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 $1.76-6.54\times$ 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 vulnerability paths. 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.

†This work was done while the author was working at Intel Corporation

978-1-5090-3508-3/16/$31.00

GraphMat Processing Model

```
For each Vertex V
  For each incoming edge E(U,V) from active vertex U
    Res ← Process_Edge (E_weight, U_prop, [OPTIONAL]V_prop)
    V_temp ← Reduce(V_temp, Res)
  End
End
For each Vertex V,
  V_prop ← Apply(V_temp, V_prop, V_const)
End
```

Fig. 1: Simplified GraphMat processing model. Note that this is slightly different from the original GraphMat [46] in that it integrates `Send_Message` with `Apply`.

II. Background and Motivation

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].

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. α 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.

$$V_{score} = \alpha + (1 - \alpha) \cdot \sum_{U|(U,V)\in E} \frac{U_{score}}{U_{deg}}$$

Breadth-First Search (BFS) BFS is a widely-used graph search algorithm from the Graph500 Benchmark, operating on an unweighted graph [1]. Starting from a given initial vertex, the algorithm iteratively explores neighboring vertices and assigns the distance to each vertex connected to the active vertices of the iteration. The equation below shows how the distance is determined for each vertex adjacent to active vertices at iteration t.

$$V_{dist} = min(V_{dist}, t)$$

Single Source Shortest-Path (SSSP) This is a graph traversal algorithm which computes the distance between a single source and all other vertices in a weighted graph. Similar to BFS, the algorithm iteratively explores neighboring vertices from starting vertices and assigns the distance to each vertex connected to the active vertices of the iteration. The main difference between BFS and SSSP is that SSSP utilizes edge weights to determine the distance while BFS does not. The equation below shows how the distance is determined for each vertex adjacent to active vertices.

$$V_{dist} = \min_{U|(U,V)\in E} (V_{dist}, U_{dist} + E_{weight}(U,V))$$

Collaborative Filtering (CF) CF is a popular machine learning algorithm for recommendation systems [2]. It estimates users' ratings for a given item based on an incomplete set of (user, item) ratings. The assumption of the algorithm is that a (user, item) rating is determined by the match of the latent features between users and items and the goal of algorithm is to calculate the hidden features of each vertex (i.e. user and item). For this purpose, a matrix factorization based on gradient descent is performed. The equation below shows how the feature in each vertex is calculated. V_f is a feature vector whose length is K (note that K=32 is used throughout the paper); λ and γ are constants. Collaborative Filtering runs on an undirected bipartite graph (i.e., a graph whose vertices can be divided into two disjoint sets — in CF, the users set and the items set

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

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.

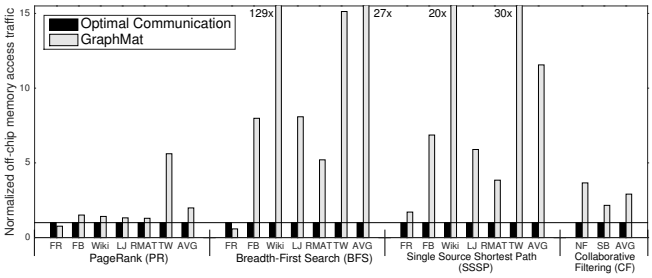

Fig. 2: Off-chip communication efficiency of GraphMat.

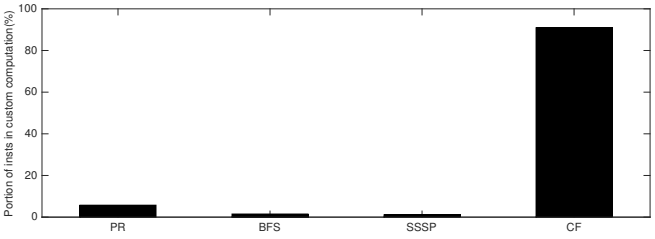

Fig. 3: Percentage of executed instructions for custom computation

are the two disjoint sets) and all vertices are considered active in all iterations.

$$V_f = V_f + \gamma \left[\sum_{U|(U,V)\in E} (E_{weight}(U,V) - V_f \cdot U_f)\, U_f - \lambda V_f \right]$$

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 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.

