Noname manuscript No.
(will be inserted by the editor)

Performance Evaluation and Analysis of Sparse Matrix and Graph Kernels on Heterogeneous Processors

Feng Zhang[1] · Weifeng Liu[2] · Ningxuan Feng[1] · Jidong Zhai[3] · Xiaoyong Du[1]

Received: date / Accepted: date

Abstract Heterogeneous processors integrate very distinct compute resources such as CPUs and GPUs into the same chip, thus can exploit the advantages and avoid disadvantages of those compute units. We in this work evaluate and analyze eight sparse matrix and graph kernels on an AMD CPU-GPU heterogeneous processor by using 956 sparse matrices. Five characteristics, i.e., *load balancing*, *indirect addressing*, *memory reallocation*, *atomic operations*, and *dynamic characteristics* are our major considerations. The experimental results show that although the CPU and GPU parts access the same DRAM, very different performance behaviors are observed. For example, though the GPU part in general outperforms the CPU part, it cannot achieve the best performance in all cases given by the CPU part. Moreover, the bandwidth utilization of atomic operations on heterogeneous processors can be much higher than a high-end discrete GPU.

Keywords Heterogeneous processor · Performance analysis · Sparse matrix computation.

1 Introduction

About a decade ago, graphics processing unit (GPU) has been introduced to high performance computing. Because of its high peak compute performance and bandwidth, a large amount of compute kernels and real-world applications have been accelerated on GPUs [36]. However, it also has been reported that

[1] Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing, China.
[2] Department of Computer Science and Technology, China University of Petroleum, Beijing, China.
[3] Department of Computer Science and Technology, Tsinghua University, Beijing, China.
Weifeng Liu is the corresponding author.
E-mail: weifeng.liu@cup.edu.cn

not all compute patterns are suitable for GPU computing due to their irregularity [20] and time consuming memory copy between host memory and GPU memory [16]. As a result, heterogeneous processor, also called accelerated processing unit (APU) or CPU-GPU integrated architecture, has been expected to exploit advantages of both CPUs and GPUs and avoid memory copy between memory areas of different devices. Schulte et al. [41] and Vijayaraghavan et al. [47] recently reported that with a good design, heterogeneous processors can be a competitive building block for exascale computing systems.

The effective design of such heterogeneous processors is challenging. For example, because CPU and GPU applications normally have very different memory access patterns, implementing efficient cache coherence between the two parts is an open problem. Several hardware and software supporting techniques have been developed [1,9,38]. Also, when both parts share the last level cache, data prefetching scheme can be improved through adding new instructions [51]. In addition, low-power and performance/watt ratio optimization are crucial design targets as well [4,60].

Despite the difficulties on architecture design, a few usable high performance heterogeneous processors, such as AMD Carrizo [18], Intel Skylake [12] and NVIDIA Denver [3], have been released in recent years. Such integrated architectures inspired a number of novel techniques for various parallel problems. Daga et al. on AMD heterogeneous processors evaluated several kernels and applications [6], and optimized B+ tree search [7] and breadth-first search (BFS) [8]. Zhang et al. [54] developed faster BFS through traversal order optimization. Puthoor et al. [39] developed new DAG scheduling methods on heterogeneous processors. Liu and Vinter designed a new heap data structure called ad-heap [29] and new sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM) algorithms [32,30] for heterogeneous processors. Said et al. [40] demonstrated that seismic imaging can be faster and more energy efficient on heterogeneous processors. Zhu et al. [58, 60] and Zhang et al. [56,57] studied co-run behaviors of various kernels, and Zhang et al. [55] developed effective workload partitioning approaches for heterogeneous processors.

However, irregular algorithms in particular sparse matrix and graph computations have not been systematically studied in existing work. Zhang et al. [57,55] took sparse matrix-vector multiplication (SpMV) and several graph kernels into consideration in their co-run benchmarks and scheduling algorithm design. But only very limited number of sparse matrices anf graphs were used for benchmarking. Also, other important sparse matrix kernels, e.g., SpGEMM [5,30,23,24], sparse matrix transposition (SpTRANS) [49] and sparse triangular solve (SpTRSV) [28,50], have not been well studied on heterogeneous processors.

We in this paper evaluate and analyze performance behaviors of eight representative sparse matrix kernels on the latest AMD APU Ryzen 5 2400G including CPU cores codenamed Zen and GPU cores codenamed Vega. Among the eight kernels, four kernels are from scientific computation, i.e., sparse matrix-vector multiplication (SpMV), sparse matrix-matrix multiplication (SpGEMM), sparse matrix transposition (SpTRANS), and sparse triangular solve (SpTRSV), and the other four kernels from graph computing, i.e., PageRank (PR), graph coloring (GC), connected component (CC), and breadth-first search (BFS). We use 956 large sparse matrices from the SuiteSparse Matrix Collection [10] as the benchmark suite for obtaining experimental results, which are statistically significant. We then analyze the best performance configurations, in terms of algorithm and compute resource, for matrices of various sparsity structures. We mainly consider five characteristics, *load balancing*, *indirect addressing*, *memory reallocation*, *atomic operations*, and *dynamic characteristics*. Moreover, a performance comparison with a high-end discrete GPU is also given for better understanding of sparse problems on various architectures. We finally discuss several challenges and opportunities for achieving higher performance for sparse matrix and graphs kernels on heterogeneous processors.

2 Background

2.1 Heterogeneous Processors

Compared to homogeneous chip multiprocessors such as CPUs and GPUs, heterogeneous processors are able to combine different types of cores into one chip, thus can deliver improved overall performance and power efficiency [41, 47], while sufficient heterogeneous parallelism exists. The main characteristics of heterogeneous processors include unified shared memory and fast communication among different types of cores in the same chip. The Cell Broadband Engine can be seen as an early form of heterogeneous processor. Currently, because of mature CPU and GPU architectures, programming environments, and various applications, the CPU-GPU integrated heterogeneous processor with multiple instruction set architectures is the most widely adopted choice.

Figure 1 shows a block diagrams of heterogeneous processors used as the experimental testbed in this paper. In general, a heterogeneous processor consists of four major parts: (1) a group of CPU cores with hardware-controlled caches, (2) a group of GPU cores with shared command processors, software-controlled scratchpad memory, and hardware-controlled caches, (3) shared memory management unit, and (4) shared global dynamic random-access memory (DRAM).

Currently, although the computation capacity of the coupled CPU-GPU processors is lower than that of the discrete GPUs, we can see that the heterogeneous processor is a potential trend for future processors. Hardware vendors all release their heterogeneous processors, such as AMD Carrizo [18], Intel Skylake [12] and NVIDIA Denver [3]. In addition, future heterogeneous processors can be more powerful; with good design, they even can be applied in exascale computing systems [41,47].

