2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)

GraphABCD: Scaling Out Graph Analytics with Asynchronous Block Coordinate Descent

Yifan Yang[†]
Tsinghua University
yifany@csail.mit.edu

Zhaoshi Li
Tsinghua University
lizhaoshi@tsinghua.edu.cn

Yangdong Deng
Tsinghua University

Zhiwei Liu
Tsinghua University

Shouyi Yin
Tsinghua University

Shaojun Wei
Tsinghua University

Leibo Liu[*]
Tsinghua University
liulb@tsinghua.edu.cn

***Abstract*—It is of vital importance to efficiently process large graphs for many data-intensive applications. As a result, a large collection of graph analytic frameworks has been proposed to improve the per-iteration performance on a single kind of computation resource. However, heavy coordination and synchronization overhead make it hard to scale out graph analytic frameworks from single platform to heterogeneous platforms. Furthermore, increasing the convergence rate, i.e. reducing the number of iterations, which is equally vital for improving the overall performance of iterative graph algorithms, receives much less attention.**

In this paper, we introduce the Block Coordinate Descent (BCD) view of graph algorithms and propose an *asynchronous heterogeneous* graph analytic framework, GraphABCD, using the BCD view. The BCD view offers key insights and trade-offs on achieving high convergence rate of iterative graph algorithms. GraphABCD features fast convergence under the algorithm design options suggested by BCD. GraphABCD offers algorithm and architectural supports for asynchronous execution, without undermining its fast convergence properties. With minimum synchronization overhead, GraphABCD is able to scale out to heterogeneous and distributed accelerators efficiently. To demonstrate GraphABCD, we prototype its whole system on Intel HARPv2 CPU-FPGA heterogeneous platform. Evaluations on HARPv2 show that GraphABCD achieves geo-mean speedups of 4.8x and 2.0x over GraphMat, a state-of-the-art framework in terms of convergence rate and execution time, respectively.

***Index Terms*—graph analytics, hardware accelerator, FPGA, heterogeneous architecture**

I. Introduction

Graphs provide a powerful abstraction for many scientific and engineering problems and thus have been widely used in many important applications. With the advent of the Big Data era, efficiently processing large scale graphs is becoming a central concern for industry-scale applications. As a result, recent years witnessed a strong momentum to develop high-performance graph analytic frameworks on a variety of computing resources ranging from CPU [2], [6], [19], [42], [52], [59], GPU [14], [15], [35], [45], FPGA [7], [9], [28], [55], [57], [58], to ASIC [1], [33], [53]. The plurality of computing platforms brings heterogeneity to graph analytics.

Many graph algorithms, e.g. PageRank, Breadth-First Search, are iterative where algorithms iterate on every vertex of the graph multiple times until an algorithm-specific convergence criterion is met. Two equally important factors $\#\_of\_iterations$ and $runtime\_per\_iteration$ contribute to the runtime[1] of iterative algorithms as shown in Equation (1):

$$\textbf{runtime} = \textbf{\#\_of\_iterations} \times \textbf{runtime\_per\_iteration} \quad (1)$$

According to the above decomposition, we see two opportunities to improve the performance of graph analytic frameworks for iterative algorithms. First, **runtime_per_iteration** can be further optimized by integrating heterogeneous computing resources into the system. Extensive optimizations on per-iteration performance have already been applied to frameworks on a single platform, e.g. CPU, GPU, FPGA, ASIC. Heterogeneity arises as a promising solution to further reduce the $runtime\_per\_iteration$ of the framework by scaling out the computation to all kinds of computing resources in the system. Nevertheless, scaling out graph analytic frameworks to heterogeneous platforms is difficult due to heavy coordination and synchronization overhead.

Asynchronous framework has the potential to drastically simplify coordination and synchronization operations in the heterogeneous system. In asynchronous graph processing, the start of a vertex's processing doesn't need to wait until the completion of another vertex's processing (or the completion of the current iteration). The vertex takes the most recent graph information visible to it and starts processing non-blockingly. Existing asynchronous solutions still involve fairly complicated coordination mechanisms relying on locks or barriers [24], [33], [60]. *Algorithm and architectural support for asynchronicity with minimum coordination overhead are imperative for scaling out graph analytic framework to heterogeneous systems.*

Second, **#_of_iterations** can also be effectively reduced with proper algorithm tools. But it receives much less attention. In the context of iterative graph algorithm, how well the algorithm converges is directly related to $\#\_of\_iterations$.

[†] This work was done while Yifan Yang was at Tsinghua University. Yifan Yang is now at MIT CSAIL.

[*] Corresponding author: Leibo Liu (liulb@tsinghua.edu.cn)

[1] We consider the total computation time by all threads.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00043

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

However, previous work either utilizes empirical knowledge to optimize convergence rate or conducts the optimization in a case-by-case manner [24], [46], [52]. *Therefore, a systematic approach to analyzing and optimizing the convergence rate of iterative graph algorithms is imperative.*

To address these challenges, we introduce the Block Coordinate Descent (BCD) view of iterative graph algorithms. We apply BCD, whose convergence properties are well studied [29], [30], to graph domain to help understand the convergence rate (number of iterations) of graph algorithms. By configuring different algorithm design options of BCD, it systematically provides trade-offs between convergence rate and runtime per iteration. In case of heterogeneity, it further provides convergence guarantee when the framework scales out to asynchronous settings with infrequent coordination.

We then propose **GraphABCD**, a heterogeneous Graph analytic framework with Asynchronous Block Coordinate Descent, which consists of a CPU and hardware accelerators. GraphABCD framework achieves fast algorithm convergence thanks to the algorithm design options suggested by BCD. Through algorithm and architecture co-design, GraphABCD supports efficient asynchronous processing with low coordination and synchronization overhead (barrierless and lock-free). This allows the GraphABCD system to scale out to heterogeneous and distributed platforms without undermining the performance and convergence rate by complicated synchronization. GraphABCD's memory system is optimized to preserve more locality and improves bandwidth utilization between the accelerator and CPU. Two optimizations, hybrid execution and BCD-guided priority scheduling, are also implemented. We further prototype the accelerator using FPGA and deploy the whole GraphABCD system on the Intel HARPv2 CPU-FPGA hybrid system [32]. Evaluations show GraphABCD achieves geo-mean speedups of 4.8x and 2.0x over the state-of-the-art GraphMat [40] framework in terms of convergence rate and execution time, respectively.

This paper makes the following contributions.

- We introduce the view of asynchronous Block Coordinate Descent on iterative graph algorithms to enable systematic discussions on the convergence properties of graph algorithms and trade-offs on achieving fast algorithm convergence. (Sec. III).
- We propose *asynchronous* GraphABCD framework on *heterogeneous* system. GraphABCD achieves fast algorithm convergence under the guidance of BCD. With minimized coordination as the other design goal, GraphABCD can efficiently scale out to asynchronously executed heterogeneous accelerators. (Sec. IV-A,IV-B).
- We prototype GraphABCD whole system on the Intel HARPv2 CPU-FPGA heterogeneous platform (Sec. IV-C), and evaluations show improved convergence rate and performance compared to the state-of-the-art framework (Sec. V).

II. Background

A. Graph Analytic Frameworks

We will give the background of the GAS programming model and BSP execution model of graph analytic frameworks.

Gather-Apply-Scatter (GAS) [10] is proposed to abstract the computation pattern of graph algorithms. Due to its simplicity, scalability and generality, GAS model has been widely adopted in mainstream graph analytic frameworks [7], [9], [10], [12], [15], [18], [28], [36], [38], [40], [47], [50], [59]. In the model, each vertex iteratively executes a vertex program until a convergence condition is met. In GAS vertex program, each vertex `Gathers` information from neighbours, `Applies` update to itself, and `Scatters` the update to neighbours. GAS maintains a clean programming model and shows how information flows across the whole graph. Most graph algorithms, like PageRank (PR), Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), as well as graph-based machine learning algorithms, such as Collaborative Filtering (CF), Label Propagation (LP), and Bayes Propagation (BP), fit in the GAS paradigm.

Bulk Synchronous Processing (BSP) execution model is proposed in Pregel [26] to solve large scale graph processing on distributed platforms, where on every iteration all vertices first compute the updated value and then all the updates are committed to memory at the same time. BSP requires a global memory barrier every iteration, which degrades its performance significantly on heterogeneous and distributed systems.

B. Block Coordinate Descent (BCD)

An optimization problem is defined to find the vector $\mathbf{x} \in \Re^n$ to minimize the objective function (loss function) $F(\mathbf{x})$. In most cases, this problem cannot be analytically solved. So it requires iterative updates to the vector $\mathbf{x}$ to gradually minimize $F(\mathbf{x})$ until it reaches its minimal point (converges), i.e., if the discrepancy of the objective function between consecutive iterations is smaller than a certain threshold, we consider its minimum is reached and terminate the algorithm.