D. Overcoming inefficiencies

To overcome the software framework inefficiencies, we employed two categories of solutions. First, we applied memory subsystem specialization and architected an on-chip storage as part of the accelerator. The on-chip storage allowed dramatic reduction of the communication latency and bandwidth by converting the frequent and inefficient random off-chip data communication to on-chip, efficient, finer-granularity scratchpad memory accesses. Second, we applied data-structure-specific specialization or datatype specialization and formed datapaths around processing graph analytics primitives, vertices and edges. This further reduces peripheral instruction overheads to prepare the operands for computation. In this work, we will demonstrate that datatype specialization coupled with the programmable and high-performing vertex program paradigm allows the Graphicionado pipeline to not only balance specificity with flexibility but also to deliver exceptional energy efficiency.

III. Graphicionado Overview

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

▷ *Processing Phase*

```
1  for (int i=0; i<ActiveVertexCount; i++) {
2   Vertex src = ActiveVertex[i]; // Sequential Vertex Read
3   int eid = EdgeIDTable[src.id]; // Edge ID Read
4   Edge e = Edges[eid]; // Edge Read
5   while (e.srcid == src.id) {
6     dst.prop = VProperty[e.dstid]; // [Optional] Random Vertex Read
7     VertexProperty res = Process_Edge(e.weight, src.prop, dst.prop);
8     VertexProperty temp = VTempProperty[e.dstid]; // Random Vertex Read
9     temp = Reduce(temp, res);
10    VTempProperty[e.dstid] = temp; // Random Vertex Write
11    e = Edges[++eid] // Edge Read
12  }
13 }
14 // Reset ActiveVertex and ActiveVertexCount
```

▷ *Apply Phase*

```
1  for (int i=0; i<TotalVertexCount; i++) {
2   VertexProperty vprop = VProperty[i]; // Sequential Vertex Read
3   VertexProperty temp = VTempProperty[i]; // Sequential Vertex Read
4   VertexProperty vconst = VConst[i];
5   temp = Apply(vprop, temp, vconst);
6   VProperty[i] = temp; // Sequential Vertex Write
7   if(temp != vprop) {
8     Vertex v;
9     v.id = i;
10    v.prop = temp;
11    ActiveVertex[ActiveVertexCount++] = v; // Sequential Vertex Write
12  }
13 }
```

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

Fig. 4 shows the workflow of Graphicionado in pseudocode. 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 `VProperty`, 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 `VProperty`, 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 `VProperty`, 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 `VProperty` 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 `VProperty` 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 `VProperty` 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.

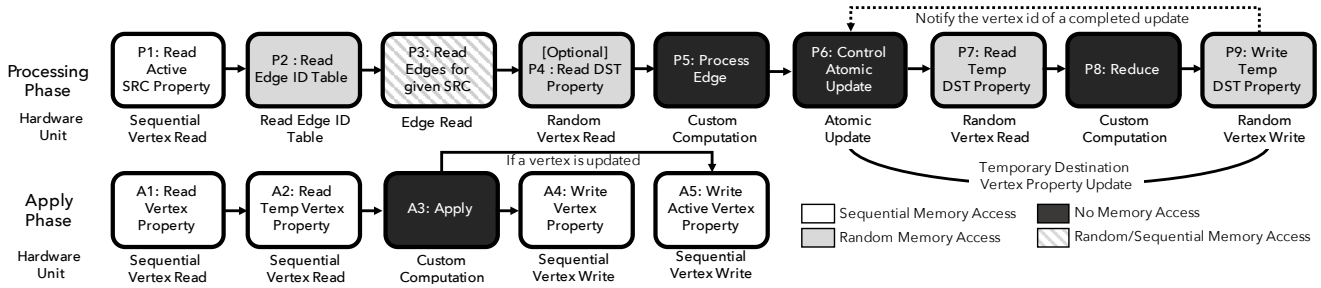

Fig. 5: Graphicionado base pipeline.

