The Journal of Supercomputing (2018) 74:6135–6155
https://doi.org/10.1007/s11227-018-2525-0

An adaptive breadth-first search algorithm on integrated architectures

Feng Zhang[1,2] · Heng Lin[3] · Jidong Zhai[3] · Jie Cheng[4] · Dingyi Xiang[4] · Jizhong Li[4] · Yunpeng Chai[1,2] · Xiaoyong Du[1,2]

Published online: 11 August 2018

Abstract
In the big data era, graph applications are becoming increasingly important for data analysis. Breadth-first search (BFS) is one of the most representative algorithms; therefore, accelerating BFS using graphics processing units (GPUs) is a hot research topic. However, due to their random data access pattern, it is difficult to take full advantage of the power of GPUs. Recently, hardware designers have integrated CPUs and GPUs on the same chip, allowing both devices to share physical memory, which provides the convenience of switching between CPUs and GPUs with little cost. BFS processing can be divided into several levels, and various traversal orders can be used at each level. Using different traversal orders on different devices (CPUs or GPUs) results in diverse performances. Thus, the challenge in using BFS on integrated architectures is how to select the traversal order and the device for each level. Previous works have failed to address this problem effectively. In this study, we propose an adaptive performance model that automatically finds a suitable traversal order and device for each level. We evaluated our method on Graph500, where it not only shows the best energy efficiency but also achieves a giga-traversed edges per second (GTEPS) performance of approximately 2.1 GTEPS, which is a 2.3 × speed improvement over the state-of-the-art BFS on integrated architectures.

Keywords Breadth-first search · Integrated architectures · Performance · Energy efficiency · Modeling

✉ Jidong Zhai
zhaijidong@tsinghua.edu.cn

1 Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China, Beijing, China

2 School of Information, Renmin University of China, Beijing, China

3 Department of Computer Science and Technology, Tsinghua University, Beijing, China

4 Huawei Technologies Co., Ltd., Shenzhen, China

1 Introduction

Researchers have found that many practical problems can be represented by graphs; consequently, graph applications have become increasingly important in various types of data analysis. Among graph analysis algorithms, breadth-first search (BFS) [11] is one of the most representative algorithms and functions as an important building block for other graph algorithms such as single-source shortest path (SSSP) [6], weakly connected components (WCC) [8], and page rank [39]. Moreover, BFS is currently the ranking benchmark of Graph500 [34], an alternative to Top500 [13], which is a big data analysis benchmark.

Researchers use GPUs to accelerate various applications, including BFS, because GPUs have massive numbers of lightweight computation cores, resulting in much higher computational power. Many linear algebra problems, such as matrix multiplication, benefit from GPU's computing capacity. The applications that benefit the most from GPUs are computation-intensive applications that typically have regular memory access patterns. However, BFS is memory bound and has an irregular memory access pattern, which makes it difficult to fully capitalize on the power of GPUs. The vertices allocated from a graph to GPU threads possess different numbers of edges; hence, the workload for each thread varies. The threads in GPUs process data in a lock-step manner, which means that a group of threads must execute the same instruction simultaneously. Consequently, the BFS algorithm has serious load imbalance problems when executed on GPUs. Even worse, GPUs have an independent on-chip memory: the data they process must be copied through the PCIe bus from main memory. This memory copying overhead becomes computationally expensive when the processed graph is large and cannot fit into the available GPU memory.

Since 2011, hardware vendors have released an increasing number of integrated architectures, such as AMD's A-series integrated architecture APU [7], Intel's Ivy Bridge and Haswell [17], and Nvidia's Tegra K1 [36]. Integrating GPUs and CPUs on the same chip presents a new opportunity for accelerating the BFS algorithm. Using this integrated architecture makes it possible to switch between CPUs and GPUs with little cost. BFS processing can be divided into many levels. At each level, we can use a top-down or bottom-up traversal order. Different traversal orders with different devices (CPUs or GPUs) present different performances. Thus, for an input graph, the BFS challenge on integrated architectures involves selecting a suitable traversal order and device for each level.

Several prior studies concerning BFS execution on heterogeneous platforms have been conducted. Beamer et al. [4] first proposed BFS in the bottom-up direction instead of the conventional top-down direction. This approach can significantly reduce the memory requirements for accessing edges in large frontiers. They also provided a simple method for switching an algorithm between the top-down and bottom-up processing. Daga et al. [12] were the first to implement hybrid top-down and bottom-up algorithms on integrated architectures. This approach is the state of the art for BFS processing on integrated architectures. In these approaches, CPUs are used for top-down traversal, while GPUs are used for bottom-up traversal. However, the previous methods do not select the proper traversal orders and devices effectively. In contrast to the previous studies, we provide a modeling method to estimate the performance of

top-down and bottom-up algorithms on both CPUs and GPUs that can select the most appropriate strategy. Moreover, experiments show that due to OpenCL space allocation limitations by vendors, we cannot allocate a large graph directly into memory. The previous BFS implementations on integrated architectures [12,50,51] do not consider this factor either. We provide a specific design that enables our BFS implementation to process large graphs.

In this paper, we analyze the factors that influence the performance of BFS on integrated architectures and develop an adaptive performance model that considers these factors. The influencing factors include search direction, device selection, data layout, large graph allocation, and vendor-specific optimizations, making it difficult to select the optimal combination for traversal orders and executing devices. Our adaptive performance model estimates the performance for both top-down and bottom-up algorithms on GPUs and CPUs with different optimizations and automatically selects the suitable parameters. We evaluated our BFS method on a Graph500 benchmark. Experiments show that our BFS not only exhibits the best energy efficiency but also achieves 2.1 GTEPS—a 2.3 × speed improvement compared to the state-of-the-art BFS algorithm on integrated architectures.

In summary, this paper makes the following contributions.

1. We provide a comprehensive performance model that can adaptively choose the optimal algorithms and devices with proper optimizations.
2. We enable our BFS implementation to process large graphs that previous implementations cannot handle.
3. The experiments evaluating our BFS implementation from a power aspect exhibit energy-efficient results.

The remainder of this paper is organized as follows: In Sect. 2, we review integrated architectures and general BFS algorithms. Section 3 provides detailed descriptions of our BFS algorithms. Section 4 shows the optimizations applied to our BFS implementation. Section 5 presents evaluation results and a comparison between our implementation and previous BFS implementations for integrated architectures. Section 6 discusses related works, and Sect. 7 concludes the paper.

2 Background

In this section, we present background information concerning integrated architectures and the BFS algorithm.