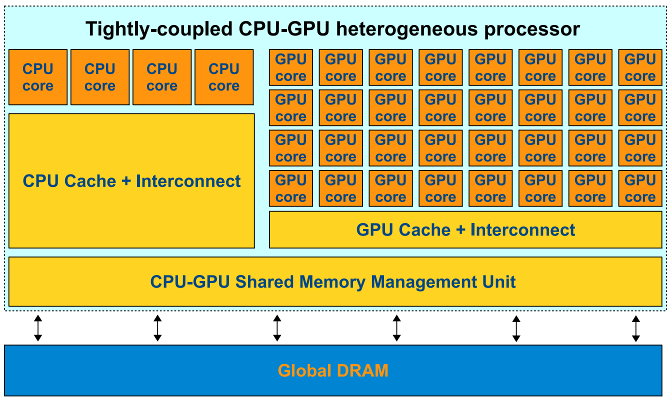
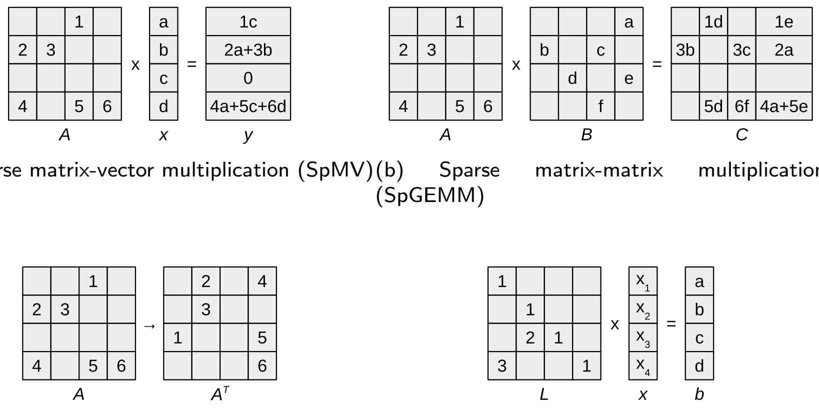

Tightly-coupled CPU-GPU heterogeneous processor
CPU core
GPU core
CPU Cache + Interconnect
GPU Cache + Interconnect
CPU-GPU Shared Memory Management Unit
Global DRAM

Fig. 1 The diagram of a tightly-coupled CPU-GPU heterogeneous processor.

2.2 Sparse Matrix and Graph Computations

Sparse matrices exist in a number of real-world applications, such as finite element methods, social network, graph computing, and machine learning. For instance, in graph computing, vertices among a graph can be represented as rows, while edges can be represented as the nonzero entries of a sparse matrix; in social network, the relation between different people can also be represented as nonzero entries in a sparse matrix. To evaluate the real impact of heterogeneous processors on sparse matrix computations, the benchmark suite of this work includes eight kernels. The first four kernels are from sparse basic linear algebra subprograms [13], and Figure 2 plots the above four operations. Calculating product of a sparse matrix and a dense vector or another sparse matrix, transposing a sparse matrix, and solving a sparse triangular system are fundamental operations for scientific computations. The last four kernels are used for graph computing. We use the implementation from the GraphBIG benchmark [35], which have different features and are used in many previous works [35,55,57].

- **Sparse matrix-vector multiplication (SpMV)** that multiplies a sparse matrix A with a dense vector x and obtains a dense vector y;
- **Sparse matrix-matrix multiplication (SpGEMM)** that multiplies a sparse matrix A with another sparse matrix B and obtains a resulting sparse matrix C;
- **Sparse matrix transposition (SpTRANS)** that transposes a sparse matrix A of row-major to A^T of row-major (or both of column-major);
- **Sparse triangular solve (SpTRSV)** that computes a dense solution vector x from a system $Lx = b$, where L is a lower triangular sparse matrix and b is a dense right-hand side vector;



(a) Sparse matrix-vector multiplication (SpMV) (b) Sparse matrix-matrix multiplication (SpGEMM)

(c) Sparse transposition (SpTRANS) (d) Sparse triangular solve (SpTRSV)

Fig. 2 Four representative sparse kernels benchmarked in this work.

- **PageRank (PR)** that ranks Internet web pages in search engines, which works by counting the links between different web pages and weighting each pages;
- **Graph coloring (GC)** that gives colors to vertices where any two adjacent vertices shall have different colors, which is a special case in graph labeling;
- **Connected component (CC)** that marks vertices in different components and calculates the number of connected components;
- **Breadth-first search (BFS)** that explores a path from a root node to each node in a graph, in a way that during each step, the unvisited neighbors of visited nodes are marked to be visited in the next step.

2.3 Characteristics of Parallel Sparse Matrix and Graph Kernels

Unlike dense matrix computations, sparse matrix and graph kernels have several unique characteristics [27].

The first one is *load balancing*. Dense matrix operations can be easily executed in parallel through row-wise, column-wise or 2D block-wise decomposition. However, the nonzero entries of a sparse matrix can randomly exist at any locations. Hence, which decomposition method gives the best load balancing depends on sparsity structure, operation pattern, and concrete hardware device.

The second is *indirect addressing*. Because of the compressed storage fashion, nonzero entries of a sparse matrix have to be accessed by indirect addresses stored in its index array. It is well known that indirect addressing brings more memory transactions and lower cache hit rate, and *cannot* be optimized at compile time since the addresses are only known at runtime.

The third is *memory reallocation*. Several sparse kernels, such as addition or multiplication of two sparse matrices, generate sparse output. The number of nonzero entries and their distribution are actually *not* known in advance. Precomputing an upper bound is one method to deal with the unknown number of nonzero entries of the output. However, this method may waste memory space. Another method is to preallocate a sparse output and reallocate more space if the initial size is small. However, such memory reallocation is expensive on GPUs currently.

The fourth is *atomic operations*. Some kernels highly depend on atomic operations to collect nonzeros or to synchronize workload of thread blocks. For example, thread blocks can use atomic operations on global variables to communicate and obtain the execution status of other thread blocks for fast synchronization. However, performance-wise, atomic operations are inherently sequential, though they can be implemented more efficiently through some architectural designs.

The fifth is *dynamic characteristics*. Some graph computing kernels have dynamic characteristics, which means that the computation process is divided into several iterations and each iteration only relates to part of a graph, i.e., part of a sparse matrix. Dynamic characteristics relate to both the input and the algorithm. When the workload related to a compute iteration is too low, the GPU fails to utilize all its compute cores.

We show a summary of the five characteristics in Table 1. The sparse kernels are used for evaluation and analysis for these characteristics in Section 3.