The *Edge Read* performs random and sequential reads of the edge data given an `eid`. The initial access is random but subsequent accesses are sequential. All edge data streamed in are examined for their `srcid`. The module continues fetching edge data from the memory as long as the `srcid` of the fetched edges matches the `srcid` of the edges being processed.

The *Atomic Update* performs the update of the destination `VProperty` in three steps. First, the module reads the `VProperty` in stage `P7`, then it performs a `Reduce` computation in stage `P8`, and finally it writes the modified `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).

Custom Modules As previously described, Graphicionado uses three user-defined custom computations – `Process_Edge`, `Reduce`, and `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.

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.

A. Utilizing on-chip scratchpad memory

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 `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 `VTempProperty` accesses.

Improving Edge ID Access Another potential performance bottleneck in the pipeline is reading the `EdgeIDTable` (Fig. 5 stage `P2`). In this stage, an `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 `eid` is only 4 bytes. As in the case with reading the destination vertex properties, Graphicionado places this `EdgeIDTable` data in the on-chip scratchpad memory as well.

Storing `VTempProperty` and `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.

B. Optimization for edge access pattern

While some graph algorithms (e.g. BFS and SSSP) operate with *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 `EdgeIDTable`, all edges in a graph are accessed and processed for every iteration; we call these algorithms *complete edge access algorithms*. For *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 *Apply*

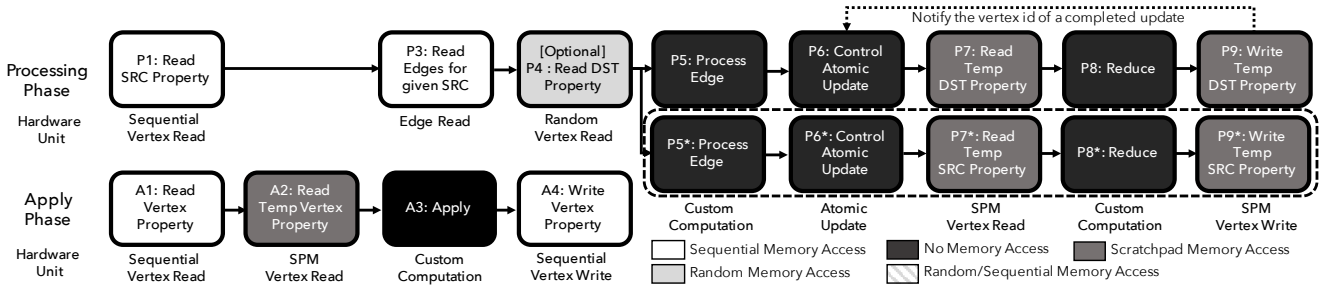

Fig. 6: Optimized Graphicionado pipeline. Note that this pipeline includes optimizations (Section IV-B, Section IV-D) that are only applicable for some of the algorithms.

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.

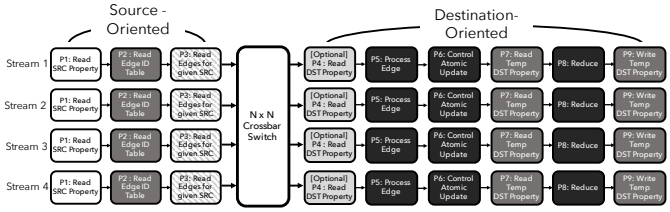

Fig. 7: Parallel implementation of Graphicionado. This diagram omits the *Apply* phase pipeline which is parallelized in a similar manner.

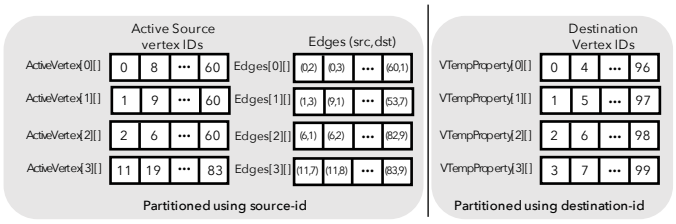

Fig. 8: Data layout for the parallel implementation of Graphicionado.

