languages [18], and task-based models [23]. In addition, the software frameworks vary widely in their performance and scalability as well, especially when compared to native applications [43].

Of the various programming interfaces for graph frameworks, the most popular is vertex programming. In this model, the entire algorithm can be expressed as operations on a single vertex and its edges. This programming model is generally considered easy to use and not overly restrictive, but implementation performance can vary significantly in practice. GraphMat [46] has been shown to have the best performance amongst many different software graph frameworks on a single node. While the exact programming APIs for each vertex-programming-based graph framework differ slightly, they all have similar structure.

Fig. 1 shows a vertex programming example using three essential operators—Process_Edge, Reduce and Apply. In a vertex program, each vertex has an application-specific vertex property that is updated iteratively. At every iteration, each vertex is inspected to check if there are incoming edges from active vertices (i.e. vertices whose states were updated in the last iteration). Then, for all incoming edges from active vertices, the corresponding vertex processes each edge separately using Process_Edge and performs reduction using Reduce to a form single value. Lastly, the reduced value, its vertex property, and a constant vertex property are used to update the current state of the vertex using Apply. Vertices whose properties have changed become active for the next iteration and iterations proceed until there are no more active vertices or until a maximum number of iterations is specified. A few optional parameters are user-controllable—for example, whether all vertices are considered active in all iterations or not, and what the convergence threshold is. This model helps specify a wide variety of useful graph algorithms [43].

\[ V_{score} = \alpha + (1 - \alpha) \cdot \sum_{U \in U \text{ or } V \in V} \frac{U_{score}}{U_{deg}} \]

B. Algorithms

Throughout the paper, we discuss four different fundamental graph algorithms which are representative from applications including machine learning, graph traversal, and graph statistics. These are core kernels and building blocks comprising the bulk of graph processing runtime in many applications. This section provides a brief introduction to these algorithms and shows how each algorithm maps to the programming model.

PageRank (PR) The PageRank algorithm calculates scores of vertices in a graph based on some metric (e.g., popularity). Web pages are represented as vertices and hyperlinks are represented as edges. The equation below shows how the PageRank score is calculated for each vertex. \( \alpha \) is a constant and \( U_{deg} \) is the out-degree and a constant property of vertex \( U \). In PageRank, all vertices are considered active in all iterations.

\[ V_{dist} = \min_{U \text{ or } V \in V} \left( V_{dist} + E_{weight}(U, V) \right) \]
Amplifier [21]. The results are shown in Fig. 2.

4- or 8-byte accesses. These algorithms become bandwidth limited because an entire cacheline is fetched but only a small portion is used, resulting in bandwidth waste. PageRank’s off-chip memory usage is much closer to optimal since it performs much fewer random memory accesses. However, in such cases, memory latency or computation throughput often become performance bottlenecks. On a small input graph such as Flickr, GraphMat can perform less off-chip accesses than optimal because the entire graph is small enough to fit within the 40MB LLC, and the data stays on-chip across iterations unlike the optimal communication case.

Software graph processing frameworks typically have another inefficiency—they must execute many instructions just to move data around and supply data for the custom computations which define a target algorithm. As Fig. 3 shows, for three out of the four algorithms, fewer than 6% of the executed instructions are for custom computations (i.e., Process_Edge, Reduce, Apply). In other words, more than 94% of the instructions executed are responsible for traversing the graphs (i.e., finding the relevant edge of a vertex, etc.) and loading the required arguments (e.g., vertex property, edge weight) for custom computations. These instruction overheads often result in low energy efficiency.

C. Software framework inefficiencies

It was shown previously in [43] that most platform-optimized native graph algorithms are memory bandwidth-limited. In order to identify the inefficiencies of off-chip memory bandwidth usage, we measured GraphMat [46] bandwidth consumption on an Intel Xeon server instrumented using Intel VTune Amplifier [21]. The results are shown in Fig. 2.

Here, the efficiency of the off-chip communication is defined as the ratio of off-chip memory traffic normalized to the optimal communication case — the amount of off-chip memory accesses in an imaginary device which has enough on-die storage to reuse all data for each iteration but not across iterations. GraphMat performs many more off-chip memory accesses than the optimal case for BFS and SSSP, because GraphMat performs cacheline-granular off-chip random accesses and these algorithms use many non-local 4- or 8-byte accesses. These algorithms become bandwidth limited because an entire cacheline is fetched but only a small portion is used, resulting in bandwidth waste. PageRank’s off-chip memory usage is much closer to optimal since it

Mapping algorithms to the programming model There can be multiple ways to map an algorithm to the vertex programming model specified here. An example mapping for the algorithms discussed above is shown in Table I. Note that the $V_{prop}$ parameter for Process_Edge is an optional parameter. In fact, many of the graph algorithms either do not need this parameter or can be expressed without this parameter. Nevertheless, GraphMat supports $V_{prop}$ since it improves the programmability in some algorithms such as Collaborative Filtering.

C. Software framework inefficiencies

It was shown previously in [43] that most platform-optimized native graph algorithms are memory bandwidth-limited. In order to identify the inefficiencies of off-chip memory bandwidth usage, we measured GraphMat [46] bandwidth consumption on an Intel Xeon server instrumented using Intel VTune Amplifier [21]. The results are shown in Fig. 2.