2.1 Integrated architectures

We implement the BFS algorithm on integrated architectures, which integrate heterogeneous accelerators and CPUs on the same chip. In our experiments, we use the AMD integrated architecture APU [34] as an experimental platform to exhibit our BFS algorithm for integrated architectures. Figure 1 shows a general view of CPU–GPU integrated architectures, in which both the CPU and GPU are able to access the same physical memory. In contrast to CPUs, GPUs have physical local memory (OpenCL

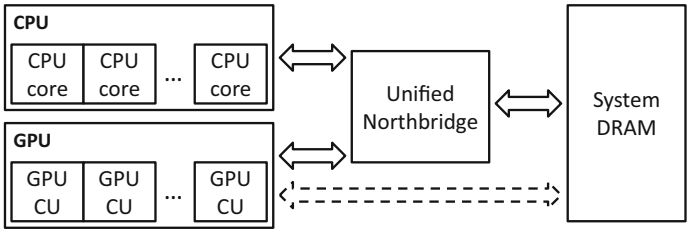


Fig. 1 A general view of integrated architectures

terminology [43]), which has a faster access speed than does global memory and is more controllable. An integrated design has two benefits: (1) the CPU does not need to transfer data to the GPU via the PCIe bus; hence, the CPU and GPU can have more cooperation; (2) the GPU can process a larger dataset without the GPU memory size limitations found in discrete CPU–GPU designs. Therefore, on integrated architectures, we have the opportunity to closely combine the use of both GPU and CPU devices to accelerate BFS.

2.2 BFS algorithm

BFS is an algorithm for graph traversal from the root vertex, v_r, to other vertices, as shown in Algorithm 1. It traverses vertices based on their distances from the root vertex. Hence, according to the nature of breadth-first, BFS computing is synchronous by levels, and BFS traversal is divided into computation at different levels. Each level has a frontier, which is the *Curr* in Algorithm 1. The frontier includes the active vertices at the current level. During computation at different levels, the current frontier generates the next frontier, which is the *Next* in Algorithm 1. BFS begins with a root vertex, v_r, and terminates when the frontier is empty. In the parallel version of BFS, the parallel region is the computation that occurs at each level. This parallelism is based on the active vertices or edges: after the computation for each level completes, a synchronization process is required.

BFS shares characteristics with many other graph algorithms, such as graph coloring [18] and connected component [14]. We summarize these characteristics as follows.

Frequent data access BFS is a memory-bound algorithm involving few compute operations. Instead, most operations in BFS read and update data structures with position information.

Random data access Each vertex may be connected to any other vertices of the graph; thus, BFS traversal may access locations in any part of the graph. Hence, the data access pattern is random, which results in high memory access pressure.

Data dependence In BFS, the memory access pattern depends on the shape of the input graph, which is unpredictable. The vertex access order cannot be obtained until the graph is loaded into memory. Hence, conventional memory optimization techniques, such as prefetching, are invalid for BFS.

Algorithm 1: Basic BFS algorithm

Input:
$G = (V, E)$: graph representation
v_r: root vertex
Output:
Prt: parent map
Data:
$Curr$: vertices in the current level
$Next$: vertices in the next level
Vst: ever visited vertices

```
1  Prt(:) ← −1, Prt(v_r) ← v_r
2  Curr ← ∅
3  Next ← {v_r}
4  Vst ← {v_r}
5  while Next ≠ ∅ do
6  |  Curr ← Next
7  |  Next ← ∅
8  |  for u ∈ {v ∈ Curr, (v, u) ∈ E, u ∉ Vst} do
9  |  |  Next ← Next ∪ {u}
10 |  |  Vst ← Vst ∪ {u}
11 |  |  Prt(u) ← v
12 |  end
13 end
```

Nonuniform data distribution The vertices of the input graph may follow a power law distribution; hence, the number of edges for each vertex is not well balanced. The unequal edge degree, which may cause load imbalance problems, is a critical issue for BFS parallelism.

All these characteristics distinguish BFS from other algorithms with regular memory access patterns. Therefore, we shall fully utilize the integrated architecture features to address these challenging characteristics in both the design and implementation of our BFS algorithm.

3 Algorithm

Previous BFS studies [4,12] use two algorithm directional strategies: top-down and bottom-up; however, these have different features and target different situations. Based on the characterizations of BFS in Sect. 2.2, we provide hybrid top-down and bottom-up algorithms in our BFS implementation and switch between these two algorithms based on performance modeling of the CPUs and GPUs in integrated architectures.

3.1 Overview

The most challenging part of Algorithm 1 involves deciding how to traverse the vertices and edges (line 8 in Algorithm 1). There are three considerations: (1) the vertices in the frontier, $u \in Curr$, (2) the edges whose starting vertices belong to the frontier, $(u, v) \in E$, and (3) the vertices not visited, $u \notin Vst$.

We can choose top-down traversal or bottom-up traversal for each level during BFS processing. We show our hybrid BFS algorithm in Algorithm 2. Note that previous studies for other problems, such as DFS [20], also use similar techniques. The function *usingTopDown*() in Algorithm 2 uses the proposed adaptive model (detailed in Sect. 3.2) to choose among different strategies. The top-down traversal is shown in lines 15–24, and the bottom-up traversal is shown in lines 25–35. The top-down traversal first scans all the vertices in the frontier, iterates through their edges, and then visits the connected vertices: this sequence processes the three considerations in the sequence (1), (2), and (3). It iterates every neighbor of the vertices in the frontier and then updates the related data structures if that neighbor has not previously been visited. In contrast to top-down traversal, bottom-up traversal first scans the vertices not yet visited and then verifies whether they connect to the vertices in the frontier. Consequently, bottom-up processing for the three conditions occurs in the sequence (3), (2), and (1).

We provide a performance model (detailed in Sect. 3.2) to choose between these two strategies. Top-down traversal is the most common BFS algorithm, but it is not suitable for all situations. As stated in Sect. 2.2, the vertices of the input graph may follow a power-law distribution, which implies that the frontier size of the middle levels during the BFS traversal can be very large. At these levels, the top-down method must scan all the edges whose source vertices are in the frontier but the destination vertices of these edges can be any vertices of the graph. Because of the large frontier size, some vertices may be visited multiple times, which is unnecessary. In this situation, bottom-up traversal is a better choice because it first checks whether the unvisited vertices have a neighbor in the frontier. Because we do not need to check the other neighbors of a given vertex if a neighbor is found in the frontier, this bottom-up method substantially reduces the number of memory accesses. However, note that bottom-up is also not suitable for all situations. When the number of unvisited vertices is large, and the frontier size is small, the top-down method is more efficient than the bottom-up method. Our performance model combines the bottom-up and top-down methods and estimates the potential number of memory accesses for each method. Then, we consider switching between the different methods and between the CPU and GPU devices at runtime. Moreover, because BFS has low computational intensity, we find that running CPUs and GPUs simultaneously does not result in further performance improvements. Hence, we do not provide a strategy for running CPUs and GPUs simultaneously.