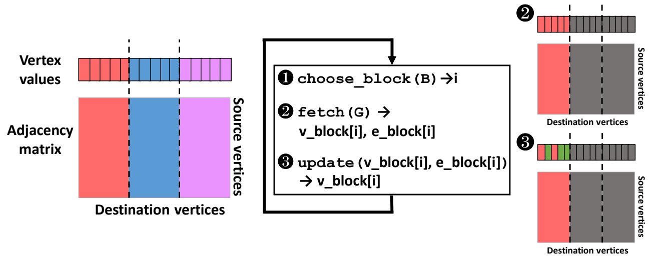
Block Coordinate Descent (BCD) is an iterative algorithm to solve the optimization problem. In BCD, $\mathbf{x}$ is decomposed into $\boxed{\text{s}}$ block variables $\mathbf{x}_1, ..., \mathbf{x}_s$. So the vector $\mathbf{x}$ in the k-th iteration is $\mathbf{x}^k = \left[\mathbf{x}_1^k, \mathbf{x}_2^k, ...\mathbf{x}_s^k\right]^T$. In iteration k, the value of all the blocks are fixed except for a $\boxed{\textit{selected block } x_i^k}$ (the i-th block). The algorithm only updates the variables belong to block i:

$$\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \alpha_{i_k} \mathbf{d}_i^k \tag{2}$$

where α_{i_k} is the step size and $\boxed{\mathbf{d}_i^k \textit{is the descent direction}}$. Note that the boxed terms in the above illustration are parameterizable. Different configurations of BCD result in various convergence rates and trade-offs, which we will discuss in Sec. III-C.

Given the well-studied BCD algorithm on solving optimization problems [29], [30], we view the graph problems from the BCD perspective and reform the convergence knowledge

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.



(a)Partition Graph into Vertex and Edge Blocks

(b)Execution Flow of BCD on Partitioned Graph

Fig. 1: Applying BCD to Graph Domain

of existing graph algorithms using BCD. In this way, we can discuss the convergence rate of iterative graph algorithms and trade-offs of different BCD algorithm selections systematically.

III. Convergence Under BCD View

Block Coordinate Descent (BCD) offers us a systematic way to understand convergence properties of graph algorithms. Applying it to the domain of graph processing (Sec. III-A) enables us to see what algorithmic design options suggested by BCD affect the convergence rate (*#_of_iterations*) of it (Sec. III-B). Thus, by configuring different combinations of algorithm design options in BCD, we can explore fast convergent configurations and trade-offs on overall performance (Sec. III-C).

The above analysis is conducted on Synchronous BCD. To scale graph analytics out to heterogeneous platforms, we introduce the view of Asynchronous BCD to reduce the synchronization overhead (Sec. III-D). In this way, we are able to jointly optimize both **runtime_per_iteration** (from the hardware architecture side) and **#_of_iterations** (from algorithm side) on a broad range of heterogeneous and distributed platforms to achieve better performance.

A. Applying BCD on Graph Algorithms

There are two questions to solve in order to apply BCD (designed for solving optimization problems) on graph algorithms (designed for solving graph problems):

1) How to map the variables in the optimization problem to the graph domain?
2) What are the optimization objectives and update functions of different graph problems?

The answer to question 1 is that each vertex's value corresponds to a variable in the optimization problem. Since graphs nowadays are extremely large. They are often partitioned into subgraphs to fully exploit the memory hierarchy in modern computer systems or conduct out of core processing [2], [7], [10], [18], [31], [51], [53], [59]. Fig. 1(a) shows a common way of partitioning graphs where the array of vertex values are partitioned into several blocks (intervals). The adjacency matrix is also sliced into the same numbers of chunks according to their destination vertices. In this way, the BCD view on executing iterative algorithms on partitioned graphs is shown in Fig. 1(b). The graph algorithm first selects a block id according to some selection rules. It then fetches the corresponding vertex block and adjacency matrix chunk (edge block). An update function takes in the inputs. Finally, the function generates new vertex values of the current block (while keeping the value of other blocks unchanged) and stores them back to memory.

The notation used in this section is as follows. Given a graph $G = \{V, E\}$ with V as the vertex set and E as the edge set, let $\mathbf{x}$, a vector of size $|V|$, denote the values associated with each vertex and $F(\mathbf{x})$ being the objective function associated with the graph problem. Let v_i denote the i-th vertex. Let e_{ij} denote the edge from v_i to v_j. Besides let d_v denote the out degree of vertex v. Let $\mathbf{A}$ denote the adjacency matrix of the graph. Unless stated, the rest of the paper follows this notation.

We will first show examples of how Collaborative Filtering (CF) and PageRank (PR), representing machine learning-based and conventional graph problems respectively, can be translated to the optimization view. Then we will generalize the BCD formulation beyond PR to more popular graph problems.

1) Collaborative Filtering: is a machine learning algorithm for recommendation system on bipartite graphs, which is based on factorizing the rating (of users to items) matrix. As a machine learning problem, the objective function is well established as the L-2 error on the training data and an additional regularization term [56]

$$\min_{\substack{\mathbf{x}^p\in\Re^{PH}\\ \mathbf{x}^q\in\Re^{HQ}}} F\left(\mathbf{x}_p, \mathbf{x}_q\right) = \sum_{e_{ij}\in E} \left(e_{ij} - x_i^p \cdot x_j^q\right)^2 + \lambda\left(\left\|x_i^p\right\| + \left\|x_j^q\right\|\right)$$

where $\mathbf{x}_p$ is the feature matrix of all users and $\mathbf{x}_p$ is the feature matrix of all items. $err_{ij} = e_{ij} - x_i^p \cdot x_j^q$ is the error between the ground truth rating and estimated rating.

Take the partial derivative of the objective function gives the following coordinate descent rule for each user and item:

$$\begin{aligned} x_i^{p,k+1} &= x_i^{p,k} + \alpha\left(err_{ij} x_j^{q,k} - \lambda x_i^{p,k}\right) \\ x_j^{q,k+1} &= x_j^{q,k} + \alpha\left(err_{ij} x_i^{p,k} - \lambda x_j^{q,k}\right) \end{aligned}$$

where α is the learning rate. This update function is consistent with the iterative formula of conventional CF algorithm.

2) PageRank: is the workhorse of modern information retrieval applications. We will detail the formulation of PR problem as an example of conventional graph problems. The algorithm iteratively updates the current vertex value $\mathbf{x}$ until it reaches the final result $\mathbf{x}_0$, which is often referred to as stationary point. This stationary point satisfies the property of $\mathbf{x}_0 = \mathbf{P}\mathbf{x}_0 + \mathbf{b}$, where $\mathbf{P}$ and $\mathbf{b}$ are determined by the graph problem. The goal of the algorithm is to make $\mathbf{x}$ approach $\mathbf{x}_0$ and therefore $\mathbf{x}$ should satisfy the same stationary property. Thus we construct the objective function in the form of L-2 norm to optimize $\mathbf{Px} + \mathbf{b} - \mathbf{x} = 0$, i.e.:

$$\min_{\mathbf{x}\in\Re^n} F(\mathbf{x}) = \frac{1}{2}\left(\mathbf{Px} + \mathbf{b} - \mathbf{x}\right)^2 \tag{3}$$

Where $\mathbf{P} = \alpha(\mathbf{G}^{-1}\mathbf{A})^T$, $\mathbf{b} = \frac{1-\alpha}{|V|}\mathbf{e}$, $\mathbf{G}$ is the diagonal matrix of out-going degrees and α is the damping factor for PR.

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

We select the descent direction $\mathbf{d}$ to be the gradient of the objective function. The gradient with respect to x_i is

$$\nabla_{x_i} F(\mathbf{x}) = \sum_{j=1}^{|V|} p_{ij} x_j - \frac{1-\alpha}{|V|} \tag{4}$$

where $p_{ij} = -\alpha / |G_j|$, if vertex i and j are adjacent; $p_{ij} = 0$ if i and j are not adjacent; and $p_{ij} = 1$ if $i = j$.

We equal Equation (4) to zero and rearrange the terms to the iterative form:

$$x_i^{k+1} = \frac{1-\alpha}{|V|} + \alpha \sum_{j \in C_i} \frac{x_j^k}{|G_j|}$$

which is the update function of x_i for iterative PageRank.

Discussion: Following the method described above and substituting $\mathbf{P}$ and $\mathbf{b}$ in Equation (3), the objective of Single Source Shortest Path (SSSP), for example, is $\min_{\mathbf{x}\in\Re^n} F(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{|V|}\left(x_i - \min_{j\in C_i}(x_j + a_{ji})\right)^2$, where C_i is v_i's source vertices and a_{ji} is the edge weight between v_j and v_i. The objective function of other popular algorithms (e.g. Breadth-First Search (BFS), Connected Components (CC)) can be similarly constructed. In this way, we formulate graph problems into optimization forms and can leverage BCD to solve them.

B. Design Parameters of BCD

BCD algorithm has 3 types of design parameters for users to specify (boxed terms in Sec. II-B) in order to trade off between convergence rate and execution efficiency. They are *block size, block selection method* and *block update method* [29]. These 3 options are classified as algorithm design options and affect $\#\_of\_iterations$ of Equation (1). We will introduce the three options in this subsection and present the trade-offs of various combinations of the options in the next subsection.

Block Size describes how many vertices are allocated into one block. It can varies from $|V|$ to *1*.

A block size of $|V|$ means there is only one block, which is full gradient descent. Each vertex stores the tentative update until all vertices have finished reading data from neighbors. Then all vertices commit their updates synchronously. This method is often called *Bulk Synchronous Processing* (BSP) [26] or Jacobi style iterative algorithm [20].

The other extreme case is a block size of *1*, meaning only 1 vertex is selected to be updated in each iteration. The update is visible to other vertices immediately. The choice sits in the middle of two extremes is a block size of n ($1 < n < |V|$). This is often referred as Gauss-Seidel iterative algorithm [20].