Table 1 Summary of the characteristics and sparse kernels.

Characteristics	Description	Sparse kernels
Load balancing	Efficient workload distribution.	SpMV
Indirect addressing	The data addresses are held in intermediate locations.	SpMV
Memory reallocation	Allocating memory space during runtime.	SpGEMM
Atomic operations	Exclusive execution by one thread.	SpTRANS, SpTRSV
Dynamic characteristics	Dynamically processing different parts during execution.	PR, GC, CC, BFS

3 Evaluation Methodology

3.1 Platform

We in this evaluation use a heterogeneous processor, AMD Ryzen 5 2400G APU, composed of four Zen CPU cores and 11 GPU cores running at 3.6 GHz and 1.25 GHz, respectively. Each CPU core can run two simultaneous threads, and each GPU core has 64 AMD GCN cores. The system memory is 16GB dual-channel DDR4-2933 of theoretical peak bandwidth 46.9 GB/s. The operating system is 64-bit Microsoft Windows 10[1]. The GPU driver version is 18.5.1. The development environment is AMD APP SDK 3.0 and OpenCL 2.0.

3.2 Sparse Kernels

To evaluate *load balancing* and *indirect addressing*, we on the CPU part benchmark a classic row-wise CSR SpMV algorithm and a CSR5 SpMV algorithm proposed by Liu and Vinter [31] parallelized with OpenMP and vectorized by compiler, and on the GPU part test the other two SpMV kernels, i.e., the CSR-adaptive algorithm proposed by Greathouse and Daga [15] and the CSR5 SpMV algorithm. The CSR-adaptive algorithm collects short rows into groups to shrink gaps of row lengths for better load balancing, and the CSR5 SpMV algorithm evenly divides nonzeros into small tiles of the same size for load balancing and uses vectorized segmented sum for utilizing wide SIMD units on GPUs. We can observe the impact of load balancing by comparing these algorithms, because their algorithms use different load balancing strategies. For the impact of indirect addressing, we can analyze the performance difference between the CPU and the GPU, because they have different data access patterns thus resulting in different performance behavior to indirect addressing.

To test *memory reallocation*, we can analyze an application that involves memory reallocation. We run an SpGEMM algorithm developed by Liu and Vinter [30] that calculates the number of floating point operations of each row, groups rows of similar number of operations into the same bin, and use different methods for rows in the same bin. The rows requiring more computations may need to allocate larger space but finally waste the space since the final result can be much shorter. Thus it is better to pre-allocate a small space and re-allocate for larger space when and only when the small space is inadequate. Because GPUs lack the ability to re-allocate memory, the program has to allocate a larger space, copy the entries from the current space, and finally release the old space. This method is actually inefficient and wastes memory space. To avoid the slow processing, Liu and Vinter [30] exploit the re-allocation scheme on the host memory to accelerate the procedure.

To benchmark *atomic operations*, we use two kernels that involves atomic operations: an atomic-based SpTRANS method described by Wang et al. [49] and a synchronization-free SpTRSV algorithm proposed by Liu et al. [28]. The SpTRANS method first uses atomic-add operations to sum the number of nonzeros in each column (assuming both the input and output matrices are in row-major) and then scatters nonzeros in rows into columns through an atomic-based counter. The SpTRSV problem is inherently sequential. Its synchronization-free SpTRSV algorithm uses atomic operations as a communication mechanism between thread groups. When a thread group finishes its

[1] Since the Linux GPU driver of this integrated GPU is not officially available yet, we have to do all benchmarks on Microsoft Windows.

work, it will atomically update some variables in global memory, and some other thread groups that are busy-waiting will notice the change and start to complete their jobs.

To evaluate *dynamic characteristics*, we use four graph computing algorithms in our experiment. Different graph applications may exhibit various dynamic characteristics [48]. PageRank involves all vertices of a graph in computation; graph coloring and connected component have large number of active vertices at first, and then the value decreases; BFS has a low parallelism at first iterations, and then the parallelism increases. These dynamic characteristics make the GPU acceleration challenging.

3.3 Matrices

We use matrices downloaded from the SuiteSparse Matrix Collection [10] (formerly known as the University of Florida Sparse Matrix Collection). There are currently 2757 matrices in the collection. To avoid experimental errors from executing small matrices, we only select relatively large matrices of no less than 100,000 and no more than 200,000,000 nonzero elements. With this condition, 956 matrices are selected and tested to obtain statistically significant experimental results. These matrices are used as the input for the sparse kernels in Section 3.

We focus on the effect of matrix variance on the sparse kernels. We use the metric of variation of row length to represent the variance of a matrix, which can be described in Equation 1. In this equation, μ represents the average of the number of non-zero elements in each row, n represents the number of rows, and e_i represents the number of non-zero elements in row i. Similar definition has been used in [34].

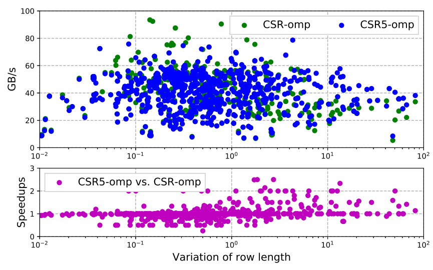
$$variation = \frac{1}{\mu}\sqrt{\frac{\sum_0^{n-1}(e_i - \mu))^2)}{n}} \tag{1}$$

4 Experimental Results

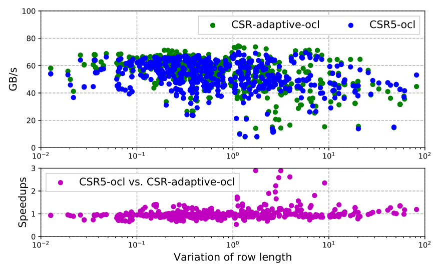
4.1 SpMV Performance and Analysis

We first demonstrate a series of performance data of SpMV operation in Figure 3. The x-axis of each sub-figure is variation of row length of the input matrix. This value depends on distribution of the nonzero entries of the matrix. A smaller value means the rows are of similar length, and larger value means the rows are in a power-law distribution. When rows are calculated in parallel, larger variation of row length may lead to serious load imbalance.