3.2 Adaptive model

As stated in Sect. 3.1, we need to build an adaptive performance model to choose between the different algorithms (top-down and bottom-up) and between the different devices (CPUs and GPUs) during BFS traversal. We denote all vertices as V_{all} and use v_{all} to represent the number of vertices, $|V_{all}|$. Similarly, we use V_{left} to denote the vertices that remain to be processed and use v_{left} to represent their number. We use $V_{frontier}$ to denote the vertices in the frontier and $v_{frontier}$ to represent their number. Table 1 shows an analysis of the following factors.

Algorithm 2: Hybrid BFS algorithm

```
Input:
    G = (V, E): graph representation
    v_r: root vertex
Output:
    Prt: parent map
Data:
    Curr: vertices in the current level
    Next: vertices in the next level
    Vst: ever visited vertices
1  Prt(:) ← −1, Prt(v_r) ← v_r
2  Curr ← ∅
3  Next ← {v_r}
4  Vst ← {v_r}
5  while Next ≠ ∅ do
6      Curr ← Next
7      Next ← ∅
8      if usingTopDown(Curr, Next, Vst) then
9          TopDown()
10     end
11     else
12         BottomUp()
13     end
14 end
15 Function TopDown()
16 for u ∈ Curr do
17     for v : (u, v) ∈ E do
18         if v ∉ Vst then
19             Next ← Next ∪ {v}
20             Vst ← Vst ∪ {v}
21             Prt(v) ← u
22         end
23     end
24 end
25 Function BottomUp()
26 for v ∈ V and v ∉ Vst do
27     for u : (u, v) ∈ E do
28         if u ∈ Curr then
29             Next ← Next ∪ {v}
30             Vst ← Vst ∪ {v}
31             Prt(v) ← u
32             break
33         end
34     end
35 end
```

When we estimate the performance of the top-down method, we first need to measure the data to be processed at each level. From line 17 of Algorithm 2, the top-down method must traverse the edges of all the vertices in the frontier. Hence, we define the number of edges as $e_{frontier}$.

Table 1 Performance factors

Factor	Description
v_{all}	The total number of vertices
v_{left}	The number of vertices left to process
$v_{frontier}$	The number of vertices in the frontier
e_i	The number of edges in vertex i

$$e_{frontier} = \sum_{i \in V_{frontier}} e_i \tag{1}$$

We denote the speed with which edges can be traversed in the top-down method as $speed(TopDown)$. Thus, for a given level, the processing time of the top-down method is $t_{TopDown}$.

$$t_{TopDown} = \frac{e_{frontier}}{speed(TopDown)} \tag{2}$$

To estimate the performance of the bottom-up method, we need to calculate the number of unvisited vertices and left edges because, in the worst case, all the left edges must be visited. We refer to the number of left edges as e_{left}.

$$e_{left} = \sum_{i \in V_{left}} e_i \tag{3}$$

For a given level of computation in the bottom-up method, not all vertex edges in the frontier require processing. As shown in line 28 of Algorithm 2, the algorithm stops traversing the edges of a vertex when it finds a neighbor in the frontier. In the optimal case, the number of visited edges equals the frontier size (i.e., one edge is processed for each vertex). Therefore, we add the parameter α ($0 < \alpha < 1$) to estimate the average bottom-up traversal time. We denote the speed with which edges can be traversed in the bottom-up method as $speed(BottomUp)$. The processing time of the bottom-up method is $t_{BottomUp}$.

$$t_{BottomUp} = \frac{\alpha \times e_{left}}{speed(BottomUp)} \tag{4}$$

During processing, we compare $t_{TopDown}$ and $t_{BottomUp}$ to choose between the top-down and bottom-up methods. In Eqs. 2 and 4, these speeds are related to the graph distribution. To achieve the best performance, we implemented five versions that cover different situations, as shown in Table 2—CPU and GPU versions for both the top-down and bottom-up methods. In addition, for the bottom-up GPU implementation, we provide two versions: BG, which does not use local memory, and BFC, which uses local memory to cache frequent data.

To estimate $speed(BottomUp)$ and $speed(TopDown)$ for different implementations, we use a training set containing a variety of patterns to measure different implementations. To form the training set, we used the generator from Graph500 [34] to generate several graphs. The generator provides five parameters: S, A, B, C, and D, where S controls the generated graph size. The generated graphs should have 2^S

Table 2 BFS implementations on integrated architectures

Strategy	Description
TC	Top-down direction + CPU
TG	Top-down direction + GPU
BC	Bottom-up direction + CPU
BG	Bottom-up direction + GPU
BGC	Bottom-up direction + GPU with local memory

vertices and 2^{S+4} edges. The parameters A, B, C, and D control the edge distribution among the graph's vertices. The graph is recursively generated with four equal-sized submatrices. A, B, C, and D represent the unequal probabilities of an edge falling in the four partitions of a matrix. The sum of A, B, C, and D is equal to one. We refer readers to [9] for a more detailed description of graph generation. We set S to {16, 17, 18, 19}; then, for each S value, we randomly generated 20 graphs by setting A, B, C, and D to different values. To measure the speed of the top-down method for CPUs and GPUs, we calculated the frontier size, $e_{frontier}$, and time, t, at each level, and used the result of $e_{frontier}$ divided by t as the speed. Finally, we used the average speed from all levels as the estimated performance. We used similar methods to estimate the speed for the bottom-up method.

4 Optimization

In this section, we provide three optimizations to improve the performance of our BFS on integrated architectures.

4.1 Data structure optimization

To implement the BFS in Algorithm 2, the first consideration involves which data structures to use. We consider optimizing three data structures in our BFS implementation: (1) the input graph, $G = (V, E)$, which is read only; (2) the assistant data structures, *Curr*, *Next*, and *Vst*; and (3) the result, *Prt*, which shows the parent of each vertex.