Here, the efficiency of the off-chip communication is defined as the ratio of off-chip memory traffic normalized to the optimal communication case — the amount of off-chip memory accesses in an imaginary device which has enough on-die storage to reuse all data for each iteration but not across iterations. GraphMat performs many more off-chip memory accesses than the optimal case for BFS and SSSP, because GraphMat performs cacheline-granular off-chip random accesses and these algorithms use many non-local 4- or 8-byte accesses. These algorithms become bandwidth limited because an entire cacheline is fetched but only a small portion is used, resulting in bandwidth waste. PageRank’s off-chip memory usage is much closer to optimal since it

Mapping algorithms to the programming model There can be multiple ways to map an algorithm to the vertex programming model specified here. An example mapping for the algorithms discussed above is shown in Table I. Note that the $V_{prop}$ parameter for Process_Edge is an optional parameter. In fact, many of the graph algorithms either do not need this parameter or can be expressed without this parameter. Nevertheless, GraphMat supports $V_{prop}$ since it improves the programmability in some algorithms such as Collaborative Filtering.

C. Software framework inefficiencies

It was shown previously in [43] that most platform-optimized native graph algorithms are memory bandwidth-limited. In order to identify the inefficiencies of off-chip memory bandwidth usage, we measured GraphMat [46] bandwidth consumption on an Intel Xeon server instrumented using Intel VTune Amplifier [21]. The results are shown in Fig. 2.

Here, the efficiency of the off-chip communication is defined as the ratio of off-chip memory traffic normalized to the optimal communication case — the amount of off-chip memory accesses in an imaginary device which has enough on-die storage to reuse all data for each iteration but not across iterations. GraphMat performs many more off-chip memory accesses than the optimal case for BFS and SSSP, because GraphMat performs cacheline-granular off-chip random accesses and these algorithms use many non-local 4- or 8-byte accesses. These algorithms become bandwidth limited because an entire cacheline is fetched but only a small portion is used, resulting in bandwidth waste. PageRank’s off-chip memory usage is much closer to optimal since it

Mapping algorithms to the programming model There can be multiple ways to map an algorithm to the vertex programming model specified here. An example mapping for the algorithms discussed above is shown in Table I. Note that the $V_{prop}$ parameter for Process_Edge is an optional parameter. In fact, many of the graph algorithms either do not need this parameter or can be expressed without this parameter. Nevertheless, GraphMat supports $V_{prop}$ since it improves the programmability in some algorithms such as Collaborative Filtering.

C. Software framework inefficiencies

It was shown previously in [43] that most platform-optimized native graph algorithms are memory bandwidth-limited. In order to identify the inefficiencies of off-chip memory bandwidth usage, we measured GraphMat [46] bandwidth consumption on an Intel Xeon server instrumented using Intel VTune Amplifier [21]. The results are shown in Fig. 2.

Here, the efficiency of the off-chip communication is defined as the ratio of off-chip memory traffic normalized to the optimal communication case — the amount of off-chip memory accesses in an imaginary device which has enough on-die storage to reuse all data for each iteration but not across iterations. GraphMat performs many more off-chip memory accesses than the optimal case for BFS and SSSP, because GraphMat performs cacheline-granular off-chip random accesses and these algorithms use many non-local 4- or 8-byte accesses. These algorithms become bandwidth limited because an entire cacheline is fetched but only a small portion is used, resulting in bandwidth waste. PageRank’s off-chip memory usage is much closer to optimal since it

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Process_Edge ($E_{weight}, U_{prop}, [Optional]V_{prop}$)</th>
<th>Reduce ($V_{temp}, Res$)</th>
<th>Apply ($V_{temp}, V_{prop}, V_{const}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>$U_{prop}$</td>
<td>$V_{temp} + \text{Res}$</td>
<td>$(\alpha + (1 - \alpha)V_{temp})/V_{deg}$</td>
</tr>
<tr>
<td>BFS</td>
<td>N/A</td>
<td>$\min(V_{temp}, \text{IterCount})$</td>
<td>$V_{temp}$</td>
</tr>
<tr>
<td>SSSP</td>
<td>$U_{prop} + E_{weight}$</td>
<td>$\min(V_{temp}, \text{Res})$</td>
<td>$\min(V_{temp}, V_{prop})$</td>
</tr>
<tr>
<td>CF</td>
<td>$(E_{weight}(U, V) - V_{prop} \cdot U_{prop})U_{prop} - \lambda \cdot V_{prop}$</td>
<td>$V_{temp} + \text{Res}$</td>
<td>$V_{prop} + \gamma \cdot V_{temp}$</td>
</tr>
</tbody>
</table>

Table I: Example mapping of algorithms to programming model. For an edge $E = (U, V)$, $U$ is the source vertex and $V$ is the destination vertex.

Fig. 2: Off-chip communication efficiency of GraphMat.

Fig. 3: Percentage of executed instructions for custom computation.
Graphicionado is a hardware accelerator specialized for processing graph analytics algorithms. For better programmability and flexibility, Graphicionado inherits the advantage of software graph processing framework GraphMat. As in software graph processing frameworks, Graphicionado allows users to express specific graph algorithms by defining three computations (Process_Edge, Reduce, and Apply). In addition, it transparently handles all necessary data movement and communication on-chip and off-chip to support those operations. Graphicionado overcomes the limitations of software graph processing frameworks by applying specializations on the compute pipeline and the memory subsystem.

### A. Graph processing model

#### Graphicionado Base Graph Processing Model

<table>
<thead>
<tr>
<th>Pseudocode for Graphicionado processing and apply phases.</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image_url" alt="Pseudocode for Graphicionado processing and apply phases." /></td>
</tr>
</tbody>
</table>

*Fig. 4: Pseudocode for Graphicionado processing and apply phases.*

Graphicionado takes an input graph in the coordinate format. In this format, a graph is represented as a list of vertices where each vertex $v$ is associated with a vertex property $vProp$, and each edge $e$ is associated with a 3-tuple $(srcid, dstid, weight)$ indexed by the edge id $eid$. This edge list is sorted according to the srcid and then the dstid. Before the input graph can be fed into the Graphicionado processing pipeline, some preprocessing of the input graph is done: an EdgeIDTable is constructed and stored in memory. This array stores the $eid$ of the first edge of each vertex to allow streaming accesses of the edges starting at a particular vertex. Graphicionado also uses memory to store the vertex property array $vProp$, the temporary vertex property array $vTempProperty$, and the constant vertex property array $vConst$ associated with all vertices.