Block Selection Method (scheduling strategy) defines the strategy to choose a block to perform BCD in each iteration (`choose_block` function in Fig. 1(b)). A block can be selected according to a predefined fixed order (*cyclic*) or a dynamically updated order (*priority*).

Cyclic selection is the simplest to implement. It repeatedly selects blocks to be updated according to fixed order (e.g., the block id). The memory access for cyclic scheduling is predictable and thus prefetching of blocks can be performed.

Priority scheduling selects a block according to a dynamically maintained priority. Definitions of priority can vary across different scheduling algorithms. For example, $\Delta-$stepping scheduling is specifically designed for Single Source Shortest Path (SSSP) and its priority is defined as the depth of each vertex bucket. More generic selection methods take into account higher-order information, e.g. gradient and Hessian, of the objective function. Either way, priority scheduling converges faster by taking advantage of more global information than cyclic selection.

Block Update Method specifies the iterative update function used by graph algorithms (`update` function in Fig. 1(b)). More specifically, it defines the step size α_{i_k} and direction $\mathbf{d}_i^k$ in Equation (2). The coordinate descent direction $\mathbf{d}_i^k$ can be decided by *gradient, newton* or other approaches. *Gradient* Descent is the most straightforward approach. $\mathbf{d}_i^k$ is along the negative gradient $-\nabla_i F\left(\mathbf{x}^k\right)$ of the objective function. Step size α_{i_k} is either fixed or obtained by a exhaustive line search. The combination of fixed step size and gradient descent has been adopted by most graph algorithms to reduce the cost of updates.

C. Constructing Fast Convergent Graph Algorithm

We first show algorithm design options to yield the best convergence rate. Then we gradually trade off convergence rate with efficiency to reveal the full spectrum of BCD algorithm.

From the intuition of update propagation, we can conclude that **a combination of block size 1, priority block selection and gradient update allows the fastest convergence of graph algorithms**, which is also empirically proven by [29], [30]. Intuitively, this improves convergence rate by allowing the most important vertex update visible to the whole graph instantaneously. While the other extreme of worst convergence rate is block size $|V|$, cyclic selection and gradient update.

However, when considering implementing BCD algorithm on computer systems, even though the fastest convergence configuration achieves the lowest $\#\_of\_iterations$, the complexity of the selection may deteriorate $runtime\_per\_iteration$ and therefore degrades the overall performance. We can relax the selection of block size and block selection method to sacrifice convergence rate for improved $runtime\_per\_iteration$.

Trade-off 1: A larger block size, though meaning slower convergence because updates are committed less frequently to memory, tends to have more explicit inter-vertex parallelism and memory locality due to batched processing. It also simplifies the frequent per-vertex fine-grained synchronization (when block size is 1) to fewer coarse-grain coordination among large blocks. For systems with large overheads in conflict resolution or random memory access, larger block sizes are recommended.

Trade-off 2: Priority scheduling converges faster because of incorporating higher-order global information. However, this scheme requires more global coordination among workers to extract information which might cause serious stalling in highly heterogeneous distributed systems. In addition, calculating and maintaining the priority of each block for large-scale

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

problems can lead to complicated logic and control flow. If a decentralized system is used or more regular control flow is preferred, cyclic rule can be adopted in penalty of deteriorated convergence rate.

Thus, selecting the best algorithm design options should consider both the BCD theory and system characteristics.

D. Asynchronous BCD

In a heterogeneous platform, the regular BCD update may incur major synchronization overheads and limited parallelism. For example, suppose block i and j are selected to be updated in iteration k and k+1 respectively. Block $\mathbf{x}_i^k$ is updated to $\mathbf{x}_i^{k+1}$ in iteration k according to Equation (2), which creates a dependency. In iteration k+1, block $\mathbf{x}_j^{k+1}$ needs to calculate the gradient $\mathbf{d}_i^k$, which may depend on the updated block $\mathbf{x}_i^{k+1}$ (if vertices in block i are neighbors of vertices in block j). This dependency chain requires a synchronization after iteration k and prohibits block i and j from executing in parallel. A similar problem arises in distributed machine learning, which leads to a resurgence of researches on asynchronous optimization algorithms [34], [39]. In Asynchronous BCD, updating $\mathbf{x}_j^{k+1}$ can use a stale version of $\mathbf{x}_i$, namely $\mathbf{x}_i^k$ instead of $\mathbf{x}_i^{k+1}$, and therefore no synchronization barrier is needed. As long as the update on $\mathbf{x}_i^k$ is propagated in a bounded delay, the convergence is assured [39]. Moreover, for convex objective function[2], **the asynchronous BCD converges as fast as synchronous one**. This convergence guarantee enables us to explore more asynchronous high-performance graph processing strategies on heterogeneous platforms as long as the communication latency between platforms is bounded.

E. Summary

In this section, we show the convergence properties of the BCD view on iterative graph algorithms, which offers two key insights: 1) Priority gradient update with a block size of 1 achieves the fastest convergence (empirically proved in Sec. V-B) and comes with two trade-offs with block size and block selection rule. 2) In asynchronous settings, the convergence properties are the same as synchronous ones.

IV. GraphABCD System

We design and prototype GraphABCD, a *heterogeneous* Graph analytic framework with *Asynchronous* Block Coordinate Descent, which consists of a CPU and hardware accelerator (Fig. 2). Using the insights of BCD, GraphABCD framework systematically improves convergence rate (reduces $\#\_of\_iterations$) of iterative graph algorithms. As a promising platform to improve per iteration performance, heterogeneous architecture suffers from heavy coordination and synchronization overhead[3]. GraphABCD's barrierless and lock-free asynchronous design reduces synchronization and coordination so that computation can efficiently scale out to heterogeneous accelerators, which further optimizes $runtime\_per\_iteration$. GraphABCD's six key design techniques are shown below:

1) We select the BCD algorithm design options to achieve fast convergence. The algorithm design leverages the convergence rate analysis provided by BCD as well as the trade-offs on heterogeneous systems.
2) System design options are tailored for high utilization of memory bandwidth. Better memory locality improves communication efficiency within heterogeneous system.
3) GraphABCD system supports asynchronous execution intrinsically with minimum synchronization overhead (barrierless and lock-free), whose convergence property is guaranteed by asynchronous BCD. Asynchronicity allows the computation of graph algorithms to distribute and scale out to heterogeneous platforms.
4) GraphABCD features hybrid execution optimization by allocating computation to both accelerator and available CPUs to efficiently exploit the heterogeneous system.
5) GraphABCD system is equipped with BCD-guided priority block selection method optimization to further improve the convergence rate.
6) We prototype the GraphABCD full system on Intel HARPv2 CPU-FPGA heterogeneous platform.

In this section, we first present the first three algorithm system co-design techniques to jointly optimize the convergence rate and per-iteration performance of graph algorithms (Sec. IV-A). Then we describe two optional optimizations to GraphABCD base system to further exploit the power of BCD and heterogeneous system (Sec. IV-B). Finally, execution example and implementation details of GraphABCD prototype on CPU-FPGA hybrid platform are revealed (Sec. IV-C).

A. GraphABCD Design

1) Achieving Fast Convergence: GraphABCD aims to achieve fast convergence with lightweight synchronization for heterogeneous platforms. So it supports both *priority* and *cyclic gradient* update with a block size of n.

Smaller **block size** results in faster convergence. However, it comes at the cost of larger conflict resolution/synchronization overhead, because the system has to synchronize and coordinate with large numbers of smaller blocks. Another overhead with small block size in the context of heterogeneous systems is communication latency and accelerator invocation overhead. LogCA [3] suggests that the offload granularity (block size) should be higher than a threshold so that the block size is large enough to hide the accelerator's communication and invocation latency. To resolve this dilemma, block size of n ($1 < n < |V|$) is used to balance between runtime overhead and convergence rate.

GraphABCD supports *priority* **block selection method** owing to its fast convergent nature. Nevertheless, the runtime overhead of priority scheduling might offset the benefits of the improved convergence rate. Therefore, we also support

[2] The generic BCD objective introduced in Equation (3) is L-2 norm, which is convex. As long as the graph problems can be converted to this form as shown in Sec. III-A, the convexity holds for the objective functions.

[3] The access latency of L2 cache on another core is ~30ns in a NUMA architecture. While exchanging data between CPU LLC and FPGA on Intel HARPv2 system via PCIe is ~300ns, which is a 10x difference.

