Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures

Feng Zhang, Jidong Zhai, Bo Wu, Bingsheng He, Wenguang Chen, Xiaoyong Du

Abstract—The integrated architecture that features both CPU and GPU on the same die is an emerging and promising architecture for fine-grained CPU-GPU collaboration. However, the integration also brings forward several programming and system optimization challenges, especially for irregular applications such as graph processing. The complex interplay between heterogeneity and irregularity leads to very low processor utilization of running irregular applications on integrated architectures. Furthermore, fine-grained co-processing on the CPU and GPU is still an open problem. Particularly, in this paper, we show that the previous workload partitioning for CPU-GPU co-processing is far from ideal in terms of resource utilization and performance. To solve this problem, we propose a system software called FinePar, which considers architectural differences of the CPU and GPU and leverages fine-grained collaboration enabled by integrated architectures. Through irregularity-aware performance modeling and online auto-tuning, FinePar partitions irregular workloads and achieves both device-level and thread-level load balance. We evaluate FinePar with eight irregular applications in graphs and sparse matrices on two integrated architectures and compare it with state-of-the-art partitioning approaches. Results show that FinePar demonstrates better resource utilization and achieves an average of 1.6X speedup over the optimal coarse-grained partitioning method.

Index Terms—Heterogeneous Computing, Integrated Architecture, Irregular Application, Workload Partitioning.

1 Introduction

In recent years, GPUs have made big strides in throughput-oriented computing thanks to the massively parallel architecture. GPUs have been used as a powerful accelerator for many database applications, including relational databases [1], [2], [3], [4], [5], [6] and graph processing [7], [8], [9], [10], [11], [12]. Moreover, *integrated architectures* coupling the CPU and GPU on the same die show great promise to bring the synergy of CPU and GPU to a significantly higher level. The CPU and GPU share the same physical memory, which eliminates the data transfer bottleneck via PCI-e bus in the discrete architecture and eases heterogeneous programming. Therefore, chip vendors have started to release integrated architectures, exemplified by AMD's Accelerated Processing Units (APUs), Intel's Ivy Bridge processor, and Nvidia's Denver architecture.

Integrated architectures have enabled a series of performance optimization opportunities over discrete architectures. First, shared memory makes it possible for different devices to access the same memory space simultaneously. Some integrated architectures have shared cache and embedded DRAM [13], which makes the communication between devices more efficient. Second, the co-processing between the CPU and the GPU can be made more fine-grained. The fine-grained cooperation needs to consider architectural differences between the CPU and GPU for optimal performance. Specifically, the GPU has a large number of processing cores but adopts a lockstep execution model, which forces the threads in the same SIMD (Single Instruction Multiple Data) group to always execute the same instruction. Hence, load imbalance among these threads greatly devastates performance because the performance is limited by the slowest thread. In contrast, the CPU has fewer, yet more powerful cores, and its threading model is more flexible.

Previous work tried to leverage the integrated architecture to accelerate irregular applications [14], [15], [16], [17], [18], [19], [20], [21], [22]. However, the interplay between heterogeneity and irregularity in integrated architectures poses severe technical challenges in the effectiveness of workload partitioning, which existing studies do not well address. First, many previous studies [14], [15], [16], [17], [19] only perform coarse-grained workload partitioning, without considering the fine-grained collaboration between the CPU and GPU. For example, Delorme et al. [19] and Pandit et al. [15] break the workload into many jobs. Each job typically operates on adjacent data and is processed by a work-group (in OpenCL terminology). The work-groups running on the GPU process the jobs from the beginning to the end, while those on the CPU process the jobs in the reverse direction. A runtime makes sure that the whole workload is processed with good load balance. Kaleem et al. [16] addressed more complicated applications and dynamically assigned the jobs to processors through lightweight online profiling. Second, although some studies [18], [20], [21], [22] use fine-grained workload partitioning, they are applied to specific applications only such as hash join in databases [21] and MapReduce [20]. They do not necessarily offer an

- *F. Zhang, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China, Beijing 100872, China. E-mail: {fengzhang,duyong}@ruc.edu.cn.*
- *J. Zhai and W. Chen are with the Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China. E-mail: {zhaijidong, cwg}@tsinghua.edu.cn.*
- *B. Wu is with the Department of Computer Science, Colorado School of Mines, Golden, CO 80401, United States. E-mail: bwu@mines.edu.*
- *B. He is with the School of Computing, National University of Singapore, 119077, Singapore. E-mail: hebs@comp.nus.edu.sg.*
(Corresponding author: Jidong Zhai, and Xiaoyong Du)

automatic or general solution to irregular applications.

In this study, we find that even if such coarse-grained workload partitioning approaches provide optimal load balance between the CPU and GPU, the computational resources may still be under-utilized. For instance, in sparse matrix vector multiplication (SpMV), a job involves the processing of tens or hundreds of adjacent rows. The numbers of non-zero elements in those rows may vary significantly. As a result, if a group of SIMD threads on the GPU process this job, the thread that processes the row with the most non-zero elements slows down all the other threads. Coarse-grained partitioning groups adjacent data (e.g., adjacent rows in SpMV) as a unit for partitioning and hence ignores the irregularity inside each unit. The shared memory on integrated architectures provides an opportunity for the CPU and GPU to co-process data in a fine-grained manner to tackle the problem of resource underutilization.