#### Processing Phase

In this phase, all outgoing edges from every active vertex are examined and the necessary user-defined computations, Process_Edge and Reduce, and updates to the associated vertex properties are calculated and stored into the temporary vertex property array $vTempProperty$. This phase is only terminated when all active vertices are processed. For some graph analytics workloads such as Collaborative Filtering, not only does the property associated with the source vertex need to be read and manipulated, but also the property associated with the destination vertex, as shown in the pseudocode (line 6).

#### Apply Phase

In this phase, properties associated with all vertices are updated using the user-defined Apply function. Apply uses input values from the vertex property array $vProp$, the constant vertex property array $vConst$ stored in memory and the temporary vertex property array $vTempProperty$ computed from the processing phase to make necessary updates to the vertex property array. The $ActiveVertex$ array keeps track of which active vertices changed their property values in this phase.

#### B. Hardware implementation

In this section we describe the microarchitecture of the hardware blocks that implement the Graphicionado graph processing model. Each module corresponds to one or more lines of pseudocode in Fig. 4 and their physical characteristics are obtained using the implementation methodology described in Section VII. Fig. 5 shows a base Graphicionado pipeline that is constructed with the modules along with small hardware queues (4-8 entry FIFOs) between the modules for communication and connection.

#### Graphicionado Modules

The **Sequential Vertex Read** performs sequential memory read accesses given a starting address. It buffers one cacheline worth of data and passes the vertex properties one at a time to the output queue for the consumption of the next pipe stage. This module is used in stage P1 of the Processing phase, and A1 and A2 of the Apply phase to read $vProp$ and $vTempProperty$.

The **Sequential Vertex Write** performs sequential memory writes given a starting address. It takes the data to be stored from the input queue and writes it to its internal buffer. When its internal buffer has a cacheline worth of data, it performs a write request to memory. This module is used to store the updated $vProp$ in stage A4 and $ActiveVertex$ in stage A5 during the Apply phase.

The **Random Vertex Read/Write** performs random reads and writes given a vertex id. It is used to read the destination $vProp$ in stage P4, read and write $vTempProperty$ in stage P7 and P9.

The **EdgeID Read** performs a read from the preconstructed EdgeIDTable given a vertex id and outputs the $eid$ of the first edge associated with the input vertex id. Implementations of this module are different for the base and the optimized Graphicionado pipeline. Section IV-A presents an optimized implementation.
The \textit{Edge Read} performs random and sequential reads of the edge data given an \textit{eid}. The initial access is random but subsequent accesses are sequential. All edge data streamed in are examined for their \textit{srcid}. The module continues fetching edge data from the memory as long as the \textit{srcid} of the fetched edges matches the \textit{srcid} of the edges being processed.

The \textit{Atomic Update} performs the update of the destination \textit{VProperty} in three steps. First, the module reads the \textit{VProperty} in stage P7, then it performs a Reduce computation in stage P8, and finally it writes the modified \textit{VProperty} back to memory in stage P9. Since this is a read-modify-write update, the process needs to be atomic for the same vertex (i.e. the same vertex cannot be in more than one of these pipeline stages at the same time). This hardware module enforces the atomicity in stage P6 by stalling the pipeline and blocking the issue of the vertex id being processed when violation of the condition is detected. We use a small 4-way associative CAM-like structure to enforce such atomicity (16 4-byte entries).

\textbf{Custom Modules} As previously described, Graphicionado uses three user-defined custom computations – \textit{Process}, \textit{Reduce}, and \textit{Apply} – which express the target graph algorithm. There are several options to implement these custom computations: 1) fully reconfigurable fabric (e.g., FPGA on-chip/on-package [26]), 2) programmable ALUs, or 3) fully custom implementations on chip for a set of algorithms. A user can choose among these options depending on his/her needs. This paper uses the third option for its evaluations; we construct the custom functions using single-precision floating point adders, multipliers, and comparators.

\section{IV. Graphicionado Optimizations}

Section III presents a base implementation of the Graphicionado pipeline. For this pipeline to work effectively, however, it is important to match the throughput of each pipeline stage to achieve maximum efficiency. This section explores the potential bottlenecks of the base implementation and presents optimizations and extensions to relieve the shortcomings.

\subsection{A. Utilizing on-chip scratchpad memory}

\textbf{Improving Destination Vertex Update} One of the most significant bottlenecks in the Graphicionado pipeline is the destination vertex update (Fig. 5 stages P6–P9). It performs poorly because random vertex reads and writes from and to memory are slow. Further, vertex properties need updating in many graph algorithms are less than 8 bytes and performing cacheline-granular memory reads and writes ends up wasting off-chip memory bandwidth. Still further, the destination vertex updates need to be atomic. Long-latency random accesses to memory can potentially stall the pipeline for a long time if the next destination to be updated is the same one that is currently being updated. Graphicionado optimizes these random vertex reads and writes by utilizing a specialized large on-chip embedded DRAM (eDRAM) scratchpad memory (Fig. 6 stages P6–P9). This on-chip storage houses the \textit{VTempProperty} array and significantly improves the throughput of random vertex reads and writes, eliminates bandwidth waste from cacheline-granular off-chip memory accesses by reading and writing on-chip, and lowers the pipeline stall time by minimizing the latency of \textit{VTempProperty} accesses.