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

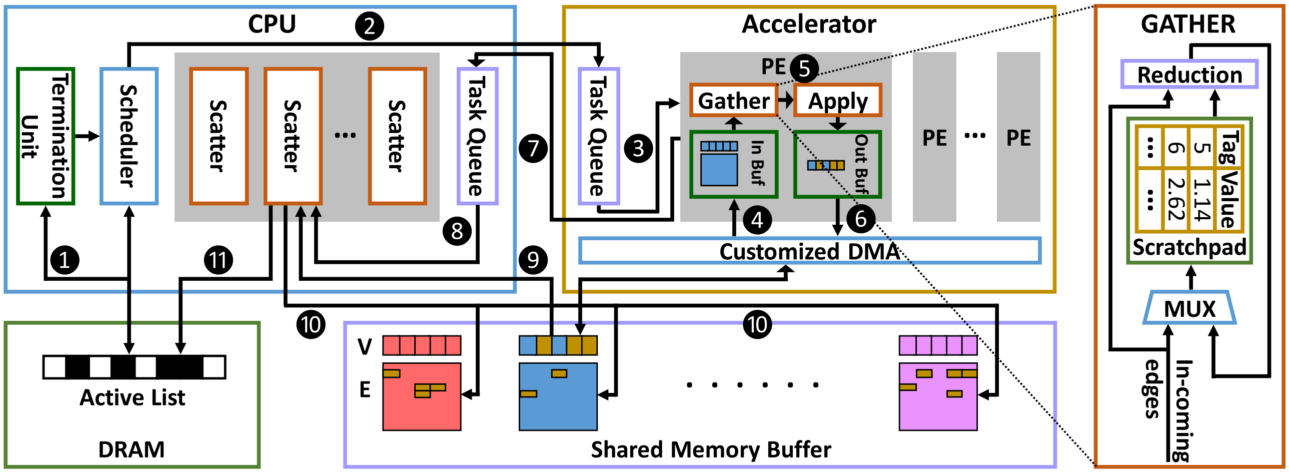
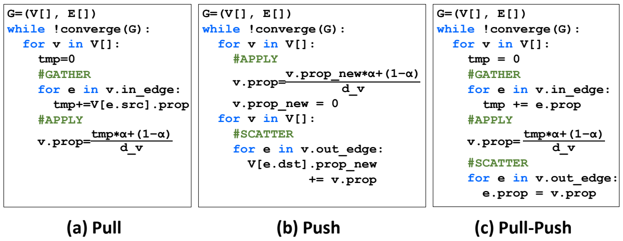


Fig. 2: Architecture, Execution Example and Memory Layout Example of GraphABCD System



```
G=(V[], E[])
while !converge(G):
  for v in V[]:
    tmp=0
    #GATHER
    for e in v.in_edge:
      tmp+=V[e.src].prop
    #APPLY
           tmp*α+(1-α)
    v.prop=───────────
               d_v
```

(a) Pull

```
G=(V[], E[])
while !converge(G):
  for v in V[]:
    #APPLY
           v.prop_new*α+(1-α)
    v.prop=──────────────────
                  d_v
    v.prop_new = 0
  for v in V[]:
    #SCATTER
    for e in v.out_edge:
      V[e.dst].prop_new
              += v.prop
```

(b) Push

```
G=(V[], E[])
while !converge(G):
  for v in V[]:
    tmp = 0
    #GATHER
    for e in v.in_edge:
      tmp += e.prop
    #APPLY
           tmp*α+(1-α)
    v.prop=───────────
               d_v
    #SCATTER
    for e in v.out_edge:
      e.prop = v.prop
```

(c) Pull-Push

Fig. 3: PageRank Example of Three Types of Vertex Operator

the simple cyclic block selection method and leave priority scheduling as an option for users to turn on/off.

2) Improving Memory Locality and Bandwidth Utilization: Graph algorithms are irregular because of the huge volume of random memory accesses. Accelerators typically have limited memory bandwidth with the host CPU. Thus, it is vital to improve memory locality and reduce the memory traffic for the accelerators to achieve better bandwidth utilization.

GraphABCD chooses *pull-push* as the vertex operator and edgelist graph format. Combined with a novel task allocation on heterogeneous system, such design ensures that all of the accelerator memory accesses are sequential.

Vertex Operator describes the type of operator applied to every vertex in a graph. Although in GAS abstraction, every vertex runs the identical GAS program until they all converge, there are three variants of GAS program *pull, push* and *pull-push*. Fig. 3 shows an example of how PageRank algorithm is written in *pull, push* and *pull-push* respectively. *Pull* (Fig. 3(a)) indicates that only GATHER−APPLY stages exist in the vertex program. Here, the GATHER function reads the in-coming edges' source vertex values of the current vertex. *Push* (Fig. 3(b)) represents a two-stage APPLY−SCATTER computation without GATHER. Contrary to pull, SCATTER function in push reads the out-going edges of the vertex and retrieves the destination vertex values using the id obtained from the edge reads. These two designs suffer from random memory accesses to vertex value array V[] in either GATHER or SCATTER. The three types of vertex operators show different trade-offs on memory traffic, memory locality and computational cost.

Pull-push (Fig. 3(c)) consists of a full chain of GATHER-APPLY-SCATTER in the GAS model. In pull-push, the vertex values are copied on the edges. The GATHER function first reads the in-coming edges of the current vertex, which contains the source vertex values. It then applies local GATHER−APPLY computation on the fetched values and generates new vertex value. The updated vertex value is finally copied to all of its out-going edges by SCATTER, waiting for its destination vertices to read. There is no random memory access on the vertex array V[] in pull-push. However, the reads on in-coming edges and writes on out-going edges could be random. Suppose there is one copy of the edges[4]. If the in-coming edges of a vertex are put continuous in memory, then its out-going edges are discontinuous. Similarly, if the out-going edges are adjacent, the in-coming edges are randomly placed. This behavior suggests either in-coming edge reads or out-going edge writes is random and the other is sequential.

In pull-push, since GATHER is the most computationally intensive part (SCATTER only copies values to out-going edges), we offload GATHER to the hardware accelerator. The APPLY function simply processes the stream of GATHER results, which can be concatenated to the GATHER pipeline. If we further offload scatter to accelerator, the memory traffic between accelerator and CPU would be $2|E|$ ($|E|$ for GATHER's in-coming edge reads and $|E|$ for SCATTER's out-going edge writes). However, offloading only GATHER−APPLY would yield less memory traffic of $|E| + |V|$ ($|E|$ for GATHER's in-coming edge reads and $|V|$ for APPLY's updated vertex value writes). Therefore, we allocate GATHER−APPLY to the

[4] If there are multiple copies of the edges, maintaining the consistency among these copies will cause huge runtime overhead. So we only have one copy of the edges in GraphABCD

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

hardware accelerator to reduce the memory traffic. We choose in-coming edge reads to be sequential and out-going edge writes to be random and place the in-coming edges of the same vertex continuously in memory. In this layout, the memory accesses on the hardware accelerator are fully sequential.

We conclude that the *pull-push* vertex operator and allocating GATHER-APPLY to the accelerator can improve locality for accelerator and help it better utilize memory bandwidth.

3) Enabling Asynchronous Execution: The high synchronization overhead among heterogeneous components in the system severely degrades performance if a synchronous execution model is deployed. Since the convergence properties of asynchronicity are guaranteed by asynchronous BCD, the challenge for designing high-performance heterogeneous system falls on how to design *asynchronous* execution with minimum coordination overhead. In GraphABCD, several design features ensure the system support for asynchronous execution, including *state-based* update information, decoupled CPU-accelerator execution, and disjoint block memory layout.

Update Information describes what information within an update is sent from one vertex to another. *State*-based approach sends the updated value to the destination, while *operation*-based approach sends the difference between the old value and the new value to the destination. Considering GraphABCD chooses the pull-push operator, the difference is either writing the updated vertex value (state) to the out-going edges in SCATTER phase or writing the value change (operation) to the edges. For example, the normal PageRank described in Fig. 3(c) is state-based while the PageRank Delta algorithm is the operation-based variant of the same graph algorithm.

In asynchronous graph processing framework, *operation*-based approach needs extra synchronization (e.g. barrier) to ensure the correctness. Otherwise, two update operations to the same destination may cause the overwrite of the previous one, which affects the correctness of the program. These phenomena are especially frequent when priority scheduling is enabled, where the same vertex (block) could be chosen multiple times in a short period resulting in frequent updates to the same destination. To simplify synchronization, we choose *state*-based approach where all updates can eventually be propagated even if some of them are delayed. The adoption of *state*-based update information doesn't have extra synchronization requirements (barrierless and lock-free), which makes asynchronicity possible in GraphABCD.

Another key element for GraphABCD's asynchronous execution design is the *decoupled CPU-accelerator execution* enabled by task queues. As shown in Fig. 2, the accelerator task queue and CPU task queue are the only control logic link between the two components. The CPU thread does not need to wait for the results of a specific block or from a certain PE and the same is true for the accelerator PE. In this way, the CPU and accelerator can execute asynchronously and independently on different blocks without extra dependency constraints, such that the whole system can scale out to more heterogeneous platforms without further coordination logic. This decoupled asynchronous architecture also improves PE utilization and amortizes the load imbalance issue of different blocks by enabling dynamic task allocations (rather than statically assigning a group of blocks to one PE). Note that the communication through task queue has a bounded delay. Therefore, asynchronous BCD ensures our design yields the same convergence properties as synchronous design.

The memory layout introduced in Sec. IV-A2 also contributes to GraphABCD's asynchronous execution. The data required by each accelerator PE is disjoint indicating no memory conflict is incurred. This ensures that PEs can execute asynchronously without extra coordination on memory access conflicts. On the CPU side, although SCATTER accesses are random, they are still disjoint sets of out-going edges meaning it can also be asynchronously executed.

B. GraphABCD Optimizations

CPU-FPGA Hybrid Execution: CPU executes the rather light-weighted scheduling and SCATTER function. Therefore, We construct a CPU version of GATHER-APPLY function and run it when the runtime detects the CPU is under-utilized. The optimization can be easily integrated into GraphABCD's architecture in Fig. 2 by adding an additional task queue on CPU serving the CPU-based GATHER-APPLY.