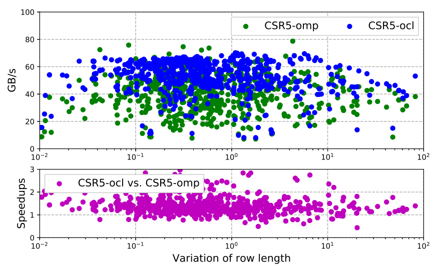
Figure 3 (a) plots a performance comparison of two SpMV methods, i.e., `CSR-omp` and `CSR5-omp`, on the CPU side, and Figure 3 (b) shows a similar performance comparison of two SpMV methods, i.e., `CSR-adaptive-ocl` and



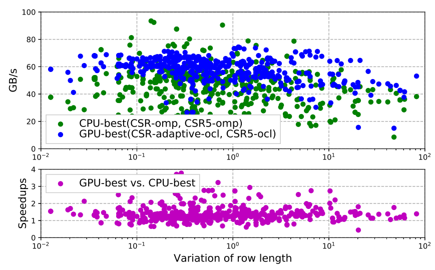
(a) SpMV performance on the CPU part.



(b) SpMV performance on the GPU part.



(c) A performance comparison of the CSR5 SpMV algorithm on CPU and GPU parts.



(d) A performance comparison of the best SpMV algorithm on CPU and GPU parts.

Fig. 3 SpMV experimental data to analyze load balancing and indirect accessing.

`CSR5-ocl`, on the GPU side. It can be seen that the methods deliver comparable performance on both parts. But it is also noticeable that when variation of row length is larger than 1 (shown as 10^0 in the figures), `CSR5-omp` and `CSR5-ocl` methods outperform `CSR-omp` and `CSR-adaptive-ocl` in many cases. However, when the variation is smaller than 10^0, `CSR-omp` and `CSR-adaptive-ocl` give better performance in some matrices. This means that even on moderate scale parallel devices of four CPU cores or of 11 GPU cores, load balancing is still a problem, and CSR5 as a load balanced method is more competitive than the naïve CSR and CSR-adaptive methods. However, it is also worth to note that the two algorithms work better for regular problems since they avoid complex operations designed for load balanced calculation.

As for indirect accessing, it is also interesting to see the performance difference of `CSR5-omp` and `CSR5-ocl` in Figure 3 (c). Although the two methods access the same DRAM thus utilize the same bandwidth, `CSR5-ocl` in most cases offers better performance and gives up to 3x speedup over `CSR5-omp`. This may be due to that the GPU runs much more simultaneous threads than the CPU, thus can hide latency of randomly accessing vector x. However, it can also be seen that `CSR5-omp` can achieve higher performance in several cases (see the green dots near 80 GB/s). This may be due to cache locality on CPUs is better than that on GPUs [22,21], and sparsity structure of the several matrices can exploit caches better.

Finally, we plot the best performance on both sides (`CPU-best` denotes the best performance of `CSR-omp` and `CSR5-omp`, and `GPU-best` denotes the best performance of `CSR-adaptive-ocl` and `CSR5-ocl`) in Figure 3 (d). It can be seen that `GPU-best` in general outperforms `CPU-best`, but the latter can achieve higher performance (`CPU-best`'s near 100 GB/s over `GPU-best`'s no more than 80 GB/s). We believe that this is also due to the effects of CPU's cache and GPU's higher amount of execution threads.

4.2 SpGEMM Performance and Analysis

We now list SpGEMM performance in Figure 4. The `Nonunified mem` label denotes the original implementation of the allocation-copy-release scheme, and the `Unified mem` label denotes using the CPU-side reallocation function for rows requiring larger memory space. The x-axis is compression rate, meaning that the matrices on the right side needs more merge operations over insertion operations when accumulating intermediate products.

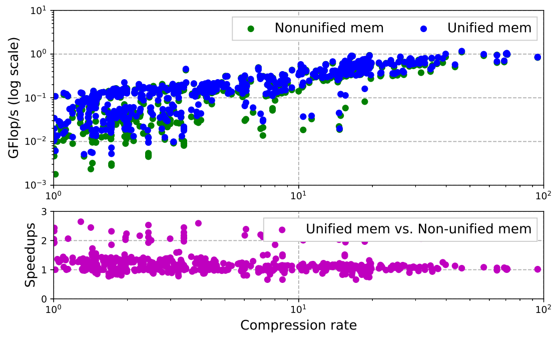
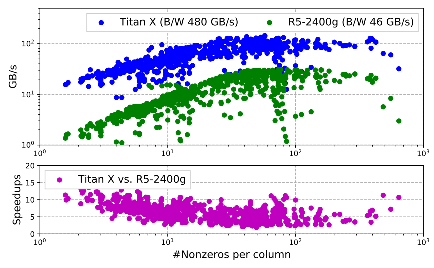


Fig. 4 SpGEMM performance comparison of using uniformed memory or not for memory reallocation.

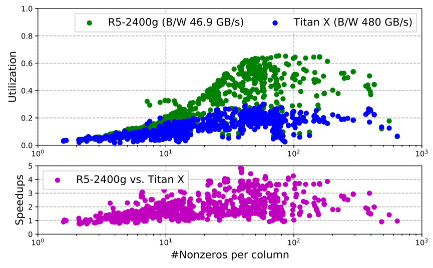
It can be seen that in most cases, the two methods deliver comparable performance. This is due to only very few very long rows requiring the progressive allocation, and the overall performance is not affected by those rows. However, there are still many cases that can receive over 2x speedups from the reallocation on unified memory. This means that the memory reallocation technique can be very useful for irregular problems such as SpGEMM. Because the integrated GPU could use shared host memory, original algorithms designed for GPUs can be further accelerated on heterogeneous processors.

4.3 SpTRANS and SpTRSV Performance and Analysis

We in Figures 5 and 6 plot a performance comparison of the same SpTRANS and SpTRSV methods implemented in CUDA and OpenCL, and they are executed on an NVIDIA Titan X GPU (3584 CUDA cores running at 1.4 GHz, and 12GB GDDR5X of bandwidth 480 GB/s) and the AMD heterogeneous processor. The x-axis of Figure 5 is the number of nonzeros per column, meaning that matrices become more dense from left to right. The x-axis of Figure 6 is parallelism, meaning that on its right side more components of the solution vector x can be solved out in parallel. Note that the execution time of the algorithms benchmarked is dominated by atomic operations.

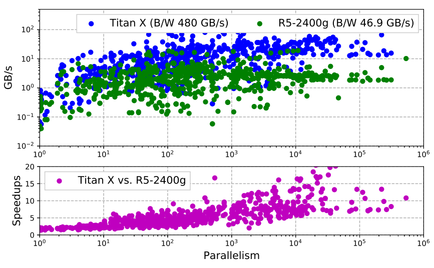


(a) SpTRANS performance on integrated and discrete GPUs.

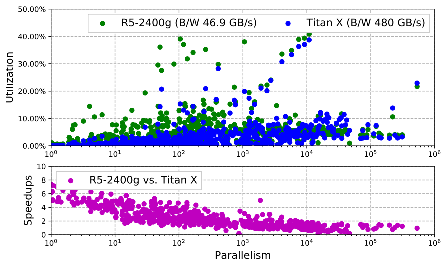


(b) SpTRANS bandwidth utilization on integrated and discrete GPUs.