\textbf{Improving Edge ID Access} Another potential performance bottleneck in the pipeline is reading the \textit{EdgeIDTable} (Fig. 5 stage P2). In this stage, an \textit{eid} of the first outgoing edge of a given source vertex is read from memory. This is another random memory access that stalls the pipeline for a long-latency access and wasting the off-chip memory bandwidth as the \textit{eid} is only 4 bytes. As in the case with reading the destination vertex properties, Graphicionado places this \textit{EdgeIDTable} data in the on-chip scratchpad memory as well.

Storing \textit{VTempProperty} and \textit{EdgeIDTable} on-chip removes almost all random memory accesses in the pipeline. In the case where the on-chip scratchpad memory size is not large enough, an effective partitioning scheme is employed and described in Section VI. The partitioning scheme allows only portions of such data to be stored on-chip while minimizing the performance impact.

\subsection{B. Optimization for edge access pattern}

While some graph algorithms (e.g. BFS and SSSP) operate with \textit{frontiers of active vertices}, there are many algorithms that simply treat all vertices as active, such as PageRank and CF. In PageRank and CF, instead of accessing a portion of the edges through the \textit{EdgeIDTable}, all edges in a graph are accessed and processed for every iteration; we call these algorithms \textit{complete edge access algorithms}. For \textit{complete edge access algorithms}, stage P3 in Fig. 6 performs sequential reads from the beginning of the edge list. This optimization along with utilizing the on-chip scratchpad memory removes any random off-chip memory accesses since all vertices are considered active and the optional destination vertex property in stage P4 is not used. A similar optimization is done in the \textit{Apply}
phase where stage A5 in Fig. 5 is removed and the resulting optimized pipeline is shown in Fig. 6.

C. Prefetching

With the optimizations described above, most of the off-chip memory accesses in Graphicionado are now sequential accesses. Since the addresses of these off-chip memory accesses are not dependent on any other portions of the pipeline, we can easily perform next-line prefetches and get the data into the accelerator before they are needed. We extend the *sequential vertex read* and *edge read* modules (stage P1 and P3 in Fig. 6) to prefetch and buffer up to N cachelines ($N = 4$ is used for our evaluation) and configure them to continue fetching the next cacheline from memory as long as the buffer is not full. With this optimization, almost all of the sequential memory access latencies can be hidden and Graphicionado can operate at a high throughput.

D. Optimization for symmetric graphs

Most graph processing frameworks (including GraphMat) work naturally with directed graphs. In such frameworks, an undirected input graph is effectively treated as a symmetric graph. That is, for each edge $(srcid, dstid, weight)$ there exists an edge $(dstid, srcid, weight)$. While this approach works, it incurs unnecessary memory accesses for complete edge access algorithms such as Collaborative Filtering. For example, in order to process an edge $(u, v, weight)$, the source vertex property $VProperty[u]$, the destination vertex property $VProperty[v]$, and the edge data $e = (u, v, weight)$ are read and $VProperty[v]$ updated at the end of the *processing* phase. The exact same set of data will be read again later when processing the symmetric edge $(v, u, weight)$ and $VProperty[u]$ is updated this time. To reduce bandwidth consumption, Graphicionado extends its pipeline so that it can update both the source and the destination vertex properties when processing an edge from a symmetric graph without having to read the same data twice. This is reflected in the optimized pipeline shown in Fig. 6 stages P5–P9 where this portion of the pipeline is replicated.

E. Large vertex property support

The current Graphicionado pipeline is designed to support processing up to 32 bytes of vertex property data per cycle. When a large vertex property is desired, for example, Collaborative Filtering implements vertex properties of 128 bytes each, the large vertex property is simply treated as a packet involving multiple flits where each flit contains 32 bytes. For most of the pipeline stages in Graphicionado, each flit is processed without waiting for an entire packet worth of data to arrive (in a manner similar to wormhole switching). For the custom computation stages, we wait for the entirety of the packet data to arrive using a simple buffering scheme before computations are performed (as in store-and-forward switching) to maintain functionality. With proper buffering (4 flits in the case for CF), the throughput of the pipeline is not significantly impacted.

V. GRAPHICIONADO PARALLELIZATION

With optimizations described in Section IV, Graphicionado can process graph analytics workloads with reasonable efficiency. However, thus far it is a single pipeline with theoretical maximum throughput limited to one edge per cycle in the *Processing* phase and one vertex per cycle in the *Apply* phase. This section discusses further improving Graphicionado pipeline throughput, by exploiting the inherent parallelism in graph workloads.

A. Extension to multiple streams

A naïve way to provide parallelism in the Graphicionado pipeline is to replicate the whole pipeline and let each of the replicated pipelines, or pipeline stream, to process a portion of the active vertices. In fact, this is the most common approach in software graph processing frameworks when increasing parallelism. Unfortunately this approach introduces some significant drawbacks in the hardware pipeline. When multiple replicated streams try to read and write the same on-chip scratchpad location, these operations are serialized and performance degrades. To avoid these access conflicts, Graphicionado divides the *Processing* phase into two portions, a source-oriented portion and a destination-oriented portion, corresponding to stages P1–P3 and stages P4–P9 in Fig. 5. The two portions are then replicated separately and connected using a crossbar switch as shown in Fig. 7. Each parallel stream in the source-oriented portion of the pipeline is responsible for executing a subset of the source vertices and each parallel stream in the destination-oriented portion of the pipeline is responsible for executing a subset of the destination vertices. The crossbar switch routes edge data by matching the destination vertex id of the edge. To maximize the throughput of the switch, standard techniques such as virtual output queues [47] are implemented.
Apply phase pipeline which is parallelized in a similar manner.

Graphicionado replicates the number of vertices [16]. This makes fetching multiple edges not need random off-chip memory accesses, this stage is still complete edge access algorithms because it needs to perform occasional random off-chip memory accesses.