Graph data are similar to a sparse matrix; thus, we consider using the format of a compressed sparse row (CSR) [38] to store the input graph. Our graph data consist of two arrays. The first array, e, represents the edges and is an integer array, containing the destination vertices of the edges. The length of array e is the total number of edges in the graph. The second array, v, represents vertices and is also an integer array, containing pointers to the starting locations of the edges for each vertex in array e. The length of array v is the total number of vertices plus one. Using array v, we can infer the degree of edges for each vertex. Because vertices with large degrees are more likely to be edge destinations, we sort the destinations in the edge array, e, for each vertex. For the bottom-up BFS, this storage format has a large probability of finding

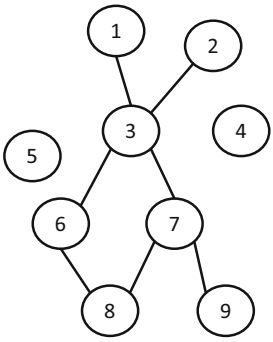
the neighbor in the frontier and thus achieving early termination during edge traversal. This storage format also performs better because it requires less memory accesses for the array e. We provide an example in Sect. 4.2.

After considering input graph storage, we design the assistant data structures, including the current frontier *Curr*, the next frontier *Next*, and the set of visited vertices *Vst*. These three assistant data structures are subsets of the vertices of the input graph. Before we choose a concrete implementation for them, we analyze the most common operations for each data structure. For the current frontier, *Curr*, our BFS implementation iterates every vertex in *Curr*; consequently, the performance depends primarily on the size of *Curr*. When *Curr* is small, a queue data structure is efficient; when it is large, a bitmap is a better choice. As stated in Sect. 3.1, the top-down method is the most efficient in situations where the frontier size is small. In such situations, the queue data structure is a good choice. For the bottom-up method, which is the most efficient when the frontier size is large, a bitmap is a better choice. The data structure of the next frontier, *Next*, involves frequent insert operations and its performance considerations are similar to those of *Curr*. For top-down traversal, we choose a queue data structure. For bottom-up traversal, a new issue for the bitmap structure is that multiple threads may change the bits in the same byte-offset range, which incurs atomic operations. To solve this problem, we use a char-map data structure instead of a bitmap structure for bottom-up traversal. For the set of visited vertices *Vst*, the BFS algorithm most frequently checks whether a vertex is visited by *Vst*. Therefore, a bitmap data structure is suitable for *Vst*.

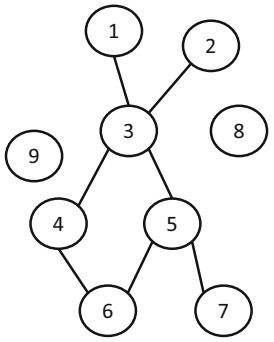
The last data structure is the result, *Prt*, which involves frequent write operations for which direct memory access is the best choice. Therefore, we use an integer array to store the parent for each vertex. Note that more than two threads may write to the same location of Prt; however, this does not influence the correctness of our BFS implementation because all the written values are equally correct.

4.2 Graph layout remapping

We analyze the bottom-up process in Algorithm 2 and find that the vertex traversal in line 26 accesses some unnecessary vertices; therefore, it can be optimized. In graphs, not all vertices have edges, i.e., some vertices have a degree of zero. Nevertheless, these vertices still belong to the unvisited vertex set and need to be counted in the bottom-up sequence (line 26 in Algorithm 2). When the number of zero-degree vertices is large, the performance cost can be large because these vertices need to be traversed once at every level. To reduce this cost, we remap the vertices with a degree greater than one to a new consecutive range, leaving the zero-degree vertices in another range because we do not need to traverse the zero-degree vertices. Figure 2 shows an example of vertex remapping in which Fig. 2a shows the original input graph and Fig. 2b shows the index remapping result, which gives each vertex a new number. During the remapping process, we reduce the size of the vertices that must be traversed from 9 to 7. The remapped graph is shown in Fig. 2c. The new vertices (8 and 9) have zero degrees; thus, we do not schedule these vertices in the range of vertices that have more than one edge. Because the size is reduced to 7, line 26 in Algorithm 2 can avoid re-accessing



Original ID	Remapped ID
1	1
2	2
3	3
4	8
5	9
6	4
7	5
8	6
9	7



(a) **(b)** **(c)**

Fig. 2 An example of index remapping for the input graph. **a** Original graph, **b** index mapping, **c** mapped graph

the zero-degree vertices. After the BFS traversal, we replace the ID for each vertex based on the index mapping in Fig. 2b. The BFS on NUMA architecture [49] uses similar techniques.

As described in Sect. 4.1, we reschedule the order of edge storage for each vertex to reduce the number of edges to traverse in the bottom-up traversal. For the data structure of each vertex, we sort the storage order of adjacent vertices by their degrees. For example, in Fig. 2a, $vertex3$ has four adjacent vertices ($vertex1$, $vertex2$, $vertex6$, and $vertex7$). The degree of $vertex1$ and $vertex2$ is 1, that of $vertex6$ is 2, and that of $vertex7$ is 3. Therefore, $vertex3$ should store its neighbors in the following sequence ($vertex7$,3), ($vertex6$,2), ($vertex1$,1), and ($vertex2$,1). This storage format increases the probability of finding the parent vertices and can reduce the number of edges that must be traversed. Moreover, we consider storing vertices with large degrees in local memory because the frontier will be checked many times during bottom-up BFS traversal (line 28 in Algorithm 2).

4.3 Enabling large dataset processing

An integrated architecture provides the opportunity to process large datasets, but due to OpenCL memory-object size limitations, we cannot directly allocate sufficient space to store all the edges in a single buffer when the input graph is too large. Therefore, we separate the edges into several parts, each of which is smaller than the memory size limitation. We also provide a decision module to verify whether the edge size exceeds the memory limitation. If so, we store these edges in separate parts with additional assistant data structures as shown in Fig. 3, including (1) the ID for each edge part, (2) the number of edges in each part, and (3) the starting offset of the edges for each part. With the information, each part contains only a subset of edges of the original graph, and our BFS implementation can select the correct part to process during traversal.

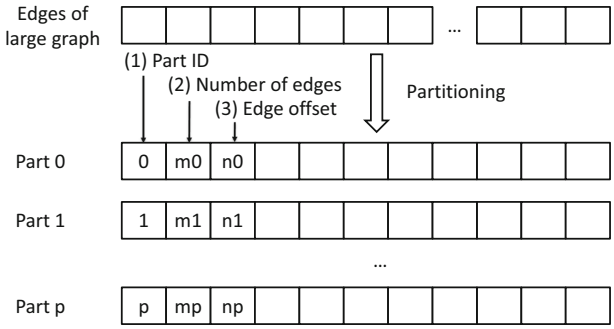


Fig. 3 Partitioning edges for large graphs

5 Evaluation

We evaluate our BFS algorithm on integrated architectures with the Graph500 benchmark. We start by describing our platforms and input graphs.