To fully exploit the benefits of integrated architectures, we propose a fine-grained workload partitioning framework for irregular applications, called FinePar. The basic idea is that we automatically identify the irregular data that introduces load imbalance for GPU threads and assign them to the CPU, while the GPU processes the remaining relatively regular data and enjoys higher performance. To realize this idea, we need to tackle multiple technical issues. First, the partitioning should be transparent to avoid tedious programming burden on users. Second, the framework should introduce no offline pre-processing for practical use, as the input data is typically unavailable until runtime. Third, the partitioning should introduce the minimum runtime overhead. A preliminary exploration has been published in [23], and this paper provides more optimizations, experiments, and implementation details of FinePar, including a more fine-grained partitioning optimization, an adaptive partitioning method for dynamic workloads, and a method to process large datasets.

Our framework employs the following key techniques: 1) We design a program transformation to automatically transform the given OpenCL program to enable fine-grained partitioning; 2) We build performance models to predict the performance of the CPU and GPU given any specific fine-grained partitioning; 3) We design an auto-tuner to guide the fine-grained workload partitioning for load balancing between the CPU and GPU. In addition, we also integrate a series of optimization strategies into FinePar.

As case studies, we focus on sparse matrix and graph processing applications. We evaluate FinePar with eight irregular applications for typical input matrices and compare it with four state-of-the-art workload partitioning methods on two integrated architectures. Results show that FinePar demonstrates better resource utilization and achieves an average of 1.6X speedup over the optimal coarse-grained partitioning method. Meanwhile, FinePar is very lightweight, only introducing less than 6% space overhead and 3% time overhead. In summary, we make the following contributions in this work:

- We propose a fine-grained workload partitioning that takes advantage of the special features of integrated architectures.
- We propose irregularity-aware performance modeling that takes architectural differences between the CPU and GPU into consideration.
- We integrate those techniques into the software framework, called FinePar, which automatically partitions the workload for irregular applications with well controlled space and time overhead.
- We further integrate a series of optimization strategies into FinePar, including a more fine-grained partitioning method, an adaptive partitioning for dynamic workloads, and partitioning techniques for large dataset.
- We evaluate FinePar on a set of irregular applications and inputs to demonstrate their benefits over state-of-the-art approaches.

The remainder of this paper is organized as follows. Section 2 reviews integrated architectures and motivates our work. Section 3 describes each component of FinePar in details. Section 4 presents the optimizations. Section 5 shows the evaluation results. Section 6 discusses related work. Section 7 concludes the paper.

2 Background and Motivation

2.1 Integrated Architecture and Execution Models

We focus on the architecture that integrates both the CPU and GPU on the same chip as illustrated in Figure 1. The most beneficial feature of such architecture is the shared physical memory accessible to both the CPU and GPU, which enables fine-grained collaboration between the two processors. Unlike in discrete architectures, the program running on integrated architectures can leverage both devices to accelerate the processing of data in shared memory.

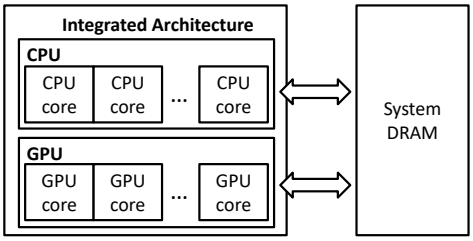
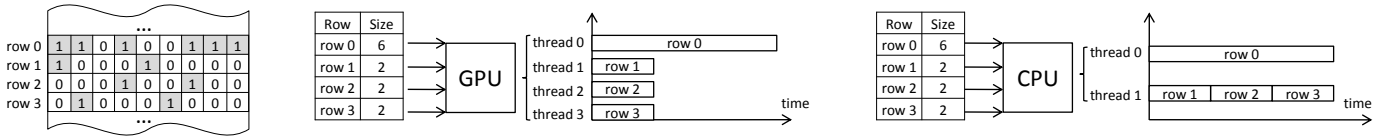


Fig. 1. A general view of the integrated CPU/GPU architecture.

A commonly used programming model for general-purpose computing on integrated architectures is OpenCL, as it is supported by both the CPU and GPU. The main computation of an OpenCL program happens in the kernel function. When a kernel is launched on a device, the OpenCL runtime creates a computation domain of many work-items (i.e., threads), each executing the same kernel function. The computation domain is composed of many work-groups; the work-items belonging to the same work-group can synchronize with each other.

The execution models on the CPU and GPU are different. When a work-group runs on the GPU, its work-items are grouped into wavefronts, each of which runs on the SIMD unit in lockstep. The CPU, on the other hand, creates a thread to perform computation for the whole work-group. When the workload of the work-group is regular, meaning that each item processes the same amount of data, the performance of the GPU is typically several times larger than that of the CPU because of the efficiency of SIMD execution.