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

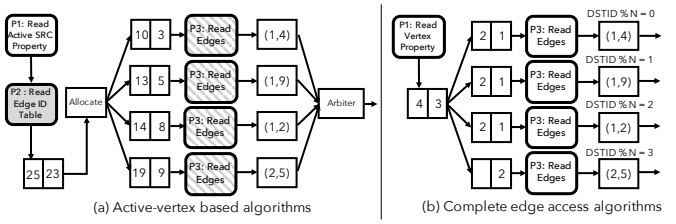

Fig. 9: Implementations of parallelized edge accesses. A single stream is shown.

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 *Number of Vertices* $\times$ *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

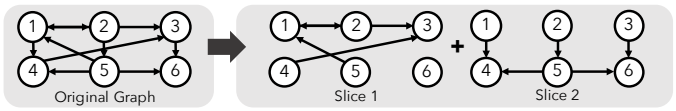

Fig. 10: Graph slicing Example.

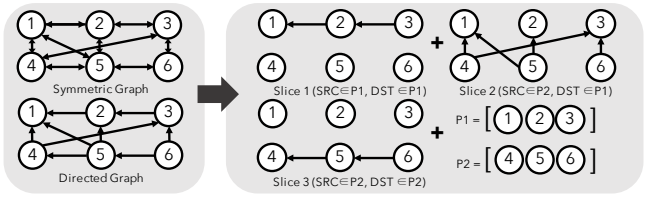

Fig. 11: Extended graph slicing scheme for symmetric graph optimization.

$(1, 2, 3)$ and $P2$ 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.

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 $= \{(u, v)|u \in P1, v \in P1\}$, Slice2 $= \{(u, v)|u \in P2, v \in P1\}$, and Slice3 $= \{(u, v)|u \in P2, v \in P2\}$. 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

Src Partition	Dst Partition	Store Current SrcSPM	Load Next SrcSPM	Apply on DstSPM	Copy SrcSPM to DstSPM
X	1				
4	1	O			
3	1	O			
2	1			O	O
X	2				
4	2	O	O		
3	2			O	O
X	3		O		
4	3			O	
4	4			O	

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

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 $\lfloor \texttt{vid}/N \rfloor$ in the coarsened `EdgeIDTable` where `vid` is the vertex id and N is the coarsening factor.

Edges[] in Memory

Index	1	2	3	4	5	6	7	8	9	10	11
Edge	(1,2)	(1,4)	(2,1)	(2,3)	(2,5)	(3,6)	(4,3)	(5,1)	(5,4)	(5,6)	*

(a) EdgeIDTable

Index	1	2	3	4	5	6
EID	1	3	6	7	8	11

(b) Coarsened (2x) EdgeIDTable

Index	1	2	3
EID	1	6	8

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 effort for area optimization. The slowest module has a critical 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 acqusition PCI board [37]. Table II shows the system parameters of the evaluated system.

	Graphicionado	Software Framework
Compute Unit	8 × Graphicionado Streams @ 1Ghz	16 × Haswell Xeon Cores @ 2.3Ghz
On-chip memory	8MB per stream (total 64MB) eDRAM scratchpad (2Ghz / 1.5ns latency)	L1I/L1D: 512KB/8-way L2: 4MB/8-way LLC: 40MB
Off-chip memory	4 × DDR4-2133 17GB/s channel	4 × DDR4-2133 17GB/s channel

TABLE II: System used for Graphicionado and software graph framework evaluation.

Graph Datasets Table III describes the graph datasets used for our 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*, *Wik*, *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.

Graph	#Vertices	#Edges	Brief Explanation
Flickr (FR) [13]	0.82M	9.84M	Flickr Crawl Graph
Facebook (FB) [51]	2.93M	41.92M	Facebook User Interaction Graph
Wikipedia (Wiki) [13]	3.56M	84.75M	Wikipedia Link Graph
LiveJournal (LJ) [13]	4.84M	68.99M	LiveJournal Follower Graph
RMAT Scale 23 (RMAT) [1]	8.38M	134.22M	Synthetic Graph
Twitter (TW) [28]	61.57M	1468.36M	Twitter Follower Graph
Netflix (NF) [7]	480K users, 18K movies	99.07M	Netflix Prize Bipartite Graph
Synthetic Bipartite (SB) [43]	997K users, 21K items	248.94M	Synthetic Bipartite Graph