5.1 Experiment setup

Platform We use the AMD A-Series APU A10-7850K [5] and Ryzen 5 2400G [3], shown in Table 3, as the experimental platform to measure the performance of our BFS algorithm. The A10-7850K CPU has four cores with four threads, and its GPU has eight compute units. We use GCC (version 4.8.2) with the $O3$ optimization level for compilation and the OpenCL library version 2.0. The Ryzen 5 2400G CPU has four cores with eight threads, and its GPU has eleven compute units. Currently, this is the latest integrated architecture. AMD provides only Windows drivers for the Ryzen 5 2400G; therefore, we conduct our experiments for this platform using Microsoft Visual Studio.

Input Graphs Graph500 [34] is a well-known large-scale benchmark based on the BFS for a large undirected graph. We use the Graph500 generator to verify the efficiency of our BFS implementation. The generated graphs are based on a Kronecker

Table 3 The experimental platform configuration

Platform	AMD A10-7850K	AMD Ryzen 5 2400G
Operating system	Ubuntu 14.04.1	Windows 10
Number of CPU cores	4	4
CPU frequency (GHz)	3.7	3.6
Number of GPU compute units	8	11
GPU max frequency (GHz)	0.72	1.25
Memory size (GB)	32	32
Memory frequency (MHz)	1600	2400

Table 4 Input graphs used in our experiments

Scale	22	23	24	25	26
Vertex	4.19M	8.39M	16.78M	33.55M	67.11M
Edge	67.11M	134.22M	268.44M	536.87M	1073.74M

graph model [19] with an average degree of 16. We evaluate the BFS using the five graphs listed in Table 4. On average, each vertex has 16 edges.

Comparison We use three BFS versions in this paper.

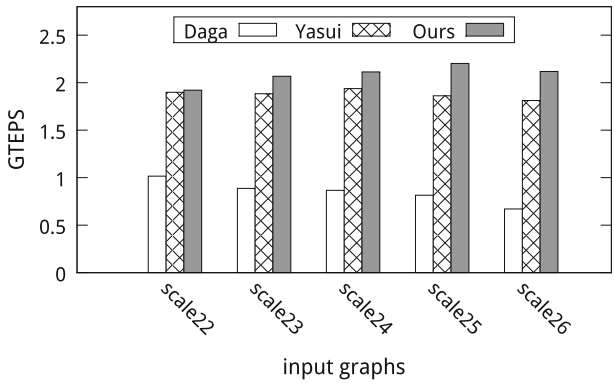
1. The Daga version [12], which is the fastest BFS version targeting CPU–GPU integrated architectures of which we are aware. We add the extension described in Sect. 4.3 to enable it to process large graphs.
2. The Yasui version [49], which is ported from a BFS implementation for NUMA architectures. We do not use its NUMA-related optimizations; however, we use this version for comparison because it achieves high performance on our platforms.
3. Our hybrid version, which uses both the CPU and GPU to process graphs with all optimizations.

The first and second algorithms are proposed in previous works, and the third is our implementation. In our implementation, for the top-down method, we parallelize the computation process by distributing different rows to different threads. For the bottom-up method, we parallelize it by making different threads process different edges. We choose OpenMP for the CPU device and OpenCL for the GPU device.

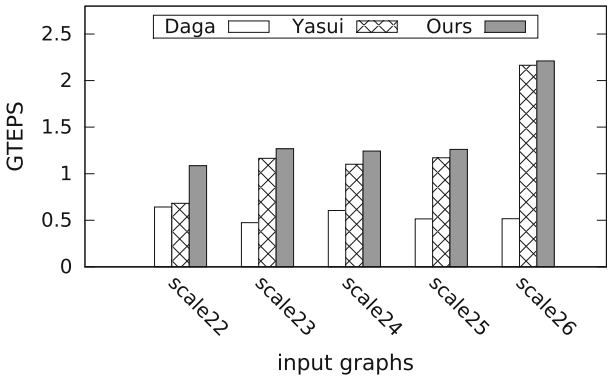
5.2 Performance

For comparison purposes, Fig. 4 shows the performance results of different BFS implementations on the five datasets. Figure 4a represents the results with the A10-7850K, and Fig. 4b represents the results with the Ryzen 5 2400G. We follow the rules of Graph500 and use giga-traversed edges per second (GTEPS) as the performance metric. In the evaluation, let m be the number of edges in a traversed component of the graph and t be the BFS execution time. The normalized performance rate (GTEPS) is defined as m/t. In general, our BFS achieves the best performance on all the input graphs for the integrated architectures—considerably better than that achieved by the other versions. Our BFS achieves averages of 2.1 GTEPS on the A10-7850K and 1.4 GTEPS on the Ryzen 5 2400G. The Daga version achieves averages of 0.9 GTEPS on the A10-7850K and 0.6 GTEPS on the Ryzen 5 2400G. The Yasui version's performance is closer to ours; it achieves averages of 1.9 GTEPS on the A10-7850K and 1.3 GTEPS on the Ryzen 5 2400G. Overall, our proposed BFS implementation is 2.3 × faster than the Daga version and 1.1 × faster than the Yasui version.

From Fig. 4, we can see that the Yasui version performs much faster than the Daga version, which indicates that the graph layout remapping is an effective optimization technique in BFS. Our BFS has higher performance than the Yasui version does, because we use the GPU to accelerate some levels of computation; however, the



(a)



(b)

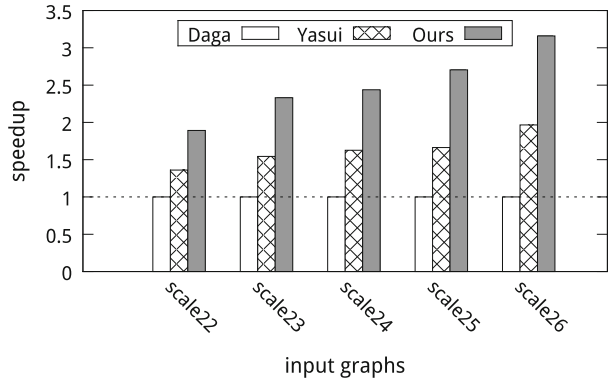
Fig. 4 Performance results of different BFS implementations. **a** A10-7850K, **b** Ryzen 5 2400G

speedup of ours over the Yasui version is not large, because BFS is a data-access-intensive application in which both the CPU and the GPU access the same memory. Figure 4 also shows that our BFS implementation is efficient, especially when the data size is large. The performance differs between the A10-7850K and the Ryzen 5 2400G, which may be related to the driver and operating system differences. The Daga version [12] is the work most similar to ours. Compared to ours, the advantage of the Daga version is that it can run directly on an integrated architecture without first having to build a performance model. However, its parameters of strategy switching are fixed. Moreover, our version has extra optimizations. The Yasui implementation [49] is efficient, but it does not take advantage of GPUs.