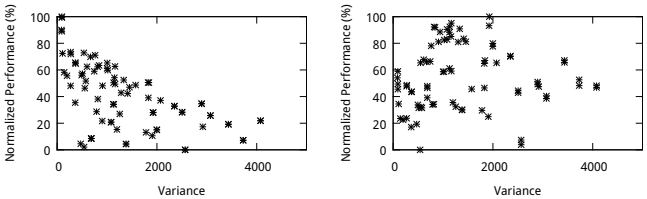
(a) Matrix (b) GPU Execution (c) CPU Execution

Fig. 2. An example to demonstrate the performance features of the CPU and GPU to execute irregular application.

The GPU's performance, however, may degrade significantly when processing irregular applications. We explain the reason through an example depicted in Figure 2. The kernel function performs SpMV (Sparse Matrix-Vector Multiplication) with each work-item processing one row. We assume the matrix is stored in CSR (Compressed Sparse Row) format.

Sparse matrices typically have rather irregular distribution of the non-zero elements. As shown in Figure 2 (a), the first row contains six non-zero elements, while the other three rows only contain two. Figure 2 (b) shows the execution on the GPU. The kernel launch creates four work-items in a wavefront to process the data. The consequence is that the last three work-items need to wait for the first work-item to finish processing all the non-zero elements, wasting 50% of the computational resources. As shown in Figure 2 (c), if a two-core CPU processes the same data, it may create two threads, with the first to process the first row and the second to process the other rows. The CPU threads do not need to wait for each other, as they do not execute in the SIMD fashion.

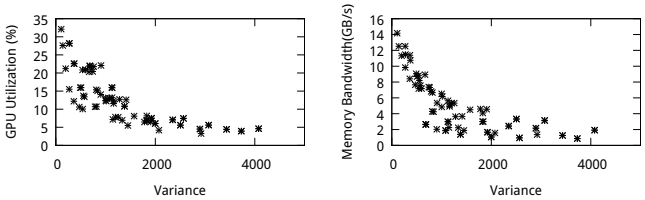
To demonstrate the sensitivity to irregularity for the CPU and GPU on real-world workloads, we run SpMV on the CPU and GPU using 80 different sparse matrices. For each matrix, we treat the number of non-zero elements in a row as a random variable and calculate its variance. Figure 3 shows the performance trend when the variance of input matrices increases. We quantify performance as the number of non-zero elements processed per second. After normalization over the input that yields the best performance, we observe that the GPU's performance drops quickly with the increase of variance, while the CPU's performance trend does not demonstrate a clear impact from the variance.



(a) GPU Performance Trend (b) CPU Performance Trend

Fig. 3. The normalized performance of the CPU and GPU given input matrices with different degrees of irregularity.

Figure 4 shows the GPU utilization and memory bandwidth trends with the increase of variance. The statistics are obtained through the profiler from AMD's CodeXL. Figure 4 (a) shows the percentage of active threads in a wavefront. Larger variances lead to more serious load imbalance for threads in the same wavefront, and hence lower GPU core utilization. Figure 4 (b) shows that because of the workload imbalance, the memory bandwidth is not fully utilized when the variance is high. The results serve as a strong motivation to consider the devices' sensitivity to irregularity when partitioning workloads.



(a) GPU Core Utilization (b) Memory Bandwidth Utilization

Fig. 4. Core utilization and memory bandwidth of the GPU given input matrices with different degrees of irregularity.

2.2 Understanding the Inefficiency of Coarse-Grained Workload Partitioning

Previous work [14], [15], [16], [17] all leverages some form of coarse-grained partitioning to optimize load balance between the CPU and GPU. In the context of sparse matrix processing using OpenCL, those approaches group many adjacent rows as a task to assign to a work group, which serves as a unit for workload partitioning. We show in Figure 5 that even if coarse-grained partitioning achieves optimal load balance, the computational resources may still get under-utilized. The input graph has 8 vertices with varied out-going degrees. Represented as an adjacency matrix, the irregular structure leads to different numbers of non-zero elements in the rows. We assume that two threads run on the CPU and a wavefront of four threads runs on the GPU. We further assume that a CPU thread is 1.5X more powerful than a GPU core, meaning that a GPU thread needs 50% extra time to process the same number of non-zero elements compared to a CPU thread.

If we want to achieve load balance between the CPU and GPU, we can group the first four rows as a job to allocate to the CPU (shown as coarse-grained partitioning), with the remaining four rows to form a job for the GPU to process. As Figure 5 shows, the slowest CPU thread (the first one) finishes at the same time as the slowest GPU thread (the first one). However, the other three GPU threads are seriously under-utilized due to the lockstep execution model. Hence, we conclude that coarse-grained partitioning has two pitfalls. First, it does not consider the irregularity of the data input (in this case demonstrated by the non-uniform distribution of the non-zero elements). Second, it does not fully exploit the capability of the integrated architecture to enable fine-grained collaboration between the CPU and GPU (demonstrated by only grouping adjacent rows into jobs in the example).