The Source-oriented portion of the pipeline reads pre-partitioned source VProperty and associated edges from memory; partitions are done using the source vertex id’s last \( \log_2(n) \) bits for \( n \) streams. Similarly, the destination-oriented portion of the pipeline reads pre-partitioned destination VProperty from memory and reads and writes pre-partitioned destination VTempProperty from the on-chip scratchpad as shown in Fig. 8. Partitions are done using the destination vertex id’s last \( \log_2(m) \) bits for \( m \) streams.

This parallelization of the source and destination streams eliminates memory access conflicts as each stream is strictly limited to only access the memory and scratchpad memory regions exclusively assigned. Another benefit of this parallelization technique is that it simplifies the on-chip scratchpad memory design. We implement \( m \) instances of dual-ported eDRAMs for Graphicionado as opposed to using a single large eDRAM with \( 2m \) ports.

B. Edge access parallelization

For this parallelized Graphicionado pipeline, Read Edges (stage P3) is likely to be one of the performance bottlenecks because it needs to perform occasional random off-chip memory accesses. Even for complete edge access algorithms which do not need random off-chip memory accesses, this stage is still likely to be the performance bottleneck as it can only process one edge per cycle, while previous stages Read Source Property and Read EdgeIDTable can process one vertex per cycle. Real-world input graphs tend to have a large discrepancy between the number of vertices and the number of edges with the number of edges being an order or orders of magnitude larger than the number of vertices [16]. This makes fetching multiple edges per cycle necessary in order to maintain high throughput and balance the pipeline design. Graphicionado replicates the Read Edges unit and parallelizes edge accesses.

We show the different implementations of parallelizing edge accesses for active-vertex based algorithms and complete edge access algorithms in Fig. 9. In the active-vertex based algorithms implementation, an active vertex id is allocated to one of the Read Edges units for edge accesses based on input queue occupancy (lowest occupancy first). Edges loaded from memory are then arbitrated and passed onto the crossbar switch. In the complete edge access algorithms implementation, each source vertex id is broadcasted to all of the Read Edges units and the units only access edges with assigned destination vertex id’s as described in Section V-A. The output of each Read Edges is directly connected to the virtual output queue of the crossbar switch.

C. Destination vertex access parallelization

While the use of the optional destination vertex property array is not common (stage P4 in Fig. 5), it becomes a performance bottleneck when it is present because it involves random vertex reads. To alleviate such performance degradation, we implement a parallelization scheme similar to the parallel edge readers (Fig. 9a) described above. The Read DST Property module is replicated and each destination vertex id is allocated to the lowest occupied module to fetch the destination VProperty. Vertex properties loaded from memory are then arbitrated to produce a single output per cycle.

VI. SCALABLE ON-CHIP MEMORY USAGE

As outlined in Section IV-A, Graphicionado utilizes the on-chip memory for two purposes: performing the temporary destination property update (P7-P9) and reading the edge ID table (P3). By storing those data in an on-chip scratchpad, Graphicionado improves the throughput of pipeline stages (P3, P7, and P9), avoids off-chip memory bandwidth wastes, and lowers the pipeline stall time by minimizing the latency of temporary destination vertex property updates. However, in many cases, the scratchpad memory size is not big enough to house all required data. This section explores strategies to effectively utilize the limited on-chip scratchpad memory.

A. Graph slicing

Storing temporary destination vertex property requires \( \text{Number of Vertices} \times \text{Size of a Vertex Property} \) bytes of on-chip scratchpad memory. Assuming a 4-byte vertex property size and a 32MB on-chip scratchpad memory size, graphs with up to 8 million vertices could be supported. To process larger graphs without losing the benefit of on-chip memory usage, Graphicionado slices a graph into several slices and processes a single slice at a time.

The slicing technique works as follows. First, vertices from the input graph are divided into \( N \) partitions based on their vertex id’s. An example graph in Fig. 10 has six vertices and they are partitioned into two partitions: P1 contains vertices...
(1, 2, 3) and $P_2$ contains vertices $(4, 5, 6)$. Then two slices are constructed depending on which partition the destination vertices of the edges fall as shown on the right side of Fig. 10. After the input graph is sliced, Graphicionado processes a slice of the graph for each sub-iteration, i.e. executing Processing phase for slice 1, executing Apply phase for slice 1, executing Processing phase for slice 2, executing Apply phase for slice 2, and repeating for all iterations. Since the slices are partitioned using destination vertex id’s, write-write conflicts are avoided and no edges need to be processed twice. The slicing does incur some overhead as the same source vertex properties could be read more than once and therefore increase the total read bandwidth usage slightly compared to no slicing.

B. Extended graph slicing for symmetric graphs

While the graph slicing is orthogonal to most of the Graphicionado optimizations, it requires extra pipeline extensions when used together with the symmetric graph optimization (Section IV-D). With the symmetric graph optimization, a scratchpad memory needs to house both the vertex property array for the source as well as the destination vertices. In this case, the slicing in Fig. 10 is not effective as its on-chip storage requirement is not reduced. We employ an extended slicing scheme as shown in Fig. 11.

![Graph slicing Example](image)

**Fig. 10: Graph slicing Example.**

![Extended graph slicing scheme](image)

**Fig. 11: Extended graph slicing scheme for symmetric graph optimization.**

Given a symmetric graph as the one shown in Fig. 11, Graphicionado pre-processes the graph and generates a directed graph which only keeps directed edges originating from a larger vertex id to a smaller vertex id.