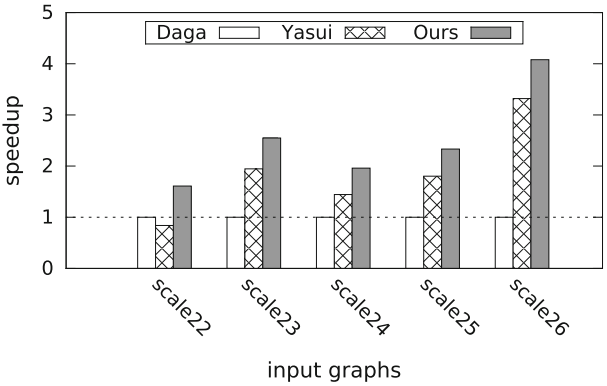
5.3 Energy efficiency

Energy efficiency is an important indicator for integrated architectures; therefore, we also evaluate our method from an energy perspective. We use an external power

meter (WT210) [35] to measure the real-time system power to calculate the energy efficiency. The WT210 monitors the input power and records the real-time power at intervals of one second. A similar method is used in [52]. Because GTEPS reflects the computing performance, we use GTEPS divided by power to represent energy efficiency. We use the Daga version as the baseline and define "speedup" as the baseline's energy efficiency (performance per watt) divided by the corresponding method's energy efficiency. Figure 5 shows the energy efficiency results for the different BFS implementations. Figure 5a shows the results on the A10-7850K, and Fig. 5b shows the results on the Ryzen 5 2400G. Our BFS is highly energy efficient and consistently achieves improvements in energy efficiency compared to the other BFS implementations, especially on large graphs. The average energy efficiency of our BFS shows a speedup of approximately 2.3 × over the Daga version on the A10-7850K and a speedup of approximately 2.5 × on the Ryzen 5 2400G. Moreover, our BFS achieves



(a)



(b)

Fig. 5 Energy efficiency results of different BFS implementations. The baseline is the Daga version. **a** A10-7850K, **b** Ryzen 5 2400G

approximately a 1.6 × speedup over the Yasui version on the A10-7850K and 1.3 × on the Ryzen 5 2400G.

Our BFS implementation shows larger speedup from an energy efficiency perspective than from a performance perspective. Energy efficiency is an important aspect of parallel computing. AMD argues that integrated architectures form an important balance between performance and power [5]. From Fig. 5, we can see that using only the CPU does not achieve optimal energy efficiency. Energy efficiency stems from two factors: runtime power and execution time. The GPU has lower power to process workloads than the CPU does on integrated architectures. Because our method uses GPUs in some iterations and executes in less time, the energy efficiency of our implementation is much better than that of the previous methods.

5.4 Analysis

We analyze the different implementations from Table 2 of our hybrid BFS in this section. In Table 5, we use the $scale26$ as an instance to show the execution times of TC, TG, BC, BG, and BGC at each level. There are eight levels during the BFS traversal, and the fifth level has the largest frontier size. As Table 5 shows, no strategy is the most suitable for all levels. Different traversal orders with different devices result in diverse performances. When the frontier size is small, the top-down method performs better than the bottom-up method does; however, when the frontier size is large, the bottom-up method performs better. In our implementation, we provide a parameter to determine the interval at which the algorithm decides whether it should switch the current strategy. By default, this parameter value is set to one, which means that the strategy selection module is called after each iteration. Although Table 5 indicates that TC, BG, and BGC are necessary on the current platform, we do not guarantee that these three strategies are the best on other platforms. Hence, we still need to consider all strategies in other situations. Moreover, Table 5 indicates that frontier size plays an important role in strategy switching, and our model considers it an important factor. Based on the frontier size and the predicted performance for each method, our model can estimate the execution time for each version, allowing it to choose a suitable strategy.

Table 5 Details for each level of different BFS strategies on the $scale26$ example

Level	Frontier size	TC (s)	TG (s)	BC (s)	BG (s)	BGC (s)
1	1	0.02	0.03	4.37	9.42	9.44
2	1	0.02	0.05	4.34	8.53	8.47
3	5198	1.81	3.44	3.16	0.29	0.27
4	55,391,290	9.76	17.35	2.66	0.08	0.07
5	126,205,388	0.30	0.97	2.84	0.06	0.06
6	414,288	0.03	0.08	2.82	0.05	0.05
7	1132	0.02	0.02	2.83	0.05	0.05
8	4	0.02	0.02	0.03	0.05	0.05

Table 6 Number of vertices before and after graph layout remapping

Scale	22	23	24	25	26
Original	4.19M	8.39M	16.78M	33.55M	67.11M
Optimized	2.40M	4.61M	8.87M	17.06M	32.80M
Saved (%)	42.87	45.03	47.12	49.15	51.12

Section 5.2 indicates that graph layout remapping is an effective optimization technique; therefore, we analyze the graph remapping technique in this section. As explained in Sect. 4.2, this technique removes the zero-degree vertices. We show the number of vertices before and after this optimization in Table 6. As Table 6 shows, approximately 48% of the vertices are saved by graph layout remapping, which avoids a large number of unnecessary memory accesses in our BFS. The graph layout remapping process occurs during the preprocessing stage. On average, the graph layout remapping process takes approximately 41.5% of the preprocessing time. The other optimizations concern data structure selection and techniques to process large datasets. These optimizations have been integrated into all versions; related explanations appear in Sect. 4.

5.5 Discussion

Limitation Our BFS algorithm needs to run a benchmark first to build the adaptive model for a given architecture. When switching to another platform, the effectiveness of the model and the accuracy of the speed estimation depend on the architectural differences between the original platform and the new platform. One solution is to perform cross-platform prediction, which we may work on in the future. Another simple but effective solution would be rerunning the benchmark on the new platform, but this does not apply to scenarios where the platforms change frequently. Regarding the optimizations, our graph layout remapping technique does not contribute that much when a graph is already sorted.