Note that the reason why GraphABCD can scale the same computation kernel to heterogeneous parts (both CPU and accelerator) in the system is that GraphABCD intrinsically supports asynchronicity. Asynchronous BCD eliminates the need for global coordination between CPU computation resources and accelerator computation resources. Therefore, hybrid execution optimization can fully leverage the additional available CPU resources and scale GraphABCD to all computation resources in the heterogeneous distributed system.

Priority Block Selection: Intuitively, processing blocks with larger gradient in priority makes the optimization objective decrease faster, resulting in improved convergence rate. GraphABCD's priority block selection method utilizes Gauss-Southwell rule [29], [30], [43]. The method states that the block with larger gradient shall be prioritized for processing, i.e. $i = \arg\max_{i \in 1,2,\ldots,s} \left\| \nabla_i F\left(\mathbf{x}^k\right) \right\|$, for faster convergence.

In reality, calculating and maintaining the gradient of each block pose serious runtime overhead to the system. We thus approximate the gradient of each vertex as vertex values' difference between consecutive iterations. Then the priority of a block is a gradient vector where each element refers to the gradient of the vertex in this block. Finally, the L-1 norm of the gradient vector, instead of the computationally expensive L-2 norm, is calculated as the block's magnitude of gradient (the priority of the block) for comparison with other blocks. In this way, the simplification drastically reduces the computational cost of priority scheduling while still retains the fast convergent nature of priority gradient update. The priority scheduling approach can be applied to all graph algorithms supported by GraphABCD transparently.

C. GraphABCD Implementation and Prototype

We prototype the hardware accelerator on FPGA and deploy the whole GraphABCD system on off-the-shelf CPU-FPGA

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

heterogeneous system, the Intel HARPv2 CPU-FPGA system. The reason for using FPGA as the accelerator is two-fold. First, the flexible spatial architecture of FPGA makes it a general platform to emulate the runtime behavior of hardware accelerators (GPU, ASIC, etc.). Second, simulating the communication and coordination between CPU and accelerators is less accurate than the full system run. Existing CPU-FPGA hybrid systems enable us to deploy the entire system and evaluate the performance in real time accurately.

The constructed GraphABCD system on HARPv2 is shown in Fig. 2. The CPU part is implemented in software threads and FPGA (accelerator) part is implemented in Verilog RTL. We explain the implementation by showing an example of how blocks are scheduled and processed on GraphABCD system asynchronously. GraphABCD adopts a similar graph partition strategy described in Fig. 1(a), where vertex value array is cut into blocks and adjacency matrix is sliced into chunks according to destination vertex. We first walk through the execution flow of a single block. Then we illustrate how to scale to multiple blocks asynchronously.

- **Step 1:** The *Termination Unit* on CPU reads the active list of the whole graph (each entry represents the activeness of a vertex block) from DRAM. If all of the blocks are inactive meaning the algorithm converges, then it terminates the program.
- **Step 2:** Software-based *Scheduler* selects a vertex block according to the active list. It then notifies the accelerator by pushing the selected block id into the *Accelerator Task Queue*. The scheduler can be configure to *cyclic* or *priority* mode.
- **Step 3:** When one accelerator *Processing Element* (PE) is idle, it dequeues a new task from the accelerator task queue and starts its execution.
- **Step 4:** The PE sends a request to on-chip *Customized DMA Unit*. The DMA Unit reads the corresponding vertex and edge block from a *memory buffer* (LLC on HARPv2), shared by CPU and accelerator, to PE's *Input Buffer*.
- **Step 5:** The PE processes the input data. It consists of a pipeline of `GATHER-APPLY` function of the GAS vertex program, which produces a new vertex value block.
- **Step 6:** The resulting new vertex block is first buffered at the PE's private *Output Buffer* and then is written back to the shared memory buffer through the DMA unit.
- **Step 7:** After executing the block, the PE sends the finished block id to *CPU Task Queue* waiting for further processing on the CPU.
- **Step 8:** When one CPU thread is idle, it dequeues a new task from the CPU task queue and starts its execution.
- **Step 9:** The thread reads the updated vertex value block from the shared memory buffer with accelerator. It conducts `SCATTER` function from the GAS vertex program.
- **Step 10:** The generated results from `SCATTER` unit are broadcast to the updated block's out-going edge blocks.
- **Step 11:** The `SCATTER` thread finally updates the active list in DRAM.

The barrierless asynchronous design also makes it easy to scale up to simultaneously processing multiple blocks with both highly paralleled accelerator and CPU. The scheduler simply pushes multiple tasks to the accelerator task queue at the same time and multiple accelerator PEs can process different blocks. The same is true for multiple CPU threads process different updated blocks by the accelerator.

The `GATHER` component of GraphABCD is specialized in a dataflow pipeline fashion [44] to boost its throughput as depicted on the right part of Fig. 2. The main computation pattern in `GATHER` function is a reduction over in-coming edges, where dependency on the partial result of the same destination vertex may lead to the stall of CPU or ASIC pipelines [12], especially for multiple-clock-cycle reductions. Our `GATHER` component resolves this dependency by adopting a dynamic dataflow design [44] using destination indices (instead of addresses for partial results) as dataflow tags. This means instead of reducing the partial sum with in-coming edges once at a time, edges can pair with each other sharing the same destination vertex and be sent to the reduction unit in an out-of-order way. Then these partial sums (of the same destination vertex) can pair together and eventually be reduced to one value for each vertex in a reduction tree manner. An on-chip scratchpad is used to store unpaired edges or partial sums temporarily. Our microarchitecture design is similar to the reduction unit in [49]. But instead of using the forwarding unit to forward the partial sum, we simply merge the partial sum stream back to the input in-coming edge stream. In this way, our design yields a consistent throughput regardless of the type and latency of the reductive operation.

GraphABCD's program decouples block scheduling with computation in the same way as the execution flow of BCD (Fig. 1). The CPU scheduler can be configured to either cyclic or priority block selection rule. For regular cyclic selection, the active list in DRAM (Fig. 2) is a bitvector. The active list implementation can be easily extended into a priority list to support priority block selection. The priority updates happen along with the `SCATTER` updates, where the CPU updates the gradient vector of the corresponding priority list entry using the updated vertex value block.

In GraphABCD, simple customized hardware modules (`GATHER`, `APPLY`) and software functions (`SCATTER`) are exposed to the end users as API. These modules and functions can be modified in order for the framework to adapt to different algorithms[5]. On the hardware side, GraphABCD offers a straightforward dataflow interface for customized logic. The customized components could be generated either by High-Level Synthesis tools or via integration of existing IPs.

V. Evaluation

In this section, we answer the following questions by presenting experimental results. (1) How does the BCD view of algorithm design options affect the convergence rate?

[5] Details regarding how to map graph algorithms to the GAS model have been described in previous works [10], [40].

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Graph Datasets Used for Evaluation

Graph	#vertices	#edges
Wikipedia Talk (WT) [21], [22]	2.39M	5.02M
Pokec (PS) [41]	1.63M	30.62M
LiveJournal (LJ) [4], [23]	4.85M	68.99M
Twitter (TW) [17]	41.65M	1.47B
SAC18 (SAC) [16]	105k users 49k movies	10.00M
MovieLens (MOL) [13]	283k users 54k movies	27.75M
Netflix (NF) [5]	480k users 17k movies	100.48M

(Sec. V-B) (2) How does GraphABCD compare to state-of-the-art CPU/ASIC graph analytic frameworks? (Sec. V-C) (3) How effective are the proposed techniques and optimizations in GraphABCD? (Sec. V-D)

A. Experimental Setup

Algorithm and Dataset: Three iterative graph algorithms, PageRank (PR), Single Source Shortest Path (SSSP) and Collaborative Filtering (CF), are run on 7 real-world graphs (in edgelist format) as summarized in Table I. Each algorithm is run until convergence 9 times and the median execution time is reported.

HARPv2 System: Our experiments are conducted on the Intel HARPv2 Platform, an experimental heterogeneous platform featuring in-package integration of a 2.4 GHz 14-core Broadwell Xeon processor and an Arria 10 FPGA. The FPGA can access data in the CPU LLC through two PCIe x8 and one QPI with a total bandwidth of 12.8GB/s. We instantiated 16 FPGA PEs and spawned 14 CPU threads. Clocked at a frequency of 200MHz, the FPGA's resource utilization for PR, SSSP, and CF are summarized in Table IV.

Baseline: A state-of-the-art GAS-modeled software framework GraphMat [40] and its ASIC implementation Graphicionado [12] are used for baseline comparison. GraphMat is executed with the same 14-core Xeon processor in HARPv2. The results of Graphicionado are reported at 1GHz under 4×DDR4-2133 17GB/s channel [12]. We observe that both GraphABCD and Graphicionado's performance are bounded by memory bandwidth. In order to fairly compare the ASIC baseline (Graphicionado) with our FPGA prototype (GraphABCD), we do our best effort comparison by reporting the projected performance results of Graphicionado (originally 68GB/s bandwidth) under a 5x smaller 12.8GB/s memory bandwidth budget, which is GraphABCD's available memory bandwidth[6].

B. Convergence Rate

In this subsection, we analyze the impact of different algorithm design options on **#_of_iterations** (Equation (1)) and test the effectiveness of BCD view. To ease the variation of Block Size, the software version of asynchronous GraphABCD framework is implemented, where SSSP and PageRank applications are evaluated. We select *cyclic* and *priority* scheduling with block sizes from 8 to 32768. A BSP baseline, i.e. block size = $|V|$, is implemented, to which all of the results are normalized. The $\#\_of\_iterations$ for every combination of **Block size** and **Block selection rule** are presented in Fig. 4.