TABLE III: Graph datasets used for the evaluation

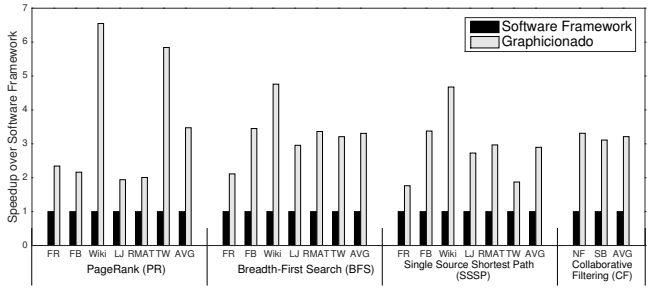

Fig. 14: Graphicionado Performance† normalized to the software graph processing framework performance.

†Scratchpad memory usage scaling techniques in Section VI are used for large graphs — PR-TW(4x), BFS-TW(8x), SSSP-TW(8x), CF-NF(2x), and CF-SB(4x)

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.

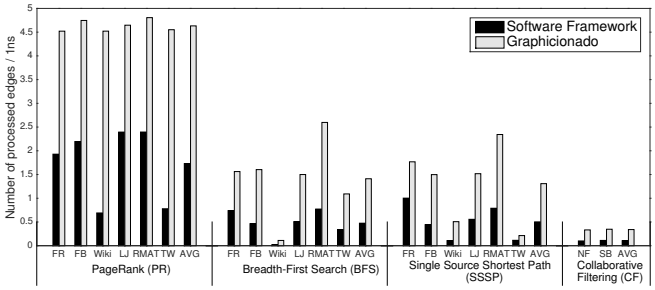

Fig. 15: Throughput of the Graphicionado and the software graph processing framework(GraphMat).

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.

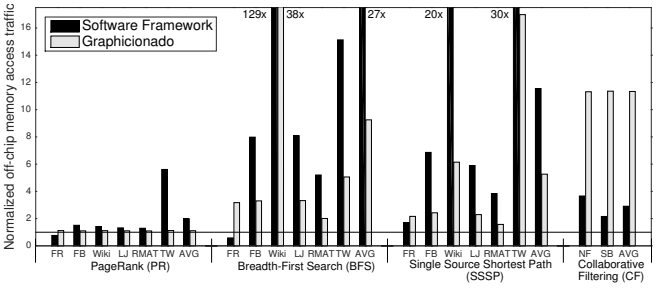

Fig. 16: Off-chip communication traffic of Graphicionado and the software graph processing framework normalized to optimal communication case.

Communication Efficiency Fig. 16 shows the efficiency of Graphicionado and GraphMat's off-chip memory accesses. The efficiency of the off-chip communication is defined as the ratio of its off-chip memory traffic normalized to the optimal communication — the amount of off-chip memory accesses in an imaginary device which has enough on-die storage to reuse all the data for each iteration but not across iterations. For PageRank, both Graphicionado and GraphMat exhibit quite a bit less off-chip communication than other algorithms because PageRank does not incur off-chip random accesses overhead. For BFS and SSPS, Graphicionado accesses 2×-3× more off-chip data than the optimal case in most of the graphs (except *Wiki*) and GraphMat uses significantly more bandwidth than Graphicionado. This is because off-chip bandwidth waste (i.e. reading and writing 64-byte memory when only a small portion of it is useful) are much more common when executing in the software framework. Lastly, Graphicionado running CF consumes roughly 10x more bandwidth than the optimal case because Graphicionado loads destination vertex properties at 128 bytes per edge. While the software framework uses much less off-chip communication for CF, its performance is heavily limited by other factors (i.e., memory latency, compute throughput) as shown in Fig. 14.