Fig. 5 SpTRANS experimental data to analyze performance behaviors of atomic operations.



(a) SpTRSV performance on integrated and discrete GPUs.



(b) SpTRSV bandwidth utilization on integrated and discrete GPUs.

Fig. 6 SpTRSV experimental data to analyze performance behaviors of atomic operations.

Figure 5 (a) and Figure 6 (a) demonstrate absolute performance. It can be seen that the Titan X GPU can be nearly up to 20x faster than the integrated GPU, but also in many cases delivers only a couple of times speedups. But in Figure 5 (b) and Figure 6 (b), it is clear to see that the AMD integrated GPU gives much better bandwidth utilization than the NVIDIA discrete GPU. Overall, when the discrete GPU offers 10x more theoretical bandwidth over the integrated GPU, SpTRANS and SpTRSV heavily dependent on atomic operations do *not* receive benefits of higher bandwidth. Although we lack implementation details of atomic operations on both architectures, it is very interesting to see that the relatively low end integrated GPU accessing host memory can bring such high atomic memory utilization.

4.4 PageRank Performance and Analysis

The performance results of PageRank is shown in Figure 7. It shows the performance comparison of GPU results using OpenCL (denoted as GPU PageRank-ocl) and CPU results using OpenMP (denoted as CPU PageRank-omp). The horizontal axis represents the variation of edges in each node; this is the same as the variation of row length in SpMV (Section 4.1), because graphs are represented as sparse matrices, where vertices correspond to rows while edges correspond to nonzero entries. The vertical axis represents the performance using giga-traversed edges per second (GTEPS).

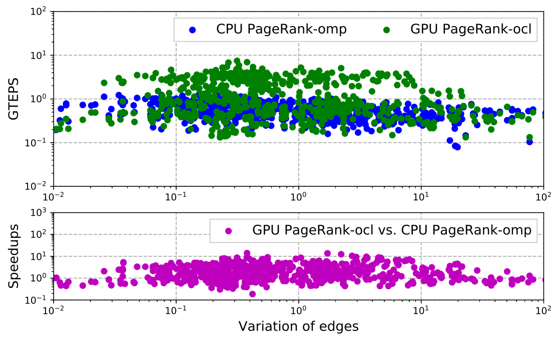


Fig. 7 PageRank experimental data to analyze performance behaviors of graph programs.

From Figure 7, we can see that PageRank on GPUs is more likely to have high performance when the variation of edges is low; in contrast, PageRank on CPUs does not have obvious performance variation along with the variation of edges. This is due to the architecture difference. GPUs have massive parallel computing cores, and a group of cores are required to execute in an SIMD manner, which means that threads in a co-execution group ("wavefront" in OpenCL terminology) need to execute the same instruction simultaneously. However, when the workload distribution is unbalanced in the co-execution group, the threads that finish workload earlier needs to wait for the others threads to process next workloads, which causes performance degradation.

Among graph computing applications, like PageRank, this problem can be more serious. Instead, CPU performance in Figure 7 keeps in a small fluctuation along with the variation of edges. CPUs do not have such problem due to the out-of-order execution model and relatively higher cache capacity. The highest performance the GPUs can reach is 7.5 GTEPS, while the CPUs can only reach 1.2 GTEPS.

Figure 7 shows the performance speedup from GPU to CPU on the integrated architecture. The average performance speedup is about 2.3x, and the highest speedup is 14.0x. From Figure 7, we can see that when the variation of edges is low, the GPU can better release its compute capacity, thus has high speedup to the CPU.

4.5 Graph Coloring Performance and Analysis

Different from PageRank, graph coloring does not involve all vertices during computation. It at first involves all vertices to do coloring; during iterations, the vertices that have been colored successfully do not need to be involved. Hence, the number of active vertices decreases as along with the proceeding of the iterations. Because program parallelism relates to the number of vertices, the GPU performance can be affected.

Figure 8 shows the performance comparison between the GPU and the CPU on integrated architectures for graph coloring. From Figure 8, we can see that the GPU performance has a clear decreasing trend along with the increase of the variation of edges of each node. When the variation of edges is 10^{-2}, the average performance for GPU is 0.2 GTEPS; when the variation of edges is 10^{2}, the average performance for GPU decreases to 0.003 GTEPS. This is due to the irregularity of the graph and the GPU architecture. When the variation of edges is low, which means that the graphs are relatively regular, graph coloring process can have more active vertices to utilize the GPU parallelism capacity than that from an irregular graph. In contrast, when the variation of edges is high, in some iterations, the number of active vertices can be low, which affects the GPU performance. Different from the GPU, the CPU remains a steady performance.

Figure 8 shows the performance speedup from the GPU to the CPU. Because the CPU part has a relatively steady performance, the performance decreasing trend on the GPU is obvious. The highest performance speedup can be 10.6x when the variation of edges is around 0.3; however, when the variation of edges is higher than 10^{1}, the CPU has a better performance than the GPU. Therefore, the variation of edges can be an important indicator for graph coloring to decide whether to run on GPUs or on CPUs for heterogeneous processors.

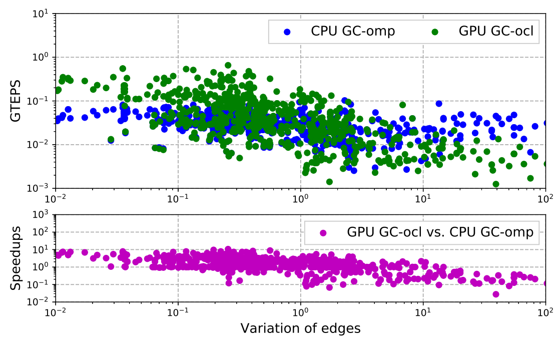


Fig. 8 Graph coloring experimental data to analyze performance behaviors of graph programs.

4.6 Connected Component Performance and Analysis

Connected component calculates the components that each vertex belongs to. In graph theory, a connected component is a subgraph where any two vertices in it can be connected by an path. In this algorithm, we assign each vertex a number; the algorithm consists of several iterations, and during each iteration, each vertex compares its number to that of its neighbors and then updates its number to the smaller value. The algorithm stops when the graph does not change, and at last, the number of unique values represents the number of connected components. The parallelism trend is similar to graph coloring; at first, all vertices need to update and the algorithm has a large parallelism; after several iterations, most components are fixed, and the parallelism can be low.