Figure 5 also demonstrates the performance gain from fine-grained partitioning. The new partitioning assigns rows 0 and 4 to the CPU threads and the remaining rows to the GPU. Note that the processing time of row 3 on the CPU is two thirds of that on the GPU due to the CPU's faster single-core performance. For the same reason, the execution time of rows 1, 2, and 3 is lengthened by 50% on the GPU. As in coarse-grained partitioning, the load balance remains optimal because the CPU and GPU finish processing at the

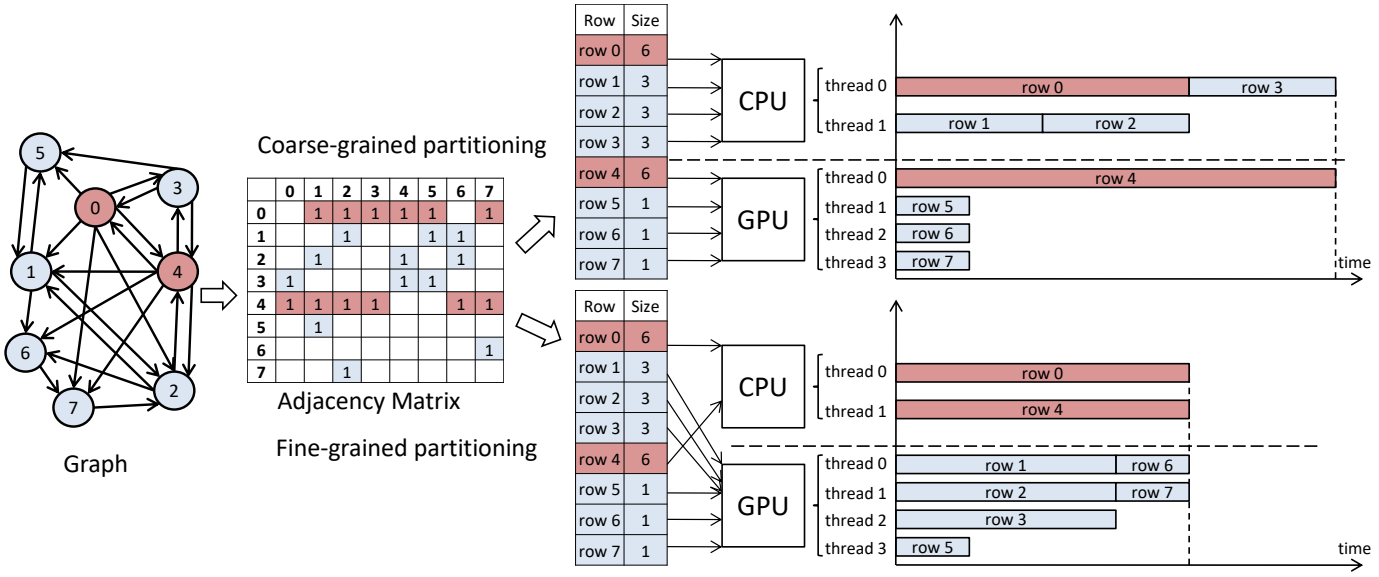


Fig. 5. An example to show the benefits from fine-grained partitioning.

same time. However, fine-grained partitioning improves the overall performance by 1.5X through better utilization of the GPU resources.

2.3 Demand for an Automatic Fine-Grained Partitioning Framework

The idea of fine-grained workload partitioning is simple, but manually realizing it places a non-trivial burden on programmers. While coarse-grained workload partitioning distributes workload to the GPU and CPU for load balance, fine-grained workload partitioning also needs to select irregular data for CPU processing to improve the performance of the GPU, which introduces several challenges. First, the irregularity of the input data is unknown until runtime. Second, the low-level load balance for the GPU threads and the high-level load balance between the CPU and GPU are both critical for performance. Third, the runtime cost incurred by the partitioning should be well controlled not to outweigh the benefit. To address the challenges, we design and implement FinePar by following four guidelines.

- FinePar should automatically transform the input program to enable fine-grained partitioning. The user only needs to focus on the functionality of the program instead of the partitioning for optimized performance.
- FinePar should partition the relatively regular workload to the GPU and the remaining workload to the CPU and still guarantee good load balance between the two processors.
- FinePar should assume no prior knowledge of the irregularity distribution in the input and identify various input features through online tuning.
- FinePar's runtime optimization should only incur marginal time and space overhead.

3 FinePar Framework

3.1 Overview

Figure 6 shows the overview of FinePar. To use the system, the only job for the user is to feed into FinePar the target OpenCL program and a set of representative inputs to train the framework for optimized performance. Once the training is done, FinePar automatically partitions the given input during runtime to optimize the utilization of the integrated architecture.

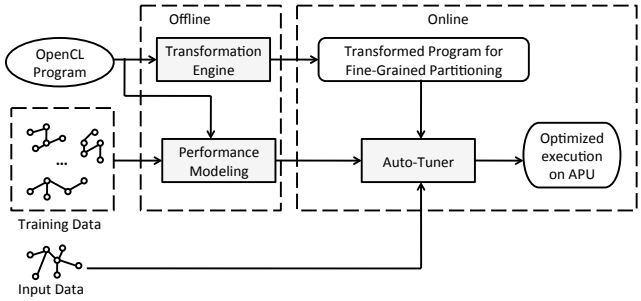


Fig. 6. The overview of FinePar.