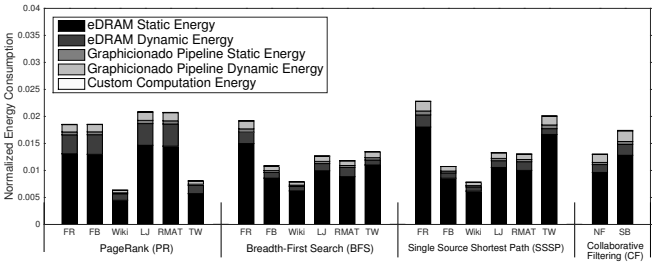

Fig. 17: Graphicionado energy consumption normalized to the energy consumption of Xeon processor running software graph processing framework.
† Custom Computation Energy contributes to less than 1% of the total energy

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 2× speedup. Applying edge access parallelization for *active-vertex based algorithms* provides an additional 1.2× speedup and the combination of edge access parallelization and destination vertex property parallelization provides another 2× speedup on CF. PageRank, however, does not see any extra speedup

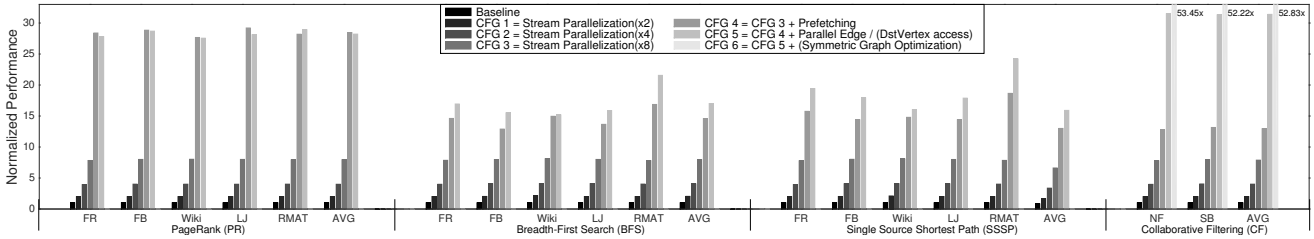

Fig. 18: Effect of parallelizations and optimizations.

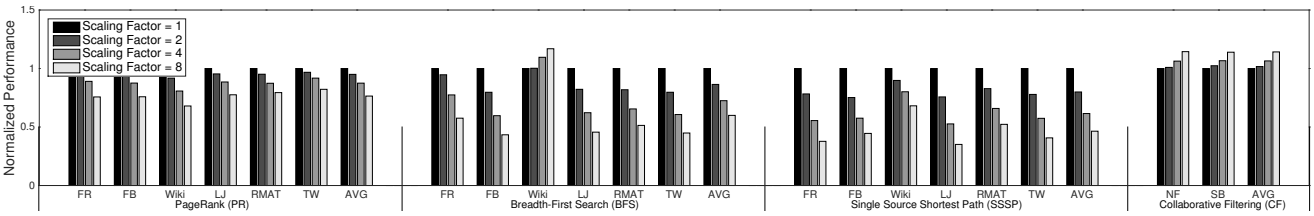

Fig. 19: Effect of graph slicing and edge table coarsening on performance.

from `CFG 4` to `CFG 5` since it is already bounded by memory bandwidth at this point. Lastly, symmetric graph optimization provides another 1.7× 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

[1] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, James A. Ang, "Introducing the Graph 500," Cray User's Group (CUG), May 2010.

[2] D. Agarwal and B.-C. Chen, "Machine learning for large scale recommender systems," 2011, ICML Tutorial.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[4] "Apache giraph," The Apache Software Foundation. [Online]. Available: http://giraph.apache.org/

[5] "Spark GraphX," The Apache Software Foundation. [Online]. Available: http://spark.apache.org/graphx/

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC, 2012.

[7] J. Bennett, S. Lanning, and N. Netflix, "The netflix prize," in *In KDD Cup and Workshop in conjunction with KDD*, 2007.

[8] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *International Journal of High Performance Computing Applications.*, vol. 25, no. 4, 2011.

[9] S. C. Carlson, "Encyclopedia Britannica: Königsberg Bridge Problem," http://www.britannica.com/topic/Konigsberg-bridge-problem.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *In 4th SIAM International Conference on Data Mining*, ser. SDM '04, 2004.