With the generated directed graph, three slices are constructed: Slice1 = $\{(u, v) \mid u \in P_1, v \in P_1\}$, Slice2 = $\{(u, v) \mid u \in P_2, v \in P_1\}$, and Slice3 = $\{(u, v) \mid u \in P_2, v \in P_2\}$. Unlike the original graph slicing scheme which only focuses on slicing a graph based on dstid’s, this extended scheme considers both srcid and dstid for slicing. With $N$ partitions, this extended slicing scheme generates $N(N+1)/2$ slices.

For Graphicionado to operate with this extended slicing scheme, an extension to the control is necessary as well. First, care should be taken in ensuring that all edges associated with the vertices about to be updated have gone through the Processing phase before the Apply phase is executed. With the extended slicing scheme, unlike the base case, Apply does not happen for every sub-iteration. Instead, it happens selectively for sub-iterations where a vertex partition is processed for the last time. In addition, since each sub-iteration does not fully process all edges associated with a vertex partition, temporary vertex property data on the scratchpad memory should be stored to memory and loaded back to the scratchpad memory between each Processing phase.

Table 12 shows an example control of a four-partition case. Note that this necessary write-back of scratchpad memory data is not dynamically decided. Instead, they are determined a-priori by the number of vertex partitions. Thus, at runtime, Graphicionado simply needs to follow the pre-determined control flow.

C. Coarsening edge ID table

Another use of the on-chip scratchpad is to store the $EdgeIDTable$ which keeps the mapping between vertices and their corresponding starting edge id’s as shown in Fig. 13-(a). When the size of the $EdgeIDTable$ is too large to fit, a coarsened $EdgeIDTable$ is stored as shown in Fig. 13-(b). A coarsened $EdgeIDTable$ stores an EID for every $N$ vertices ($N = 2$ here) instead of storing the EID’s for every vertex. To find edges corresponding to a given vertex id, the Read Edges unit starts from index $\lfloor vid/N \rfloor$ in the coarsened $EdgeIDTable$ where $vid$ is the vertex id and $N$ is the coarsening factor.

![EdgeIDTable and coarsened EdgeIDTable](image)

**Fig. 12: Extended slicing scheme control example for four partitions. Each row represents a sub-iteration and 3rd-6th column shows the set of events taken after the processing phase of each sub-iteration.**

**Fig. 13: EdgeIDTable and coarsened EdgeIDTable.**

When using a coarsened $EdgeIDTable$, extra edge accesses will incur and care needs to be taken in trading off the on-chip storage size and the extra edge accesses. Note that the slicing technique actually reduces the average degree per vertex and therefore reduces the overhead for coarsening the $EdgeIDTable$.

VII. GRAPHICIONADO EVALUATION

A. Evaluation Methodology

**Overall Design** We implemented each pipeline stage of Graphicionado in Chisel [6], generated Verilog, and synthesized
the blocks using a proprietary sub-28nm design library to produce timing, area, and power numbers. We gave the synthesis tools an operating voltage of 0.7V, a target clock cycle of 1ns, and requested medium path of 0.94ns including setup and hold time, putting the Graphicionado design comfortably at 1GHz.

Evaluation Tools For the performance evaluation, a custom cycle-accurate simulator was designed. This simulator models the microarchitecture behavior of each hardware module described in Section III-B. In addition, the performance model implements a detailed cycle-accurate scratchpad memory model. It is also integrated with the open-source DRAM simulator DRAMSim2 [41] to simulate the cycle-accurate behavior of the off-chip memory accesses. For the on-chip scratchpad memory, 32nm eDRAM model of CACTI 6.5 [19] is used to obtain the cycle time, access latency, and dynamic/static energy consumption. Table II shows the system parameters of the evaluated Graphicionado implementation.

Software Framework Evaluation To compare Graphicionado’s performance and energy efficiency with an optimized software implementation, a software graph processing framework GraphMat [46] is evaluated. We chose GraphMat because it has been shown to have the best performance amongst many different software graph processing frameworks and the performance is better or within 20% of the representative native software implementation. We measure the Xeon chip power consumption using National Instrument’s power measurement data acquisition PCI board [37]. Table II shows the system parameters of the evaluated Graphicionado implementation.

Graph Datasets Table III describes the graph datasets used for the evaluation. A mixture of real-world graphs – FR, FB, Wiki, LJ, TW, NF, and synthetic graphs – RMAT, SB are used for the evaluation. For synthetic graphs, the Graph500 RMAT data generator [1] is used to generate the RMAT graph [10] and a bipartite graph generator described in [43] is used to generate a SB graph which has similar edge distributions with the real-world Netflix graph but at a different scale. Amongst these graphs, FR, FB, Wiki, LJ, RMAT, and TW are used to evaluate PageRank, BFS, and SSSP; bipartite graphs NF and SB are used to evaluate CF which requires a bipartite input graph. For the SSSP evaluation on unweighted real-world graphs, random integer weights between 0 and 256 were assigned.

<table>
<thead>
<tr>
<th>Graph</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>Brief Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flickr (FR) [13]</td>
<td>0.82M</td>
<td>9.84M</td>
<td>Flickr Crawl Graph</td>
</tr>
<tr>
<td>Facebook (FB) [51]</td>
<td>2.93M</td>
<td>41.92M</td>
<td>Facebook User Interaction Graph</td>
</tr>
<tr>
<td>Wikipedia (Wiki) [13]</td>
<td>3.56M</td>
<td>84.75M</td>
<td>Wikipedia Link Graph</td>
</tr>
<tr>
<td>LiveJournal (LJ) [13]</td>
<td>4.84M</td>
<td>68.99M</td>
<td>LiveJournal Follower Graph</td>
</tr>
<tr>
<td>RMAT Scale 23 (RMAT) [1]</td>
<td>8.38M</td>
<td>134.22M</td>
<td>Synthetic Graph</td>
</tr>
<tr>
<td>Twitter (TW) [28]</td>
<td>61.57M</td>
<td>1468.36M</td>
<td>Twitter Follower Graph</td>
</tr>
<tr>
<td>Netflix (NF) [7]</td>
<td>408K users, 18K movies</td>
<td>99.07M</td>
<td>Netflix Prize Bipartite Graph</td>
</tr>
<tr>
<td>Synthetic Bipartite (SB) [43]</td>
<td>997K users, 21K items</td>
<td>248.94M</td>
<td>Synthetic Bipartite Graph</td>
</tr>
</tbody>
</table>