The FinePar framework consists of three major components, transformation engine, performance modeling, and auto-tuner. The FinePar transformation engine and the performance modeling components are used in the offline stage. The transformation engine transforms the input OpenCL program to enable fine-grained partitioning. More specifically, the transformed code takes a parameter as the irregularity threshold (to be detailed in Section 3.2). The more irregular part of the data and the less irregular part are dispatched to the CPU and GPU, respectively. The performance modeling component takes both architecture differences and data irregularity into consideration. It trains itself with the provided training data and builds matrix category-specific performance models for both the CPU and GPU.

The auto-tuner component is active during runtime and completes two tasks. First, it determines the performance model to use based on input sampling. Second, it searches for a partitioning threshold based on the performance model and input features.

3.2 Code Transformation

The goal of the transformation engine is to transform the input irregular OpenCL program to enable workload partitioning in a fine-grained manner. The FinePar framework

can also target coarse-grained partitioning for performance comparison. We present the flowchart of code transformation engine in Figure 7. The transformation engine consists of two main modules: analysis module and transformation module. Analysis module is used to identify computation kernels and related parameters in OpenCL programs, while transformation module is in charge of performing actual code transformation. OpenCL programs usually include two parts: management part (host.c in Figure 7), which contains device and kernel calling information, and computation kernel part (compute.cl in Figure 7), which defines computation kernels. Our transformation engine manipulates two parts separately. Analysis module only parses the management part and identifies main kernels and parameters. Transformation module changes code in both management part and computation kernel part, enabling the fine-grained CPU/GPU co-running. Specifically, it initializes key data structures, launches both CPU and GPU kernels, and finally releases resource in the management part, as well as redefines computation kernels in computation part, shown in Figure 7. Note that FinePar handles inputs that can be represented in the CSR format or adjacency lists. In case of more complex situations, FinePar allows users to provide the kernels instead of directly transforming the kernels. More details are discussed in Section 4.4.

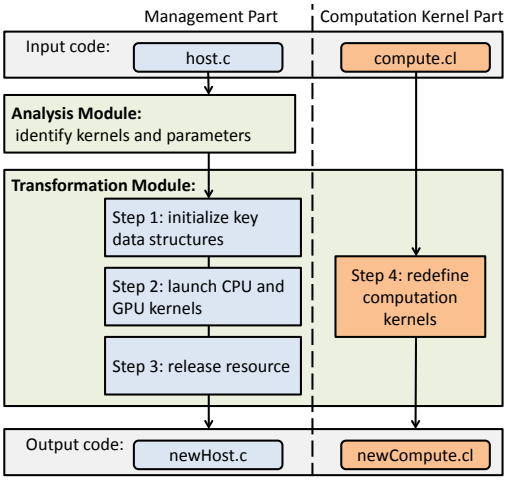


Fig. 7. Illustration of code transformation engine.

Figure 8 presents the basic ideas of the transformations using sparse matrix processing as an example. Figure 8 (a) shows the pseudocode of the original program. The host code initializes the sparse matrix M, and invokes a kernel function to process it. Each work-item executes the same kernel function, which processes the corresponding row according to its global ID. Note that when launching the kernel, the host code needs to specify whether to use the CPU or GPU, but not both.

To utilize both the CPU and GPU resources for coarse-grained partitioning, the framework only needs to slightly change the program as shown in Figure 8 (b). On the host code part, FinePar inserts a function $getCoraseGrainedPartitioningThreshold$ (detailed in Section 5.2), which analyzes the matrix to return a partitioning parameter T_c. Logically, the framework breaks the input matrix M with N rows into two parts, with the CPU processing the first part (i.e., the first T_c rows) and the GPU

```
main() { //host code
        InitializeMatrix(M); // N = M.size;
        launchKernel(N); // create N work-items
}
kernel(M) { //device code
        rowID = globalID;
        processRow(M, rowID);
}
```

(a) Original Program

```
main() { //host code
        initializeMatrix(M);
        Tc = getCoraseGrainedPartitioningThreshold(M);
        launchCPUKernel(Tc);
        launchGPUKernel(N-Tc);
}
kernelCPU(M) { // CPU kernel
        rowID = globalID;
        processRow(M, rowID);
}
kernelGPU(M) { // GPU kernel
        rowID = globalID + Tc;
        processRow(M, rowID);
}
```

(b) Coarse-Grained Partitioning

```
main() { //host code
        initializeMatrix(M);
        Tf = getFineGrainedPartitioningThreshold(M);
        for( row in matrix) {
                if (row.length >= Tf)
                        cpuRowMap.append(row.ID);
                else // (row.length < Tf)
                        gpuRowMap.append(row.ID); }
        launchCPUKernel(cpuRowMap.size);
        launchGPUKernel(gpuRowMap.size);
}
kernelCPU(M) { // CPU kernel
        rowID = cpuRowMap[globalID];
        processRow(M, rowID);
}
kernelGPU(M) { // GPU kernel
        rowID = gpuRowMap[globalID];
        processRow(M, rowID);
}
```