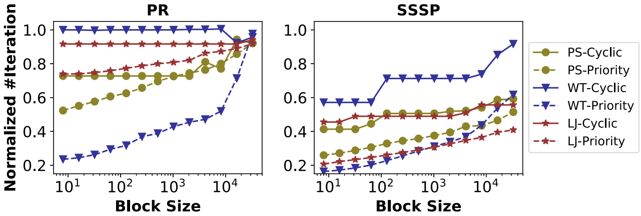


Fig. 4: Convergence rate (normalized to BSP) of PR and SSSP using cyclic and priority scheduling on Graph PS, WT and LJ. (lower is better)

With a smaller block size, both PR and SSSP converge faster. Consistent with the theory of BCD, a smaller block size results in a higher convergence rate ranging from 1.2x to 5x. We also observe that generally priority scheduling converges faster than cyclic scheduling by up to 5x. The results show the BCD-guided priority scheduling strategy effectively improves the convergence rate of all graph algorithms once they are framed in the BCD perspective. It confirms with previous works [29], [30], [43] which theoretically justify the superiority of priority scheduling for faster convergence. The advantage of priority scheduling is more significant when the block size is smaller. When the block size is 1, our proposed gradient estimation is the actual gradient, whereas priority selection is equivalent to cyclic selection when we have a single block.

Discussions: The experiments corroborate the two key insights suggested by BCD in Sec. III, which shows the effectiveness of applying BCD to graph domain to improve its convergence rate. 1) Small block size with priority block selection converges fastest. 2) Asynchronous BCD holds the same convergence properties as synchronous BCD.

C. Comparison to Previous Works

Table II summarizes execution time and throughput (in terms of Million Traversed Edges per Second or MTES) of GraphABCD, GraphMat, and Graphicionado. The two major columns 'execution time' and 'throughput' of Table II correspond to **runtime** and **runtime_per_iteration** of Equation (1), respectively. We further measure the convergence rate ($\#\_of\_iterations$) of GraphABCD and GraphMat and present the results in Table III.

GraphABCD is evaluated on four configurations (turn on/off priority scheduling, turn on/off hybrid execution) and the best result is reported. The execution time of Graphicionado is extracted from the original paper [12] when GraphABCD and Graphicionado execute the same algorithm on the same

[6] The frequency difference between Graphicionado and GraphABCD is about the same (∼5x) as the memory bandwidth. This suggests that if we scale up both frequency and bandwidth of GraphABCD by 5x, its performance is still bottlenecked by memory bandwidth, as is Graphicionado. So our normalization to Graphcionado's result won't change its bottleneck.

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Execution Time and Throughput of GraphABCD, GraphMat, and ASIC (Graphicionado)

App	Graph	Execution Time (s)			Throughput (MTES)	
		GraphABCD	GraphMat	ASIC [a]	GraphABCD	GraphMat
PR	WT	**0.123**	0.255		531	630
	PS	**0.619**	1.420		704	1078
	LJ	**1.577**	3.997	9.993	954	1001
	TW	**42.810**	108.015	93.116	857	1034
SSSP	WT	0.034	**0.026**		540	442
	PS	0.280	**0.262**		1328	861
	LJ	**0.652**	0.717	1.195	1120	555
	TW	**8.367**	9.556	23.890	2059	791
CF	SAC	**0.206**	0.556		388	719
	MOL	**0.853**	2.092		390	517
	NF	**2.090**	6.832	9.760	385	397

[a] Results of ASIC is derived by scaling the results of Graphicionado from 68GB/s bandwidth down to GraphABCD's 12.8GB/s for fair comparison.

graph (PR, BFS on graph LJ, TW and CF on graph NF). GraphABCD's priority and cyclic scheduling versions are abbreviated as 'Priority' and 'Cyclic'. Since GraphMat and Graphicionado share the same algorithm design options, they have the same convergence rate and are reported in the same column in '$\#\_of\_iterations$'.

Comparison with GraphMat: GraphABCD achieves speedups of 2.1x-2.5x and 2.5x-3.3x over GraphMat on PageRank and Collaborative Filtering, respectively. But it performs worse than GraphMat on SSSP in some cases (0.76x-1.14x). GraphABCD significantly improves convergence rate over GraphMat through the novel BCD view of iterative graph algorithms. In Table III, GraphABCD's PR features a 72%-76% reduction on $\#\_of\_iterations$ compared to GraphMat's PR, which is attributed to the suitable algorithm design choices guided by BCD. As an intrinsic optimization problem, the convergence rate of CF is better illustrated using RMSE[7]. We plot the RMSE change (lower is better) of GraphABCD's priority and cyclic version as well as GraphMat's in Fig. 5. GraphABCD can achieve a much better quality of results in even fewer iterations (20 iterations, RMSE=1.04) compared with GraphMat (60 iterations, RMSE=1.34). This is because GraphMat adopts the algorithm design option of *block size* $|V|$ and GraphABCD has a much smaller block size. These observations conform to BCD's statement that smaller block size leads to faster convergence.

However, the $\#\_of\_iterations$ of GraphABCD's SSSP is 1.5x-1.8x larger than that of GraphMat, which primarily leads to runtime reduction on GraphMat. Because GraphMat deviates from its BSP model in SSSP and uses an active vertex list to filter unnecessary computation, which in fact reduces its block size. Thus the improvement of GraphMat in SSSP comes from the better convergence rate. This observation further justifies the necessity of using BCD view to improve the convergence rate of iterative graph algorithms.

In terms of $runtime\_per\_iteration$, GraphMat's throughput surpasses that of GraphABCD in some cases, because CPU has larger memory bandwidth (58GB/s vs 12.8GB/s) and are less likely to be bounded by it.

App	Graph	$\#\_of\_iterations$		
		Priority	Cyclic	GraphMat
PR	WT	8.03	13.02	32
	PS	14.20	15.88	50
	LJ	13.78	21.79	58
SSSP	WT	3.66	4.01	2.29
	PS	10.71	12.12	7.37
	LJ	10.58	11.59	5.77
CF	SAC	4	4	27
	MOL	6	6	39
	NF	4	4	37

TABLE III: Convergence Rate

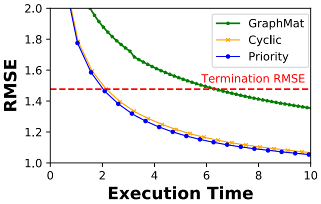


Fig. 5: RMSE of CF

Comparison with Graphicionado: We also compare GraphABCD with Graphicionado, the ASIC implementation of GraphMat. GraphABCD is 4.3x, 2.3x and 4.8x faster than Graphicionado on PR, SSSP, and CF on average. Apart from the acceleration over GraphMat, GraphABCD's further performance improvement over Graphicionado can be attributed to its efficient PE design and less inter-PE coordination. GraphABCD's fully-pipelined dataflow design of GATHER module decreases the possibility of stalling the pipeline in the PE compared to the atomic GATHER of Graphicionado. Also, GraphABCD's barrierless and lock-free asynchronous design prevents PEs from stalling due to frequent global memory barrier and coordination in Graphicionado.

GraphABCD consumes 2.69MB FPGA BRAM and 35MB CPU LLC while Graphicionado uses 64MB-256MB on-chip eDRAM depending on the graphs and applications. The reason why only a small buffer is required on the reconfigurable logic in GraphABCD is the *pull-push* vertex operator, thus the selected edge blocks are simply streamed onto the FPGA from LLC, requiring much smaller on-chip buffer. On the contrary, Graphicionado uses a *push* design, which demands a large on-chip scratchpad storing all of the vertex values, in order to amortize the random vertex access overhead (random access array V[] in last line of Fig. 3(b)).

D. GraphABCD Speedup Breakdown

Comparison with CPU-based GraphABCD: To evaluate the efficiency of the FPGA-accelerated GATHER-APPLY module, we implement the same module in software and construct the software version of GraphABCD. We fuse the GATHER-APPLY function with SCATTER function to reduce data movement and inter-thread coordination to further optimize the software baseline while still retaining its asynchronicity.

The execution time of FPGA-accelerated GraphABCD (with hybrid execution and cyclic/priority block selection) is compared with software GraphABCD (cyclic/priority block selection) in Fig. 6. Even with kernel fusion in software baseline, FPGA-accelerated GraphABCD achieves a speedup of 1.2x-9.2x and 3.4x on average over the baseline. The efficiency of FPGA acceleration is two-fold: 1) The customized memory subsystem on FPGA ensures all memory accesses between CPU and FPGA are sequential. Even without the cache memory hierarchy, the FPGA can still do prefetches and overlap communication with computation to reduce memory access overhead. 2) The fully-pipelined high throughput GATHER

[7] Root-Mean-Square Error, it is the square root of the CF objective function divided by number of edges. The decrease rate of RMSE is often used to demonstrate the convergence rate of CF algorithm.