Figure 9 shows the performance results of connected component. It shows the performance comparison between the GPU OpenCL version (`GPU CC-ocl`) and the CPU OpenMP version (`CPU CC-omp`) of connected component. The performance trend is similar to that from graph coloring: the GPU performance has a clear decreasing trend along with the increase of the variation of edges of each node. The reason is that graphs with high variation of edges are likely to consume more iterations for computation, and the last few iterations do not have high parallelism. For the CPU device, the performance upper bound is relatively stable.

Figure 9 shows the performance speedup from GPUs to CPUs. We can see a clear trend that the speedup decreases along with the increase of the variation of edges. When the variation of edges is about 10^{-2}, the speedup is 4.7x on average; however, when the variation is around 10^2, the speedup decreases to 0.05x.

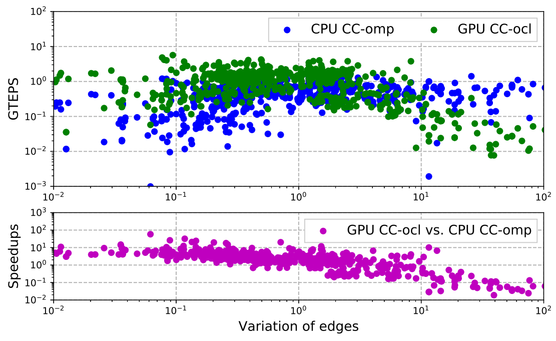


Fig. 9 Connected component experimental data to analyze performance behaviors of graph programs.

4.7 BFS Performance and Analysis

BFS traverses the graph from a root to the other vertices to obtain the shortest path between the root and the other vertices. BFS explores vertices based on their distance to the root. BFS includes several iterations for computation, and during each iteration, it includes the unvisited neighbors to the set of visited vertices. The newly involved vertices at each iteration form a frontier for the next iteration, and only the neighbors to the frontier needs to be explored. The exploration of each vertex in the frontier can be distributed to different threads in parallel, so the frontier size relates to the parallelism. Hence, GPU performance may be affected due to inadequate active vertices. Moreover, compared to other applications, BFS has a low computation density; most operations in BFS relate to memory access.

Figure 10 shows the performance comparison between the CPU and the GPU on heterogeneous processors. From Figure 10, we can see that in most cases, the CPU OpenMP version (`CPU BFS-omp`) achieves better performance than the GPU OpenCL version (`GPU BFS-ocl`) does. The reason is that both the GPU and the CPU share the same physical memory, and thus have the same memory bandwidth. Because the frontier size may influence the power of parallelism of GPUs in some iterations, the GPU part does not achieve as high performance as the CPU part does on heterogeneous processors. However, in some cases, such as the performance around the variation of 10^{-2}, the GPU still performs better than the CPU does; this implies that for BFS, relatively regular input can have larger frontier size on average, which makes GPUs exert higher performance.

Figure 10 presents the performance speedup from GPUs to CPUs on heterogeneous processors. It denotes that when both the GPUs and the CPUs share the same memory, the CPUs have better performance than the GPUs do for BFS. In addition, when the input is relatively regular (a low variation of edges), GPUs can still outperform CPUs.

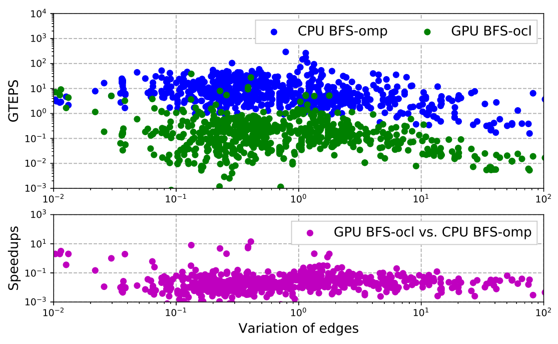


Fig. 10 BFS experimental data to analyze performance behaviors of graph programs.

4.8 Comparison between Scientific Computing kernels and Graph Computing Kernels

From Section 4.4 to Section 4.7, we can see that graph computing kernels are more complex than scientific linear algebra kernels. Linear algebra programs usually launch kernel once, and use the whole sparse matrix for computation; graph computing kernels can be divided into several iterations, and these iterations may have different parallelism. For PageRank, the computation in each iteration involves all vertices, so PageRank has similar performance compared to linear algebra kernels. For graph coloring and connected component, the parallelism has a decreasing process, and each iteration may not involve all vertices for computation; hence, they have better performance on GPUs than on CPUs when the variation is not large, and vice versa. BFS has a low parallelism at first, and then the parallelism increases. Another reason why GPUs have lower performance for BFS than CPUs is that BFS is a memory bound program; most operations in BFS relate to memory access. Because CPUs and GPUs share the same bandwidth, higher architecture parallelism has a negative influence on the bandwidth utilization.

5 Related Work

5.1 Performance Analysis for Coupled Heterogeneous Processors

Because coupled heterogeneous processors pose non-trivial challenges from both programming and architecture aspects, many researchers focus on performance analysis on the coupled heterogeneous processors to understand their performance behaviors for optimizations. Daga et al. [6] analyzed the efficiency of the coupled heterogeneous processors, and pointed out that such heterogeneous processor is a step in the right direction for efficient supercomputers. Doerksen et al. [11] used 0-1 knapsack and Gaussian elimination as two examples to discuss the design and performance on fused architectures. Spafford et al. [45] studied the tradeoffs of the shared memory hierarchies on coupled heterogeneous processors, and identified a significant requirement for robust runtime systems on such architectures. Lee et al. [19] provided a comprehensive performance characterization for data-intensive applications, and revealed that the fused architecture is promising for accelerating data-intensive applications. Zakharenko et al. [53] used FusionSim [52] to characterize the performance on fused and discrete architectures. Mekkat et al. [33] analyzed the management policy for the shared last level cache. Zhang et al. [56,57] studied the co-running behaviors of different devices for the same application, while Zhu et al. [59,58] studied co-running performance degradation for different devices for separate applications. Garzó et al. [14] proposed an approach to optimize the energy efficiency of iterative computation on heterogeneous processors. Zhu et al. [60] presented a systematic study on heterogeneous processors with power caps considered. Moreover, low-power, reliability, and performance/watt ratio optimization are also crucial considerations [4,60,25,26]. Different from these research, our study focuses on sparse matrix and graph kernels. We analyze the load balancing, indirect addressing, memory reallocation, atomic operations, and the difference between those kernels.