(c) Fine-Grained Partitioning

Fig. 8. Code transformation for coarse-grained and fine-grained partitioning.

processing the second part (i.e., the last $N - T_c$ rows). The kernel function for the CPU is the same as that in the original program, but its launch should only create T_c work-items. The GPU kernel is different from the original kernel because it should start the processing from the T_cth row with $N - T_c$ work-items. In the case of coarse-grained partitioning in Figure 5, the CPU processes the first 4 rows, and hence the value for T_c is 4. By adding it to the global ID of all work-items of the GPU kernel, the GPU will work on the last four rows.

Figure 8 (c) shows the transformed code for fine-grained partitioning. The host counts for each row the number of non-zero elements. If the number is larger than the threshold T_f returned by $getFineGrainedPartitioningThreshold$, the row is appended to the queue $cpuRowMap$, indicating its processing on the CPU. Otherwise, the row should be processed by the GPU. In the kernel functions, the work-items running on the CPU and GPU figure out the rows to work on through the row IDs recorded in $cpuRowMap$ and $gpuRowMap$, respectively. Similar as

in the transformation for coarse-grained partitioning, the number of work-items to create for each kernel launch depends on the number of rows it processes. The function $getFineGrainedPartitioningThreshold$ needs sophisticated performance models and the input features to determine T_f for both load balance and optimized GPU utilization. We delay its discussion in the next two subsections. For the example shown in Figure 5, the optimal value for T_f should be 4. Hence, the values in $gpuRowMap$ are $\{1, 2, 3, 5, 6, 7\}$, and the values in $cpuRowMap$ are $\{0, 4\}$.

3.3 Performance Modeling

FinePar uses linear regression to build performance models because they are lightweight and efficient for online use. Moreover, the performance models should be automatically generated and general enough to cover various inputs and irregular applications. Since the input graphs can be represented by adjacency matrices, we use non-zero elements processed per second as the prediction goal in the performance models. We build a separate performance model for the CPU and GPU, respectively, due to their different architectures.

Accurate performance models for irregular applications are notoriously difficult to build. Particularly in this work, we address two challenges. First, we need to select several features that are easy to obtain and have great impact on performance. Second, the model should be lightweight for online use. We next describe how the performance modeling component addresses these challenges.

Feature Selection We select features that are closely related with the OpenCL programming model and those that represent irregularity of the workload. More specifically, we select four features: 1) the average workload for a work-item (AW), 2) the variance of the distribution of non-zero elements across the rows (VW), 3) the number of work-items in the computation domain (NW), and 4) the size of the whole workload (SW). Please note that these features belong to metadata. We obtain the metadata with a preprocessing process in an offline manner, since each dataset only requires calibrating for once. Thus, at the runtime we do not need to scan the entire dataset to obtain the values. All four features greatly influence the performance explained as follows:

- **Average workload for a single work-item**: Work-items need enough workload to amortize the overhead of thread creation. We use the *mean* of the numbers of non-zero elements in the rows to represent the average workload for a single work-item because the input program uses one work-item to process a row.
- **Variance of the distribution of non-zero elements**: As explained in Section 2, the irregularity of the workload may dramatically influence the performance of the GPU. We use the variance of the distribution of non-zero elements to quantify the irregularity of the workload.
- **Number of work-items in the computation domain**: This feature plays an important role in the performance of the GPU, because the GPU needs to create enough threads to utilize the computational resource. Due to the one-to-one mapping between the work-items and rows, this feature is the same as the number of rows in the workload.
- **Size of the whole workload**: The amount of data fed to the processor affects performance because large data size may lead to better utilization of the memory bandwidth.

Addressing Substantial Differences among Matrices One tricky feature of graph and sparse matrix applications is their irregular memory access pattern, which affects cache performance and the main memory bandwidth utilization. However, the memory access pattern is not captured by the linear regression model. To illustrate its impact, we run SpMV on two matrices ($M1$ and $M2$) of similar features as selected for the modeling. The difference between these two matrices is that $M1$ is a quasi-diagonal matrix (i.e., its non-zero elements are close to the diagonal), while $M2$ is not. We observe that on both devices, the processing of $M1$ can be 2X faster than that of $M2$.

Despite its importance, the memory access pattern depends on the distribution of the non-zero elements and the interleaved execution of the threads, which is expensive to profile and hard to model. Hence, to circumvent this problem, we categorize the training matrices into quasi-diagonal matrices and non-quasi-diagonal ones, which are referred as Type 1 and Type 2 matrices, respectively, in the remainder of the paper. We build different performance models for each type. Note that we can create more categories to further differentiate the matrices, but leave that to future work.

We quantify the closeness of the non-zero elements to the diagonal in the following way. For each row, we say a non-zero element is close to the diagonal if its column ID within one eighth of the width of the matrix away from the diagonal. We calculate the ratio of the number of such non-zero elements to the total number of non-zero elements. If the result is larger than the threshold T_{diag} (0.8 in our experiments), we categorize the matrix as a Type 1 matrix. Otherwise, we categorize it as a Type 2 matrix.