[11] Cray, "Real-Time Discovery in Big Data Using the Urika-GD Appliance," 2014. [Online]. Available: http://www.cray.com/sites/default/files/resources/Urika-GD-WhitePaper.pdf

[12] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA," in *International Symposium on FPGA*, 2016.

[13] T. Davis, "The university of florida sparse matrix collection." http://www.cise.ufl.edu/research/sparse/matrices.

[14] 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*, ser. GRADES'14. ACM, 2014.

[15] D. Gage, "The Wall Street Journal: The New Shape of Big Data," 2013, http://www.wsj.com/articles/SB10001424127887323452204578288264046780392.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012.

[17] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD, 2013.

[18] 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*, ser. ASPLOS XVII. ACM, 2012.

[19] HP Labs, "CACTI," http://www.hpl.hp.com/research/cacti/.

[20] "Intel Core i7-5775C Processor," Intel, 2015. [Online]. Available: http://ark.intel.com/products/88040/Intel-Core-i7-5775C-Processor-6M-Cache-up-to-3_70-GHz

[21] Intel Corporation, "Intel vtune amplifier 2016," 2016, https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[22] "ICT Facts and Figures - The world in 2015," International Telecommunications Union, 2015. [Online]. Available: http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2015/ITU_Key_2005-2015_ICT_data.xls

[23] "Galois," ISS Group at University of Texas. [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois

[24] J. M. Pérez and P. Sánchez and M. Martínez, "High memory throughput fpga architecture for high-definition belief-propagation stereo matching," in *The 3rd International Conference on Signals, Circuits and Systems*, ser. SCS, 2009.

[25] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 9th IEEE International Conference on Data Mining*, ser. ICDM. IEEE Computer Society, 2009.

[26] P. K.Gupta, "Intel Xeon+FPGA Platform for the Data Center," 2015, FPL 2015 Workshop on Reconfigurable Computing for the Masses. [Online]. Available: http://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf

[27] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2013.

[28] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW, 2010.

[29] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012.

[30] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Technical Report, 1998.

[31] D.-F. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Temam, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator." in *The 20th International Conference on Architectural Support for Programming Langugages and Operating Systems*, ser. ASPLOS, 2015.

[32] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2015.

[33] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new parallel framework for machine learning," in *UAI*, July 2010.

[34] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *The 22nd IEEE Symposium on High Performance Computer Architecture*, ser. HPCA, 2016.

[35] 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*, ser. SIGMOD, 2010.

[36] S. McGettrick, D. Geraghty, and C. McElroy, "An FPGA architecture for the Pagerank eigenvector problem," in *International Conference on Field Programmable Logic and Applications*, ser. FPL, 2008.

[37] National Instruments, "Make Accurate Power Managements with NI Tools." [Online]. Available: \http://www.ni.com/white-paper/7077/en/

[38] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martíinez, and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM, 2014.

[39] NVIDIA Corporation, "The NVIDIA Graph Analytics Library (nvGRAPH) ," 2016, https://developer.nvidia.com/nvgraph.

[40] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA, 2016.

[41] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP. ACM, 2013.

[43] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2014.

[44] T. Stocky and L. Rasmussen, "Introducing Graph Search Beta," http://newsroom.fb.com/news/2013/01/introducing-graph-search-beta/.

[45] J. Stuecheli, "POWER8," 2013, Hot Chips: A Symposium on High Performance Chips. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.210-POWER-Studecheli-IBM.pdf

[46] 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," *Proceedings of the VLDB Endowment*, 2015.

[47] Y. Tamir and G. L. Frazier, "High-performance multi-queue buffers for VLSI communications switches," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. IEEE Computer Society Press, 1988.

[48] The MITRE Corporation, "An Overview of the MITRE Cyber Situational Awareness Solutions," 2015, https://www.mitre.org/sites/default/files/publications/pr-15-2592-overview-of-mitre-cyber-situational-awareness-solutions.pdf.

[49] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP, 2016.

[50] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA, 2010.

[51] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys, 2009.

[52] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2014.

[53] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating large-scale single-source shortest path on FPGA," in *IEEE International Parallel and Distributed Processing Symposium Workshop*, ser. IPDPSW, 2015.