Insights Although the proposed adaptive method in this paper is applied to BFS, the idea of using the most suitable device for different iterations could also apply to other situations. For example, iterations with high parallelism should use GPUs; otherwise, they should use CPUs. As stated in Sect. 2.2, other graph algorithms also have similar characteristics as BFS. For example, in parallel implementations of graph coloring and connected component algorithms, the frontier sizes of active vertices can be large at the beginning and small at the end. Accordingly, they should use GPUs first, and use CPUs when the frontier size is small. The equations would need to be adjusted slightly because the evaluation indicator for the processing speed of these two applications involves vertices rather than edges.

6 Related work

In recent years, graph computing has been a hot research topic, and BFS, as a representative algorithm, has attracted attention from many researchers [12,42,48,49,54].

Among these studies, the BFS implementation from Daga et al. [12] is the closest to ours. They are the first to implement a hybrid BFS algorithm on integrated architectures. Our BFS and their work differ in both applicability and main methodology. First, limited by the maximum size of the OpenCL object, their work cannot handle large graphs, while we include a special design for partitioning and processing large graphs. Second, their work uses a simple strategy to select a top-down or bottom-up algorithm, while ours uses a runtime model that provides a more complete analysis. Moreover, our algorithm includes additional optimizations and achieves better performance.

The most challenging problem in BFS on heterogeneous architectures is the load imbalance problem. On GPUs, one simple approach is to map vertices in the frontier to discrete threads [32], but that approach suffers from serious load imbalance problems. Hong et al. [15] proposed a method to balance the process by assigning a warp with a set of vertices. Merrill et al. [33] carefully designed and optimized the neighbor-gathering and status-lookup stage within a cooperative thread array (CTA). Other accomplishments, such as those in [24,46], use threads, warps, CTAs or the full processor to address vertices of different degrees. Moreover, Scarpazza et al. [40] implemented BFS in Cell/BE by splitting the data into chunks that can fit into local storage with synergistic processing elements. Liu et al. [26] implemented iBFS, which can perform multiple BFS traversals on the same graph. In contrast to these BFS implementations, our adaptive BFS algorithm has a special design intended to target integrated architectures.

Some researchers have attempted to maximize the performance of the BFS algorithm on a single machine. Thus, many BFS optimization techniques have been developed, such as using bitmaps or a hierarchical structure for frontiers to restrict random access in a fast cache [1,10,27,28,53], lock-free array updates [10], and serializing random access and refining data structures in NUMA architectures [1,49]. However, as the size of the input graph increases, the memory of a single GPU becomes insufficient. Several studies have investigated how to process such large graphs, including using multi-GPUs [33,46], collaborative CPU–GPU designs [16], I/O optimizations [22,25], and using external storage [21,37] to extend the ability of their BFS implementations.

BFS is a typical irregular application. Graphs can be represented in sparse matrix format. Several new storage formats have been proposed to optimize applications based on sparse matrices [2,23,30,41,44,47,51]. For example, Liu and Vinter [30] proposed the CSR5 format for sparsity-insensitive sparse applications. Sedaghati et al. [41] summarized the features of sparse matrices and built a model to select the most suitable format automatically. Ashari et al. [2] used sparse matrices to calculate graph applications and performed row reordering in sparse matrices for GPU threads. Integrated architectures have also been used to accelerate some other parallel algorithms, such as sparse matrix–vector multiplication [31], sparse matrix–matrix multiplication [29], and sparse triangular solve [45].

7 Conclusion

In this study, we have developed an adaptive hybrid BFS algorithm for integrated CPU–GPU architectures. Our BFS integrates the top-down and bottom-up BFS algorithms and utilizes both the CPU and GPU devices. Along with optimizations, we provide a performance model that can select a suitable algorithm and device for each level during BFS traversal. Our BFS implementation can process graphs with more than 67 million vertices and one billion edges, and it executes at approximately 2.1 GTEPS on a single integrated architecture. We also demonstrate our BFS algorithm's energy efficiency, which is approximately 2.3 × better than the state-of-the-art BFS on integrated architectures.

Acknowledgements The authors sincerely thank the anonymous reviewers for their valuable comments and suggestions. This work is partially supported by the National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant Nos. 61732014, 61722208, 61472201, and 61472427). This work is also supported by Huawei Technologies Co. Ltd, Beijing Natural Science Foundation (No. 4172031), China Postdoctoral Science Foundation (2017M620992), and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China (Nos. 16XNLQ02, 18XNLG07). Jidong Zhai is the corresponding author of this paper.

References

1. Agarwal V, Petrini F, Pasetto D, Bader DA (2010) Scalable graph exploration on multicore processors. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, pp 1–11
2. Ashari A, Sedaghati N, Eisenlohr J, Parthasarath S, Sadayappan P (2014) Fast sparse matrix–vector multiplication on GPUs for graph applications. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC14. IEEE, pp 781–792
3. AMD (2018) AMD Ryzen 5 2400G with Radeon RX Vega 11 Graphics. https://www.amd.com/en/products/apu/amd-ryzen-5-2400g
4. Beamer S, Asanović K, Patterson D (2013) Direction-optimizing breadth-first search. Sci Program 21(3–4):137–148
5. Bouvier D, Sander B (2014) Applying AMDs Kaveri APU for heterogeneous computing. In: Hot Chips: A Symposium on High Performance Chips (HC26)
6. Brandes U (2001) A faster algorithm for betweenness centrality. J Math Sociol 25(2):163–177
7. Branover A, Foley D, Steinman M (2012) AMD fusion APU: Llano. IEEE Micro 32(2):28–37
8. Broder A, Kumar R, Maghoul F, Raghavan P, Rajagopalan S, Stata R, Tomkins A, Wiener J (2000) Graph structure in the web. Comput Netw 33(1):309–320
9. Chakrabarti D, Zhan Y, Faloutsos C (2004) R-MAT: a recursive model for graph mining. In: SDM, vol 4. SIAM, pp 442–446
10. Chhugani J, Satish N, Kim C, Sewall J, Dubey P (2012) Fast and efficient graph traversal algorithm for CPUs: maximizing single-node efficiency. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp 378–389
11. Cormen TH (2009) Introduction to algorithms. MIT Press, Cambridge
12. Daga M, Nutter M, Meswani M (2014) Efficient breadth-first search on a heterogeneous processor. In: 2014 IEEE International Conference on Big Data (Big Data). IEEE, pp 373–382
13. Dongarra JJ, Meuer HW, Strohmaier E et al (1997) Top500 supercomputer sites. Supercomputer 13:89–111
14. Erdös Rényi (1959) On random graphs I. Publ Math Debr 6:290–297
15. Hong S, Kim SK, Oguntebi T, Olukotun K (2011) Accelerating CUDA graph algorithms at maximum warp. In: ACM SIGPLAN Notices, vol 46. ACM, pp 267–276