TABLE III: Graph datasets used for the evaluation

![Fig. 14: Graphicionado Performance](image1)

B. Graphicionado Results

Overall Performance Fig. 14 shows the normalized Graphicionado speedup with respect to GraphMat performance. Graphicionado’s on-chip storage is sized at 64MB which is not large enough for all workloads to store their VTempProperty arrays and EdgeIDTable. Techniques discussed in Section VI, namely graph slicing and coarsening of the edge table, are used for large graphs as stated in the caption. Graphicionado achieves a 1.7× to 6.5× performance advantage over the software graph processing framework GraphMat running on a 16-core 32-thread Xeon server. Graphicionado’s main source of performance advantage over the software framework is the efficient use of the on-chip scratchpad memory. While the software framework is run on a processor with vast parallelism, the performance is often limited by inefficient memory bandwidth usage. On the other hand, Graphicionado avoids wasting off-chip memory bandwidth by utilizing scratchpad memory and benefits from extra effective bandwidth.

![Fig. 15: Throughput of the Graphicionado and the software graph processing framework(GraphMat)](image2)
**Throughput**  Fig. 15 shows the throughput of the Graphicionado and the software graph processing framework. The y-axis in the figure represents the average number of edges processed every nanosecond (or every cycle for the Graphicionado design since it runs at 1GHz). We make the following observations: PageRank exhibits high and stable throughput at about 4–4.5 edges per nanoseconds when executing on Graphicionado. This is because Graphicionado’s PageRank pipeline does not have any random memory accesses. As long as all the data is prefetched in time, it can theoretically reach the peak throughput of processing 8 edges per cycle, at 1 edge per cycle per stream for a total of 8 streams. However, its throughput is limited by the off-chip memory bandwidth. For active-vertex based algorithms such as BFS and SSSP, there are random memory accesses and thus they achieved lower throughput than PageRank. Amongst all workloads, Wiki’s throughput is particularly low when executing on Graphicionado and using GraphMat. This is because Wiki’s graph structure is narrow and deep and it exhibits a large number of iterations updating very few vertices each iteration. CF has very low edge processing throughput because it has a large vertex property size at 128 bytes; the off-chip memory bandwidth is further bounded by the vertex property accesses in addition to the edge accesses.

**Communication Efficiency**  Fig. 16 shows the efficiency of Graphicionado and the software graph processing framework normalized to optimal communication case.

**Power/Energy**  Fig. 17 shows the energy consumption of the Graphicionado normalized to GraphMat running on a Haswell Xeon server. For this diagram, energy per access and the leakage power of the eDRAM is obtained using the 32nm eDRAM model in CACTI 6.5 [19]. Note that the reported number from CACTI was more conservative than eDRAM power models shown in other literature [50]. For the Graphicionado pipeline design, each hardware unit is conservatively assumed to be at its peak activity. As shown in the diagram, the energy consumption is about 1-2% of the processor energy consumption. The power consumption differs by around 20x (∼150W on a Xeon chip vs. ∼7W on Graphicionado), coupled with the 2-5x runtime difference, resulted in a total of 50x-100x energy difference. In Fig. 16, most of the energy (∼90%) is spent on the eDRAM. The hardware modules themselves are mostly small specialized routing, control, interface to the memory elements units and thus it is natural that they do not consume much energy. Note that Graphicionado’s energy consumption does not include the DRAM controller energy since it could be placed off-die while Xeon’s energy consumption includes its integrated memory controller.

**C. Effects of Graphicionado optimizations**

**Parallelization/Optimization**  This subsection explores the impact of parallelization and optimization on the Graphicionado pipeline. In Fig. 18, the leftmost bar represents the single stream baseline case. This baseline utilizes the on-chip scratchpad memory (Section IV-A) and edge access pattern optimization (Section IV-B) and is denoted as Baseline. From this, the number of streams are doubled for CFG 1 to CFG 3. Then, CFG 4 shows the effects of prefetching while CFG 5 shows the effects of edge and destination vertex property access parallelization. Finally CFG 6 is only shown for CF and it shows the effects of applying symmetric graph optimization. As depicted, parallelizing Graphicionado streams provides near-linear speedups. Enabling data prefetching achieves another 2x speedup. Applying edge access parallelization for active-vertex based algorithms provides an additional 1.2x speedup and the combination of edge access parallelization and destination vertex property parallelization provides another 2x speedup on CF. PageRank, however, does not see any extra speedup.
from $\text{CFG}^4$ to $\text{CFG}^5$ since it is already bounded by memory bandwidth at this point. Lastly, symmetric graph optimization provides another $1.7 \times$ extra speedup for CF by essentially halving the number of edges that need processing.