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

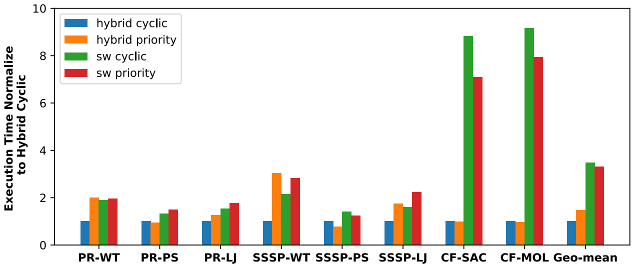


Fig. 6: Effect of Hardware Acceleration (lower is better)

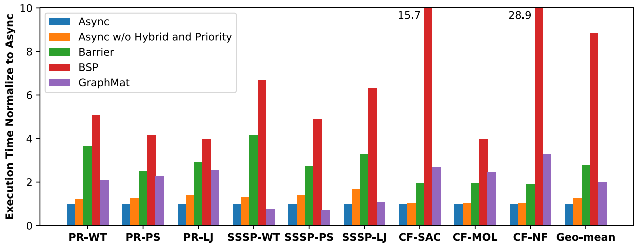


Fig. 7: Speedup Breakdown (lower is better)

	PR	SSSP	CF
Register	9%	6%	14%
ALM	26%	25%	31%
BRAM	34%	34%	41%
DSP	12%	0%	54%

TABLE IV: Resorce Utilization

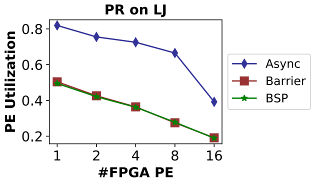


Fig. 8: FPGA Utilization

module design significantly amortizes the reduction overhead performed in CPU.

Effect of Asynchronicity over BSP: We evaluate the key feature of GraphABCD, asynchronous execution, over synchronous BSP model. We build two additional baselines of GraphABCD denote as 'Barrier' and 'BSP' and the asynchronous GraphABCD is abbreviated as 'Async'. For the 'Barrier' baseline, we add memory barriers after the completion of each block's GAS vertex processing. We add a global memory barrier every iteration in our 'BSP' baseline.

Fig. 7 shows the execution time of each baseline. Compared with 'Barrier', GraphABCD achieves 1.9x-4.2x acceleration due to its barrierless and lock-free asynchronous design. This shows adding barriers to coordinate within heterogeneous and distributed systems comes at a great loss on performance. Note that the 'Barrier' baseline achieves a similar convergence rate as 'Async' which is consistent with the argument that asynchronous BCD holds the same convergence properties as synchronous one (Sec. III-D). Apart from the overhead of barrier and other synchronization primitives, 'BSP' slows down the runtime by 1.4x-15.2x. This is because GraphABCD and 'Barrier' adopt a smaller block size compared to $|V|$ in 'BSP'. The slow down mainly comes from convergence rate.

Asynchronicity also improves FPGA PE utilization. Fig. 8 plots the FPGA PE utilization as we simultaneously decrease the number of FPGA PEs and CPU threads from 16 to 1. Asynchronous GraphABCD improves PE utilization by 1.6x-2.4x over synchronous GraphABCD. This is because our barrierless asynchronous design avoids PEs from waiting until all the PEs finish the processing and allows them to execute non-blockingly. Note that the PE utilization decreases sharply when PE number is increased from 8 to 16. This is because memory bandwidth reaches its limits and PEs are waiting for data to arrive (we will elaborate it below). Furthermore, PE utilization in 'Barrier' and 'BSP' is similar, which also suggests their performance difference largely attributes to different convergence rates caused by block size.

Asynchronous execution drastically simplifies coordination protocol and improves work efficiency of the execution units. These two advantages of asynchronicity allow GraphABCD scaling out to heterogeneous distributed systems efficiently.

Effect of Memory System Design: We now evaluate GraphABCD's memory subsystem design by breaking down the memory traffic between the CPU and FPGA for 3 algorithms. We also plot the CPU-FPGA memory bus utilization w.r.t FPGA PE number change while fixing CPU threads to 14. In Fig. 9(a), GraphABCD utilizes 98%, 99% and 80% of the memory bandwidth and the reads and writes are fully sequential. This is because GraphABCD's *pull-push* design and disjoint block partition effectively make all memory accesses from FPGA continuous. This design maximumly exploits the potential of the CPU-FPGA memory bus even without an explicit memory hierarchy on FPGA and allows prefetching due to the better memory locality. Also, the memory read dominates memory traffic because in pull-push design in-coming edge reads ($|E|$) are larger than updated vertex block writes ($|V|$). This makes the coordination in the heterogeneous system easier since reads are generally less intrusive to memory system than writes, which also justifies our choice of pull-push.

Fig. 9(b) offers insights on how bandwidth utilization changes w.r.t number of on-chip PEs. Bandwidth utilization steadily increases as it reaches the upper bound of 100% at 8 PEs and stays fully used with 16 PEs. This not only suggests that memory bandwidth is fully utilized in GraphABCD but also indicates that **memory bandwidth between CPU and FPGA in HARPv2 platform is the bottleneck of the current system**. Thus the PE utilization drops dramatically in Fig. 8 since there isn't enough bandwidth to feed 16 PEs. The extra PEs are stalled for the data to arrive.

Scalability and Effect of Hybrid Execution: We evaluate the scalability of GraphABCD as well as the effect of Hybrid Execution optimization. We first fix the CPU thread to 14 and increase FPGA PE number from 1 to 16 and measure the execution time (with hybrid execution turning on and off separately). Then we do the reverse by fixing the FPGA PE to 16 and increase CPU thread from 1 to 14.

We first analyze the scalability without hybrid execution

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

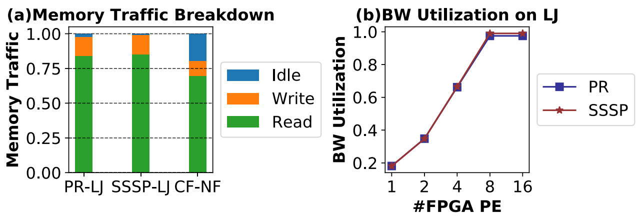


Fig. 9: Memory Traffic

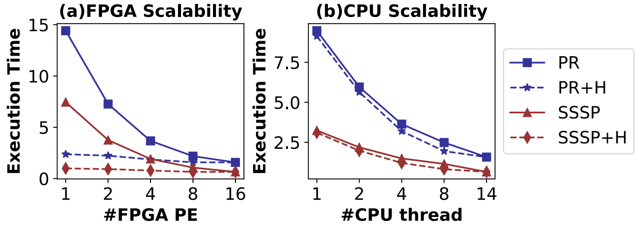


Fig. 10: Scalability on LJ (w/ and w/o Hybrid Execution)

optimization. As shown in Fig. 10(a) and (b), the execution time decreases linearly when the parallelism increases. The decrease slows down when FPGA PE number exceeds 8, which also confirms that the system performance is bounded by CPU-FPGA memory bandwidth. Also, the system (without hybrid execution) is more sensitive to FPGA PE number change than CPU thread change. This suggests FPGA is responsible for most computation in the system and the 14 CPU threads are under-utilized.

Hybrid Execution optimization is proposed to improve CPU thread utilization. In Fig. 7, Hybrid Execution improves the performance by up to 66% in SSSP-LJ and by 24% on average. This is because hybrid execution further exploits the under-utilized CPU threads in the system and adds available CPU threads to the heterogeneous processing system. In Fig. 10(a), after adding the hybrid execution optimization, the execution time is less sensitive to FPGA PE number decrease, which suggests that the CPU can serve as extra computation resources (weaker version of FPGA PE) to compensate for the FPGA PE number decrease. Therefore, hybrid execution can fully leverage the available CPU resources and scale GraphABCD to all computation resources in the heterogeneous HARPv2 system.

Effect of Priority Scheduling: We evaluate the priority scheduling optimization implemented in GraphABCD. In Table. III, Priority scheduling can cut down 11%-38% and 8%-12% $\#\_of\_iterations$ compared with cyclic block selection in PR and SSSP, respectively. Priority scheduling also helps to reduce RMSE faster by 10% in CF. Though the increased convergence rate, the cases where priority block selection runs slower than cyclic are small graphs like WT, where the improvements in convergence rate cannot offset the runtime overhead imposed by priority scheduling. In other cases, priority scheduling can speed up execution by 3%-28% over cyclic scheduling. Nevertheless, it also experimentally proves that the BCD insights that priority scheduling leads to faster convergence is correct and the BCD-guided priority scheduling strategy works well for graph algorithms.

VI. Related Work

A. Convergence of Graph Algorithms

Prior systems address the convergence rate issue mostly in a case-by-case manner. GraphLab [24] and PowerGraph [10] implement synchronous and asynchronous engines and experimentally show that asynchronous execution improves convergence rate. Galois designs a priority-based PageRank algorithm [46] specialized for the push vertex operator. Wonderland [52] proposes abstraction-guided priority execution that leverages Δ-stepping variant of traversal-based algorithm instead of Bellman-Ford variant. Therefore, a systematic approach to analyzing and optimizing the convergence rate of iterative graph algorithms is needed.

B. Asynchronous Processing

Asynchronous model is proposed as an alternative to the BSP model to avoid the requirement of global memory barriers. However, existing asynchronous solutions still use synchronization primitives such as barrier and fine-grained locking as coordination mechanisms within the system, which stumbles the scaling out to heterogeneous and distributed systems. Distributed GraphLab [24], PowerGraph [10] and D-Galois [8], [27] impose fine-grained locking on each vertex whenever conflict memory accesses occur. An ASIC framework [33] relies on a centralized GRC (Global Rank Counter) for coordination, which will bottleneck multi-chip designs. [25] inserts a memory fence before the computation of each subgraph and flushes the memory write buffer. GraphQ [60] exploits asynchronicity among multiple memory blocks in the PIM system. But global barriers are inserted every iteration.