Building and Training Lightweight Linear Regression Models For each type of matrices, we build a linear regression model for the CPU and one for the GPU. Given a training matrix or graph, we choose a value for T_f (the partitioning threshold) from $\{16, 32, 64, 128, 256, 512, 1024, 2048\}$ and partition the matrix into CPU and GPU workloads as described in the fine-grained partitioning approach in Section 3.2. We then run the partitioned workloads on the CPU and GPU to collect execution times for the training, which capture performance degradation due to co-running. Moreover, we choose to use $log(NW)$ instead of NW in the model because GPU can only simultaneously run up to a certain number of threads. Further increasing the number of threads does not improve performance. Similarly, we use $log(SW)$ instead of SW because of the memory bandwidth limit of the shared physical memory. Equation 1 and Equation 2 show the performance models for the GPU and CPU, respectively. The C_i's ($i = 1 \cdots 5$) are the parameters of the model we need to train.

To quickly generate training data with various patterns, we use the graph generator from Graph 500 [24] to generate all the training data. The generator has five parameters: S, A, B, C, and D. The scale parameter S controls the size

$$performance_{GPU} = C1_{GPU} \times AW_{GPU} + C2_{GPU} \times VW_{GPU} + C3_{GPU} \times log(NW_{GPU}) + C4_{GPU} \times log(SW_{GPU}) + C5_{GPU} \quad (1)$$

$$performance_{CPU} = C1_{CPU} \times AW_{CPU} + C2_{CPU} \times VW_{CPU} + C3_{CPU} \times log(NW_{CPU}) + C4_{CPU} \times log(SW_{CPU}) + C5_{CPU} \quad (2)$$

of the generated graph, which has 2^S vertices and $2^{(S+4)}$ edges. The other four parameters control the distribution of non-zero elements in the adjacency matrix that represents the generated graph. We refer the readers to [25] for the detailed meaning of these parameters, but note that the sum of the four parameters should be 1. We set S to be each of $\{16, 17, 18, 19\}$. For each scale parameter S, we randomly generate 20 quadruplets. Each quadruplet contains four positive floating-point numbers whose sum is 1. The largest number is assigned to A, and the other three are randomly assigned to B, C, and D. We hence generate 80 matrices of Type 2. We generate Type 1 matrices by placing the non-zero elements in each row of Type 2 matrices around the diagonal. Note that because T_f has eight possible values, the training process needs 1280 runs in total.

3.4 Online Tuning

Given the input data, the goal of online tuning is to select the threshold (T_f in Figure 8) for fine-grained partitioning to achieve the best performance. It consists of two stages: (1) matrix category detection, and (2) threshold search. The detection stage determines the matrix category and subsequently the performance models to use. The search stage leverages the performance models to predict performance given a threshold and search for the optimal threshold.

While we can use the method discussed in Section 3.3 to determine the category the input belongs to, the overhead is prohibitive. To be suitable for online use, FinePar samples a number of rows from the input matrix and only counts the non-zero elements close to the diagonal for the sampled rows. For the quantification to determine the category, we scale down the total number of non-zero elements according to the sampling ratio. We tried multiple sampling ratios and found that the sampling ratio 0.001 introduces acceptable overhead and always categorizes the input matrix as the offline training phase does.

Threshold search uses the hill climbing algorithm [26] to search for the optimal threshold. FinePar first chooses an initial value for T_f such that the ratio between the numbers of non-zero elements in the two partitioned workloads matches the ratio of the peak performance between the CPU and GPU. It then uses the performance model to estimate the execution time given T_f, $(T_f - step)$, and $(T_f + step)$ as the threshold, respectively. If T_f produces the optimal performance, the tuning process terminates. Otherwise, T_f is assigned one of the two other values, which yields better performance, and the search process continues. We empirically choose 64 for the $step$ parameter, which performs well in the experiments.

4 Optimizations

In this section, we further provide three optimization strategies to improve the performance of FinePar. We provide the detailed optimizations below.

4.1 More Fine-Grained Partitioning Method

Although FinePar removes substantial irregularity through fine-grained partitioning between the CPU and GPU, the workload in each device may still contain certain irregularity for some applications, as shown in Figure 9 (a). To further improve the performance for each device, we propose a more fine-grained partitioning method to map the data into different groups, and each group is processed by a computation kernel. Compared to the original FinePar [23] in which each device only launches one kernel to process the workload after partitioning, the optimized version allows each device to launch multiple kernels to further reduce irregularity.