5.2 Accelerating Irregular Applications on Heterogeneous Processors

Many researchers focus on the optimization for applications on coupled heterogeneous processors. Kaleem et al. [17] provided an adaptive workload dispatcher for heterogeneous processors to co-run CPUs and GPUs together; Pandit et al. [37] proposed Fluidic Kernels, which can perform cooperative execution on multiple heterogeneous devices; this work can be applied on heterogeneous processors directly. However, these research does not consider the optimization for workload irregularity. Shen et al. [43] provided Glinda, which is a framework for accelerating imbalanced applications on heterogeneous platforms. Barik et al. [2] tried to map irregular C++ applications to the GPU device on heterogeneous processors. Fairness and efficiency are two major concerns for shared system users; Tang et al. [46] introduced multi-resource fairness and efficiency on heterogeneous processors. Zhang et al. [55] considered the irregularity inside workload and architecture differences between CPUs and GPUs, and then proposed a method that can distribute the relatively regular part of workload to GPUs while remain irregular part to CPUs on integrated architectures. Daga et al. [8] proposed a hybrid BFS algorithm that can select appropriate algorithm and devices for iterations on heterogeneous processors. Zhang et al. [54] further developed a performance model for BFS algorithm on heterogeneous processors.

5.3 Accelerating Irregular Applications on Discrete GPUs

There are many works about accelerating irregular algorithms in sparse matrix and graph computations. For example, Liu and Vinter [31] developed

CSR5, which is an efficient storage format for SpMV on heterogeneous platforms. Sparse matrix-matrix multiplication (SpGEMM) is another fundamental building block for scientific computation, and Liu and Vinter [30] proposed a framework for SpGEMM on GPUs and integrated architectures. Shen et al. [44, 42] proposed a method to match imbalanced workloads for GPUs, and performed workload partitioning for accelerating applications.

6 Conclusions

We in this work have conducted a thorough empirical evaluation of four representative sparse matrix kernels, i.e., SpMV, SpTRANS, SpTRSV, and SpGEMM, and four graph computing kernels, i.e., PageRank, Connected Component, Graph Coloring, and BFS, on an AMD APU heterogeneous processors. We benchmarked 956 sparse matrices and obtained experimental results which are statistically significant. Based on the data, we analyzed performance behaviors of the kernels' load balancing, indirect addressing, memory reallocation, atomic operations, and dynamic characteristics on heterogeneous processors, and identified several interesting insights.

7 Acknowledgments

This work has been partly supported by the National Natural Science Foundation of China (Grant No. 61732014, 61802412, 61671151), Beijing Natural Science Foundation (No. 4172031), and SenseTime Young Scholars Research Fund.

References

1. Agarwal, N., Nellans, D., Ebrahimi, E., Wenisch, T.F., Danskin, J., Keckler, S.W.: Selective GPU caches to eliminate CPU-GPU HW cache coherence. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 494–506 (2016)
2. Barik, R., Kaleem, R., Majeti, D., Lewis, B.T., Shpeisman, T., Hu, C., Ni, Y., Adl-Tabatabai, A.R.: Efficient mapping of irregular C++ applications to integrated GPUs. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p. 33. ACM (2014)
3. Boggs, D., Brown, G., Tuck, N., Venkatraman, K.S.: Denver: Nvidia's First 64-bit ARM Processor. IEEE Micro **35**(2), 46–55 (2015)
4. Branover, A., Foley, D., Steinman, M.: AMD Fusion APU: Llano. IEEE Micro **32**(2), 28–37 (2012)
5. Buluç, A., Gilbert, J.: Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. SIAM Journal on Scientific Computing **34**(4), C170–C191 (2012)
6. Daga, M., Aji, A.M., c. Feng, W.: On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In: 2011 Symposium on Application Accelerators in High-Performance Computing, pp. 141–149 (2011)

7. Daga, M., Nutter, M.: Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 240–247 (2012)
8. Daga, M., Nutter, M., Meswani, M.: Efficient breadth-first search on a heterogeneous processor. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 373–382 (2014)
9. Dashti, M., Fedorova, A.: Analyzing Memory Management Methods on Integrated CPU-GPU Systems. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, pp. 59–69 (2017)
10. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
11. Doerksen, M., Solomon, S., Thulasiraman, P.: Designing APU oriented scientific computing applications in opencl. In: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, pp. 587–592. IEEE (2011)
12. Doweck, J., Kao, W., Lu, A.K., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A., Yoaz, A.: Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. IEEE Micro **37**(2), 52–62 (2017)
13. Duff, I.S., Heroux, M.A., Pozo, R.: An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. ACM Transactions on Mathematical Software (TOMS) **28**(2), 239–267 (2002)
14. Garzón, E.M., Moreno, J., Martínez, J.: An approach to optimise the energy efficiency of iterative computation on integrated GPU–CPU systems. The Journal of Supercomputing **73**(1), 114–125 (2017)
15. Greathouse, J.L., Daga, M.: Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pp. 769–780 (2014)
16. Gregg, C., Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, pp. 134–144 (2011)
17. Kaleem, R., Barik, R., Shpeisman, T., Hu, C., Lewis, B.T., Pingali, K.: Adaptive heterogeneous scheduling for integrated GPUs. In: Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on, pp. 151–162. IEEE (2014)
18. Krishnan, G., Bouvier, D., Naffziger, S.: Energy-Efficient Graphics and Multimedia in 28-nm Carrizo Accelerated Processing Unit. IEEE Micro **36**(2), 22–33 (2016)
19. Lee, K., Lin, H., Feng, W.c.: Performance characterization of data-intensive kernels on AMD fusion architectures. Computer Science-Research and Development **28**(2-3), 175–184 (2013)
20. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pp. 451–460 (2010)
21. Li, A., Liu, W., Kristensen, M.R.B., Vinter, B., Wang, H., Hou, K., Marquez, A., Song, S.L.: Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pp. 26:1–26:14 (2017)
22. Li, A., Song, S.L., Liu, W., Liu, X., Kumar, A., Corporaal, H.: Locality-Aware CTA Clustering for Modern GPUs. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, pp. 297–311 (2017)
23. Liu, J., He, X., Liu, W., Tan, G.: Register-based implementation of the sparse general matrix-matrix multiplication on gpus. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, pp. 407–408 (2018)
24. Liu, J., He, X., Liu, W., Tan, G.: Register-aware optimizations for parallel sparse matrix–matrix multiplication. International Journal of Parallel Programming (2019)