16. Hong S, Oguntebi T, Olukotun K (2011) Efficient parallel graph exploration on multi-core CPU and GPU. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, pp 78–88
17. Intel Corporation (2014) The compute architecture of Intel processor graphics Gen7.5. https://software.intel.com/
18. Jensen TR, Toft B (2011) Graph coloring problems, vol 39. Wiley, London
19. Kepner J, Gilbert J (2011) Graph algorithms in the language of linear algebra. SIAM, Philadelphia
20. Korf RE (1985) Depth-first iterative-deepening: an optimal admissible tree search. Artif Intell 27(1):97–109
21. Korf RE, Schultze P (2005) Large-scale parallel breadth-first search. In: Association for the Advancement of Artificial Intelligence (AAAI), vol 5, pp 1380–1385
22. Kumar P, Huang HH (2016) G-store: high-performance graph store for trillion-edge processing. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC16. IEEE, pp 830–841
23. Li J, Tan G, Chen M, Sun N (2013) SMAT: an input adaptive auto-tuner for sparse matrix–vector multiplication. In: ACM SIGPLAN Notices, vol 48. ACM, pp 117–126
24. Liu H, Huang HH (2015) Enterprise: breadth-first graph traversal on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, p 68
25. Liu H, Huang HH (2017) Graphene: fine-grained IO management for graph computing. In: USENIX Conference on File and Storage Technologies (FAST), pp 285–300
26. Liu H, Huang HH, Hu Y (2016) iBFS: concurrent breadth-first search on GPUs. In: Proceedings of the 2016 International Conference on Management of Data. ACM, pp 403–416
27. Liu T, Chen CC, Kim W, Milor L (2015) Comprehensive reliability and aging analysis on SRAMs within microprocessor systems. Microelectron Reliab 55(9):1290–1296
28. Liu T, Chen CC, Wu J, Milor L (2016) Sram stability analysis for different cache configurations due to bias temperature instability and hot carrier injection. In: 2016 IEEE 34th International Conference on Computer Design (ICCD). IEEE, pp 225–232
29. Liu W, Vinter B (2015) A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. J Parallel Distrib Comput 85:47–61
30. Liu W, Vinter B (2015) CSR5: an efficient storage format for cross-platform sparse matrix–vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing. ACM, pp 339–350
31. Liu W, Vinter B (2015) Speculative segmented sum for sparse matrix–vector multiplication on heterogeneous processors. Parallel Comput 49:179–193
32. Luo L, Wong M, Hwu W (2010) An effective GPU implementation of breadth-first search. In: Proceedings of the 47th Design Automation Conference. ACM, pp 52–55
33. Merrill D, Garland M, Grimshaw A (2012) Scalable GPU graph traversal. In: ACM SIGPLAN Notices, vol 47. ACM, pp 117–128
34. Murphy RC, Wheeler KB, Barrett BW, Ang JA (2010) Introducing the Graph 500. In: Cray Users Group (CUG) Proceedings
35. YOKOGAWA (2017) WT210/WT230 digital power meters. http://tmi.yokogawa.com/products/digital-power-analyzers/
36. Nikolskiy VP, Stegailov VV, Vecher VS (2016) Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. In: 2016 International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp 682–689
37. Pearce R, Gokhale M, Amato NM (2013) Scaling techniques for massive scale-free graphs in distributed (external) memory. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS). IEEE, pp 825–836
38. Saad Y (1990) SPARSKIT: a basic tool kit for sparse matrix computations. NASA technical report, NASA, pp 1–30
39. Satish N, Sundaram N, Patwary MMA, Seo J, Park J, Hassaan MA, Sengupta S, Yin Z, Dubey P (2014) Navigating the maze of graph analytics frameworks using massive graph datasets. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. ACM, pp 979–990
40. Scarpazza DP, Villa O, Petrini F (2008) Efficient breadth-first search on the Cell/BE processor. IEEE Trans Parallel Distrib Syst 19(10):1381–1395

41. Sedaghati N, Mu T, Pouchet LN, Parthasarathy S, Sadayappan P (2015) Automatic selection of sparse matrix representation on GPUs. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, pp 99–108
42. Shi X, Zheng Z, Zhou Y, Jin H, He L, Liu B, Hua QS (2018) Graph processing on GPUs: a survey. ACM Comput Surv 50(6):81
43. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng 12(3):66–73
44. Su BY, Keutzer K (2012) clSpMV: a cross-platform OpenCL SpMV framework on GPUs. In: Proceedings of the 26th ACM International Conference on Supercomputing. ACM, pp 353–364
45. Wang X, Liu W, Xue W, Wu L (2018) swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, pp 338–353
46. Wang Y, Davidson A, Pan Y, Wu Y, Riffel A, Owens JD (2016) 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. ACM, p 11
47. Yan S, Li C, Zhang Y, Zhou H (2014) yaSpMV: yet another SpMV framework on GPUs. In: ACM SIGPLAN Notices, vol 49. ACM, pp 107–118
48. Yang C, Buluc A, Owens JD (2018) Implementing push–pull efficiently in GraphBLAS. In: International Conference on Parallel Processing (ICPP)
49. Yasui Y, Fujisawa K (2015) Fast and scalable NUMA-based thread parallel breadth-first search. In: 2015 International Conference on High Performance Computing and Simulation (HPCS). IEEE, pp 377–385
50. Zhang F, Zhai J, Chen W, He B, Zhang S (2015) To co-run, or not to co-run: a performance study on integrated architectures. In: 2015 IEEE 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, pp 89–92
51. Zhang F, Wu B, Zhai J, He B, Chen W (2017) FinePar: irregularity-aware fine-grained workload partitioning on integrated architectures. In: International Symposium on Code Generation and Optimization (CGO). IEEE Press, pp 27–38
52. Zhang F, Zhai J, He B, Zhang S, Chen W (2017) Understanding co-running behaviors on integrated CPU/GPU architectures. IEEE Trans Parallel Distrib Syst 28(3):905–918
53. Zhang R, Liu T, Yang K, Milor L (2017) Analysis of time-dependent dielectric breakdown induced aging of SRAM cache with different configurations. Microelectron Reliab 76:87–91
54. Zhong J, He B (2014) Medusa: simplified graph processing on GPUs. IEEE Trans Parallel Distrib Syst 25(6):1543–1552