C. Optimizations on Memory Bottleneck

Lots of efforts have been made to alleviate memory bottleneck in graph analytics, which is the performance bottleneck of GraphABCD. Graphit [54] is a domain specific language for graph analytics which allows customizing data structure for a more compact data representation. Grazelle [11] introduces Vector-Sparse graph representation to properly align memory accesses. Ligra+ [37] applies compression schemes on graphs. HyGCN [48] leverages window slicing and shrinking to reduce redundant memory accesses. GraphABCD can integrate those optimizations to have more compact graph representation and less memory traffic to address the bandwidth bottleneck.

VII. Conclusions

In this paper, we present the Block Coordinate Descent (BCD) view of iterative graph algorithms. Then we propose GraphABCD, a *heterogeneous* graph analytic framework with *asynchronous* block coordinate descent. The BCD view of graph algorithms guides us to effectively improve the convergence rate of GraphABCD framework. GraphABCD features

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

barrierless and lock-free asynchronous execution which enables computation to scale out to heterogeneous distributed platforms. GraphABCD's priority scheduling, memory subsystem design, and hybrid execution further exploit the heterogeneous system. Experiments on a prototype show that GraphABCD achieves geo-mean speedups of 4.8x and 2.0x in convergence rate and execution time over GraphMat, a state-of-the-art framework.

Acknowledgments

We thank Shaolei Du, Daniel Sanchez, the anonymous reviewers and shepherd for insightful discussions and feedback. This work is supported in part by the National Natural Science Foundation of China (Grant No.61834002), and in part by the National Key R&D Program of China (Grant No. 2018YFB2202101). The presented HARPv2 results were obtained on resources hosted at the Paderborn Center for Parallel Computing (PC^2) in the Intel Hardware Accelerator Research Program (HARP).

References

[1] 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, Portland, OR, USA, June 13-17, 2015*, 2015, pp. 105–117.

[2] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Clip: A disk I/O focused parallel out-of-core graph processing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 45–62, 2019.

[3] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, 2017, pp. 375–388.

[4] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, 2006, pp. 44–54.

[5] J. Bennett, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007. New York, NY, USA., 2007, p. 35.

[6] A. Ching, H. Choi, J. Homan, C. Kunz, O. O'Malley, J. Mannix, D. Ryaboy, C. Martella, R. Shaposhnik, S. Schelter, E. Koontz, A. Presta, E. Reisman, M. Kabiljo, N. Joffe, S. Edunov, P. Kumar, I. Kabiljo, and H. Eslami, "Apache giraph," 2013. [Online]. Available: http://giraph.apache.org/

[7] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017, pp. 217–226.

[8] R. Dathathri, G. Gill, L. Hoang, H. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, 2018, pp. 752–768.

[9] N. Engelhardt and H. K. So, "Gravf: A vertex-centric distributed graph processing framework on fpgas," in *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, 2016, pp. 1–4.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 17–30.

[11] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, A. Krall and T. R. Gross, Eds. ACM, 2018, pp. 246–260.

[12] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 56:1–56:13.

[13] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *TiiS*, vol. 5, no. 4, pp. 19:1–19:19, 2016.

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, 2011, pp. 267–276.

[15] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, 2014, pp. 239–252.

[16] D. Kotkov, J. A. Konstan, Q. Zhao, and J. Veijalainen, "Investigating serendipity in recommender systems based on real user feedback," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, 2018, pp. 1341–1350.

[17] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 591–600.

[18] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 31–46.

[19] K. Lakhotia, R. Kannan, S. Pati, and V. K. Prasanna, "GPOP: a cache and memory-efficient framework for graph processing over partitions," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, 2019, pp. 393–394.

[20] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016.

[21] J. Leskovec, D. P. Huttenlocher, and J. M. Kleinberg, "Predicting positive and negative links in online social networks," in *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, 2010, pp. 641–650.

[22] ——, "Signed networks in social media," in *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*. ACM, 2010, pp. 1361–1370.

[23] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *arXiv e-prints*, p. arXiv:0810.1355, Oct. 2008.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[25] L. Luo and Y. Liu, "Improving parallel efficiency for asynchronous graph analytics using gauss-seidel-based matrix computation," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, 2019.

[26] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 135–146.

[27] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 456–471.

[28] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An FPGA framework for vertex-centric graph computation," in *22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014, Boston, MA, USA, May 11-13, 2014*, 2014, pp. 25–28.

[29] J. Nutini, I. Laradji, and M. Schmidt, "Let's Make Block Coordinate Descent Go Fast: Faster Greedy Rules, Message-Passing, Active-Set Complexity, and Superlinear Convergence," *ArXiv e-prints*, Dec. 2017.

[30] J. Nutini, M. W. Schmidt, I. H. Laradji, M. P. Friedlander, and H. A. Koepke, "Coordinate descent converges faster with the gauss-southwell rule than random selection," in *Proceedings of the 32nd International*

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.

Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, 2015, pp. 1632–1641.

[31] T. Oguntebi and K. Olukotun, “Graphops: A dataflow library for graph analytics acceleration,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, 2016, pp. 111–117.

[32] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, “A reconfigurable computing system based on a cache-coherent fabric,” in *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011*, 2011, pp. 80–85.

[33] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. M. Burns, and Ö. Öztürk, “Energy efficient architecture for graph analytics accelerators,” in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 166–177.

[34] B. Recht, C. Ré, S. J. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, Eds., 2011, pp. 693–701.

[35] A. H. N. Sabet, J. Qiu, and Z. Zhao, “Tigr: Transforming irregular graphs for gpu-friendly graph processing,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, 2018, pp. 622–636.

[36] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, Shenzhen, China, February 23-27, 2013*, 2013, pp. 135–146.

[37] J. Shun, L. Dhulipala, and G. E. Blelloch, “Smaller and faster: Parallel processing of compressed graphs with ligra+,” in *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, Eds. IEEE, 2015, pp. 403–412.

[38] P. Stutz, A. Bernstein, and W. W. Cohen, “Signal/collect: Graph algorithms for the (semantic) web,” in *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, 2010, pp. 764–780.

[39] T. Sun, R. Hannah, and W. Yin, “Asynchronous Coordinate Descent Under More Realistic Assumption,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. USA: Curran Associates Inc., 2017, pp. 6183–6191, event-place: Long Beach, California, USA.

[40] N. Sundaram, N. Satish, M. M. A. Patwary, S. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *PVLDB*, vol. 8, no. 11, pp. 1214–1225, 2015.

[41] L. Takac and M. Zabovsky, “Data analysis in public social networks,” *International Scientific Conference and International Workshop Present Day Trends of Innovations*, pp. 1–6, 01 2012.

[42] S. Tasci and M. Demirbas, “Giraphx: Parallel yet serializable large-scale graph processing,” in *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, 2013, pp. 458–469.

[43] P. Tseng, “Convergence of a block coordinate descent method for nondifferentiable minimization,” *Journal of optimization theory and applications*, vol. 109, no. 3, pp. 475–494, 2001.

[44] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 2014, pp. 205–216.

[45] Y. Wang, S. Baxter, and J. D. Owens, “Mini-gunrock: A lightweight graph analytics framework on the GPU,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, 2017, pp. 616–626.

[46] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, “Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned,” in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, vol. 9233, pp. 438–450.

[47] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, “Pregel algorithms for graph connectivity problems with performance guarantees,” *PVLDB*, vol. 7, no. 14, pp. 1821–1832, 2014.

[48] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Hygcn: A GCN accelerator with hybrid architecture,” *CoRR*, vol. abs/2001.02514, 2020.

[49] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, 2019, pp. 615–628.

[50] E. Yoneki and A. Roy, “Scale-up graph processing: a storage-centric view,” in *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, 2013, p. 8.

[51] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, “Exploring the hidden dimension in graph processing,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 285–300.

[52] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, “Wonderland: A novel abstraction-based out-of-core graph processing system,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, 2018, pp. 608–621.

[53] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “Graphp: Reducing communication for pim-based graph processing with efficient data partition,” in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 544–557.

[54] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, “Graphit: a high-performance graph DSL,” *PACMPL*, vol. 2, no. OOPSLA, pp. 121:1–121:30, 2018.

[55] S. Zhou, C. Chelmis, and V. K. Prasanna, “High-throughput and energy-efficient graph processing on FPGA,” in *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*, 2016, pp. 103–110.

[56] S. Zhou, R. Kannan, Y. Min, and V. K. Prasanna, “FASTCF: fpga-based accelerator for stochastic-gradient-descent-based collaborative filtering,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, 2018, pp. 259–268.

[57] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, “An FPGA framework for edge-centric graph processing,” in *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018*, 2018, pp. 69–77.

[58] S. Zhou and V. K. Prasanna, “Accelerating graph analytics on CPU-FPGA heterogeneous platform,” in *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, 2017, pp. 137–144.

[59] X. Zhu, W. Chen, W. Zheng, and X. Ma, “Gemini: A computation-centric distributed graph processing system,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 301–316.

[60] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, “Graphq: Scalable pim-based graph processing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, 2019, pp. 712–725.

Authorized licensed use limited to: MIT Libraries. Downloaded on January 17,2021 at 01:06:55 UTC from IEEE Xplore. Restrictions apply.