25. Liu, T., Chen, C.C., Kim, W., Milor, L.: Comprehensive reliability and aging analysis on SRAMs within microprocessor systems. Microelectronics Reliability **55**(9), 1290–1296 (2015)
26. Liu, T., Chen, C.C., Wu, J., Milor, L.: SRAM stability analysis for different cache configurations due to Bias Temperature Instability and Hot Carrier Injection. In: Computer Design (ICCD), 2016 IEEE 34th International Conference on, pp. 225–232. IEEE (2016)
27. Liu, W.: Parallel and scalable sparse basic linear algebra subprograms. Ph.D. thesis, University of Copenhagen (2015)
28. Liu, W., Li, A., Hogg, J.D., Duff, I.S., Vinter, B.: Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides. Concurrency and Computation: Practice and Experience **29**(21), e4244–n/a (2017)
29. Liu, W., Vinter, B.: Ad-heap: An Efficient Heap Data Structure for Asymmetric Multicore Processors. In: Proceedings of Workshop on General Purpose Processing Using GPUs, GPGPU-7, pp. 54:54–54:63 (2014)
30. Liu, W., Vinter, B.: A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. Journal of Parallel and Distributed Computing **85**(C), 47–61 (2015)
31. Liu, W., Vinter, B.: CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In: Proceedings of the 29th ACM International Conference on Supercomputing, ICS '15, pp. 339–350 (2015)
32. Liu, W., Vinter, B.: Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors. Parallel Computing **49**(C), 179–193 (2015)
33. Mekkat, V., Holey, A., Yew, P.C., Zhai, A.: Managing shared last-level cache in a heterogeneous multicore processor. In: Proceedings of the 22nd international conference on Parallel architectures and compilation techniques, pp. 225–234. IEEE Press (2013)
34. Merrill, D., Garland, M.: Merge-based parallel sparse matrix-vector multiplication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 58. IEEE Press (2016)
35. Nai, L., Xia, Y., Tanase, I.G., Kim, H., Lin, C.Y.: GraphBIG: understanding graph computing in the context of industrial solutions. In: High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for, pp. 1–12. IEEE (2015)
36. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE **96**(5), 879–899 (2008)
37. Pandit, P., Govindarajan, R.: Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p. 273. ACM (2014)
38. Power, J., Basu, A., Gu, J., Puthoor, S., Beckmann, B.M., Hill, M.D., Reinhardt, S.K., Wood, D.A.: Heterogeneous system coherence for integrated CPU-GPU systems. In: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 457–467 (2013)
39. Puthoor, S., Aji, A.M., Che, S., Daga, M., Wu, W., Beckmann, B.M., Rodgers, G.: Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture. In: Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, GPGPU '16, pp. 53–62 (2016)
40. Said, I., Fortin, P., Lamotte, J., Calandra, H.: Leveraging the accelerated processing units for seismic imaging: A performance and power efficiency comparison against CPUs and GPUs. The International Journal of High Performance Computing Applications **0**(0) (2017)
41. Schulte, M.J., Ignatowski, M., Loh, G.H., Beckmann, B.M., Brantley, W.C., Gurumurthi, S., Jayasena, N., Paul, I., Reinhardt, S.K., Rodgers, G.: Achieving Exascale Capabilities through Heterogeneous Computing. IEEE Micro **35**(4), 26–36 (2015)
42. Shen, J., Varbanescu, A.L., Lu, Y., Zou, P., Sips, H.: Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. IEEE Transactions on Parallel and Distributed Systems **27**(9), 2766–2780 (2016)
43. Shen, J., Varbanescu, A.L., Sips, H., Arntzen, M., Simons, D.G.: Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In: Proceedings of the ACM International Conference on Computing Frontiers, CF '13, pp. 14:1–14:10 (2013)

44. Shen, J., Varbanescu, A.L., Zou, P., Lu, Y., Sips, H.: Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In: Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14, pp. 241–250 (2014)
45. Spafford, K.L., Meredith, J.S., Lee, S., Li, D., Roth, P.C., Vetter, J.S.: The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In: Proceedings of the 9th conference on Computing Frontiers, pp. 103–112. ACM (2012)
46. Tang, S., He, B., Zhang, S., Niu, Z.: Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 75. IEEE Press (2016)
47. Vijayaraghavan, T., Eckert, Y., Loh, G.H., Schulte, M.J., Ignatowski, M., Beckmann, B.M., Brantley, W.C., Greathouse, J.L., Huang, W., Karunanithi, A., Kayiran, O., Meswani, M., Paul, I., Poremba, M., Raasch, S., Reinhardt, S.K., Sadowski, G., Sridharan, V.: Design and Analysis of an APU for Exascale Computing. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 85–96 (2017)
48. Wang, H., Geng, L., Lee, R., Hou, K., Zhang, Y., Zhang, X.: Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, pp. 38–52 (2019)
49. Wang, H., Liu, W., Hou, K., Feng, W.c.: Parallel Transposition of Sparse Data Structures. In: Proceedings of the 2016 International Conference on Supercomputing, ICS '16, pp. 33:1–33:13 (2016)
50. Wang, X., Liu, W., Xue, W., Wu, L.: 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, PPoPP '18, pp. 338–353 (2018)
51. Yang, Y., Xiang, P., Mantor, M., Zhou, H.: CPU-assisted GPGPU on fused CPU-GPU architectures. In: IEEE International Symposium on High-Performance Comp Architecture, pp. 1–12 (2012)
52. Zakharenko, V.: FusionSim: characterizing the performance benefits of fused CPU/GPU systems. Ph.D. thesis (2012)
53. Zakharenko, V., Aamodt, T., Moshovos, A.: Characterizing the performance benefits of fused CPU/GPU systems using FusionSim. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 685–688. EDA Consortium (2013)
54. Zhang, F., Lin, H., Zhai, J., Cheng, J., Xiang, D., Li, J., Chai, Y., Du, X.: An Adaptive Breadth-First Search Algorithm on Integrated Architectures. The Journal of Supercomputing (2018)
55. Zhang, F., Wu, B., Zhai, J., He, B., Chen, W.: FinePar: Irregularity-aware Fine-grained Workload Partitioning on Integrated Architectures. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO '17, pp. 27–38 (2017)
56. Zhang, F., Zhai, J., Chen, W., He, B., Zhang, S.: 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, pp. 89–92 (2015)
57. Zhang, F., Zhai, J., He, B., Zhang, S., Chen, W.: Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. IEEE Transactions on Parallel and Distributed Systems **28**(3), 905–918 (2017)
58. Zhu, Q., Wu, B., Shen, X., Shen, K., Shen, L., Wang, Z.: Understanding co-run performance on cpu-gpu integrated processors: observations, insights, directions. Frontiers of Computer Science **11**(1), 130–146 (2017)
59. Zhu, Q., Wu, B., Shen, X., Shen, L., Wang, Z.: Understanding co-run degradations on integrated heterogeneous processors. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 82–97. Springer (2014)
60. Zhu, Q., Wu, B., Shen, X., Shen, L., Wang, Z.: Co-Run Scheduling with Power Cap on Integrated CPU-GPU Systems. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 967–977 (2017)