**Graph Slicing and Edge Table Coarsening** Figure 19 explores the impact of graph slicing and edge table coarsening techniques described in Section VI. In Fig. 19, the leftmost bar represents the case where a scratchpad memory size is enough to house all required data, denoted as Scaling Factor $=1$. The next few bars represent cases where a scratchpad memory size is $1/N$ of the total required size where $N$ is the scaling factor. For a scaling factor of $N$, the input graph is sliced into $N$ slices (or $N(N+1)/2$ for the extended slicing scheme), and the EdgeIDTable is coarsened by a factor of $N$. In general, reducing the scratchpad memory size to one-eighth of the required data size results in around a 30% performance degradation in PageRank, a 35% degradation in BFS, and a 50% degradation in SSSP. Note that even with the 30%–50% performance degradation, Graphicionado still outperforms the software graph processing framework running on a server-grade multiprocessor while consuming less than 2% of the energy. For BFS-Wiki and CF, slicing provided a distribution of the graph where some slices contain either no edges to process or no vertices to update and therefore skip either the Processing or the Apply phase for multiple iterations causing the performance to actually be better than the no-slicing case.

In summary, as depicted in Fig. 19, Graphicionado can support the processing of very large graphs with reasonable performance degradation. In addition, given the recent trend of increasing on-chip storage for processors (e.g., Intel’s i7-5775C [20], IBM’s Power8 [45]), we expect that most of the real-world graph’s intermediate data will fit in a larger on-chip eDRAM with a reasonable scaling factor. With a scaling factor of 16, a 128MB eDRAM can store intermediate data for graphs having up to 512 million vertices which exceeds the number of Internet users in Europe (487 Millions in 2015 [22]).

**VIII. RELATED WORK**

**GPU-based Graph Processing Frameworks** There are a few graph analytics software frameworks and libraries specifically optimized for GPUs; Gunrock [49], MapGraph [14], nvGraph [39], and Enterprise [32] are representative examples. Fair comparisons against GPU-based frameworks are difficult since GPUs often have much larger memory bandwidth than what we provisioned for Graphicionado (68GB/s). Here we present a few datapoints for comparison. Recent work [32] compares the throughput of GPU-based BFS implementations across different frameworks. When run on a Tesla K40c GPU (with 288GB/s memory bandwidth) using the Twitter input graph, Enterprise (a specialized GPU BFS implementation) is able to traverse 4.5 edges/ns, Gunrock 0.7 edges/ns, and MapGraph 0.2 edges/ns. Graphicionado processes 1.1 edges/ns (on BFS-Twitter) with 68GB/s available memory bandwidth. If we scale these GPU results assuming a bandwidth of only 68GB/s, Enterprise can process 1.06 edges/ns, Gunrock 0.17 edges/ns, and MapGraph 0.047 edges/ns. Note that the TDP of the Tesla K40c is 245W while Graphicionado only uses around 7W.

**Hardware-based Graph Analytics Acceleration** Hardware acceleration for graph analytics has been recently analyzed with an emphasis on FPGA-based accelerators. First, there are a few accelerators designed for a specific graph analytics algorithm (e.g., SSSP [53], Belief Propagation [24], and PageRank [36]). They can be very efficient but cannot be utilized for a domain of applications. Recently, a couple of vertex-programming model based graph analytics accelerators were explored [12, 38]. While both share the same goal with Graphicionado, GraphGen [38] focuses on generating an application-specific accelerator for a given vertex program specification rather than providing a single re-usable domain-specific accelerator. On the other hand, FPGP [12] targets a different problem where edges are stored in a device with extremely limited bandwidth (e.g., disk). GraphOps [40] is a concurrent work that provides a set of modular hardware units implemented in FPGA for graph analytics. GraphOps optimizes for graph storage and...
layout to provide efficient use of the off-chip memory while Graphicionado optimizes for graph access patterns utilizing an on-chip scratchpad and eliminating unnecessary off-chip memory accesses for efficiency. Lastly, Tesseract [3] targets the same domain as our work, but explores different technology by implementing specific hardware extensions using the logic layer of a 3D-stacked DRAM.

**Software Graph Processing Frameworks** In addition to the popular software graph processing frameworks described in Section II-A, GraphChi [29], TurboGraph [17], and X-Stream [42] are also similar frameworks utilizing disk-based systems for graph processing. Since these frameworks often focus on optimizing for efficient data locality and access patterns, they are closely related to Graphicionado; however, Graphicionado is a hardware implementation that optimizes for off-chip memory bandwidth consumptions rather than memory-to-disk bandwidth consumptions.

**Domain-specific accelerators** Domain specific accelerators are becoming more popular since application-specific accelerators are prone to obsoleteness. Example domain-specific accelerators in machine learning (PuDianNao [31], Tabla [34]) and databases (Q100 [52], Widx [27]) share the common principle with Graphicionado in that they identify the key characteristics and bottlenecks of applications in a specific domain and try to overcome them with efficient hardware.

**IX. Conclusion**

In this paper, we present a domain-specific accelerator Graphicionado specialized for graph analytics processing. Based on the well-defined, popular vertex programming model used in many software processing frameworks, Graphicionado allows users to process graph analytics in a high-performance, energy-efficient manner while retaining the flexibility and ease of a software graph processing model. The Graphicionado pipeline is carefully designed to overcome inefficiencies in existing general purpose processors by 1) utilizing an on-chip scratchpad memory efficiently, 2) balancing pipeline designs to achieve higher throughput, and 3) achieving high memory level parallelism with minimal cost and complexity. Based on the fact that Graphicionado achieves significantly higher speedup ($1.76 - 6.54 \times$) for the same memory bandwidth while consuming less than 2% of the energy compared to the state-of-the-art software graph analytics framework running on a 16-core Haswell Xeon server, we conclude that Graphicionado could be a viable solution to meet the ever increasing demand for an efficient graph processing platform.

**Acknowledgment**

Tae Jun Ham was supported in part by Samsung Fellowship. This work was supported in part by Parallel Computing Lab, Intel Corporation. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was supported in part by the NSF under the grant CCF-1117147. The authors also wish to thank the anonymous reviewers for their time and valuable feedback.

**References**