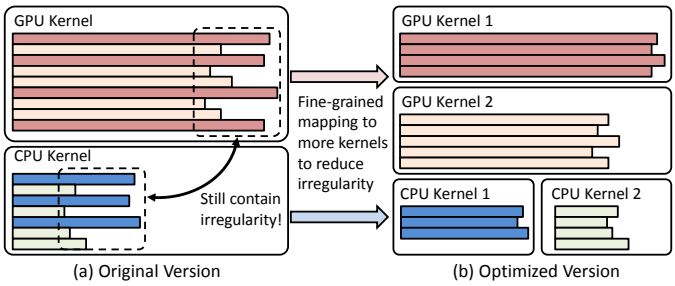


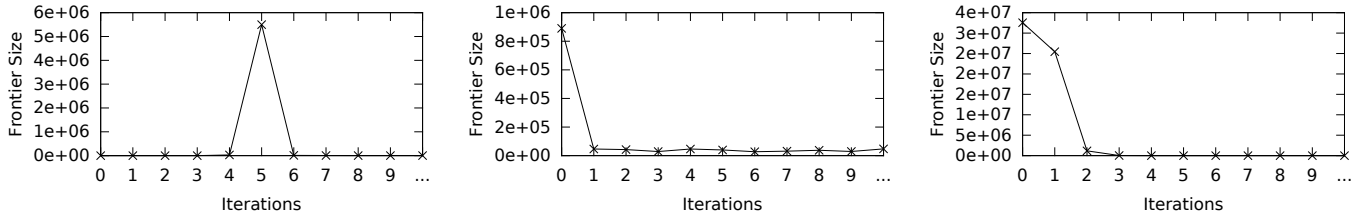
Fig. 9. A more fine-grained partitioning method. The length of each bar represents the number of non-zero elements in each row.

We show the CPU and GPU kernels in FinePar after this mapping in Figure 9 (b). For each device, we map the rows that have the similar number of non-zero elements into one group, and launch a separate computation kernel for this group. With this method, we can further reduce irregularity for each device. By default, FinePar uses the power of 4 to determine the groups; $group_i$ contains the rows with the number of non-zero elements greater than or equal to $\lfloor 4^{i-1} \rfloor$, and less than 4^i (i is a non-negative integer). Experimental results show that this method achieves significant performance benefits in the situation where the dataset size and variance of non-zero elements in each row are large. The reason is that the benefit of reducing irregularity outweighs the cost of launching multiple kernels.

4.2 Dynamic Workload

So far, FinePar only considers the situation where the workload does not change during the execution. However, for some applications such as BFS (Breadth First Search), the data processed in each iteration are changed dynamically. For a given graph data in dynamic workloads, the sizes of nodes or edges needed to be processed for each iteration (active elements in frontier) vary greatly. A static partitioning strategy may achieve a sub-optimal result for these dynamic workloads.

We use three dynamic applications, BFS, connected components, and graph coloring, on dataset *circuit5M* as an example to describe this phenomenon, shown in Figure 10. We find that the frontier size (number of active elements) can be very high in a limited number of iterations, especially for some long-tailed distribution graphs. In the remaining iterations, the frontier size remains low. To address this problem, we adjust the partitioning strategy to only use the CPU for processing these iterations, because the elements to



(a) BFS (b) Graph Coloring (c) Connected Component

Fig. 10. Frontier size for different iterations in *circuit5M*.

be processed in these iterations are not enough to utilize the high parallelism of the GPU.

We further propose an adaptive partitioning in FinePar targeting dynamic workload. FinePar maintains the size of the frontier at runtime, and leverages both the CPU and GPU to process an iteration only when the size of the frontier exceeds a predefined threshold (20% of the total elements). Moreover, after processing the majority of the non-zero elements (e.g., 80%), we do not count the number any more to reduce the runtime overhead. We take BFS as an example. In BFS, the active elements are vertices in a frontier list. We can count the number of frontier list for each iteration and adjust our partitioning strategy accordingly. In our implementation, we use the CPU to calculate the frontier size for the next iteration. Moreover, the majority of the total elements are usually processed in the first few iterations, and then the procedure of calculating frontier size is not needed; thus, compared to the whole program execution, the overhead incurred by the calculation of frontier size in a limited number of iterations can be ignored.

4.3 Large Datasets

One major advantage of the integrated architecture is that it provides much larger memory capacity than the discrete GPU architecture. Therefore, we can leverage the integrated architecture to process large datasets. However, due to the limitation of the creation size of OpenCL memory object, we cannot directly process a large dataset on such an architecture. In FinePar, we partition a large dataset into several medium-sized parts and create a separate OpenCL memory object for each part. Moreover, for a large dataset exceeding the total memory, these medium-sized objects need to be processed sequentially, to avoid memory overflow by releasing the resource of processed objects. Specifically, we propose a pipeline-based method to maximize the performance of processing large datasets, as shown in Figure 11. The processing of each part consists of two stages: IO stage and computation stage, which can be pipelined. After each iteration, FinePar collects the intermediate result and reuses the allocated memory space. With this method, FinePar can effectively hide processing latency and take full advantage of the integrated architecture.

4.4 Discussion

Building More Sophisticated Models Although Linear Regression model is simple and efficient in our experiment, a natural question is whether the linear model is good enough and whether more sophisticated learning models can further improve performance. As a sanity check, we also built a Multi-Layer Perceptron (MLP) model [27]. The MLP model is a supervised machine learning model that is

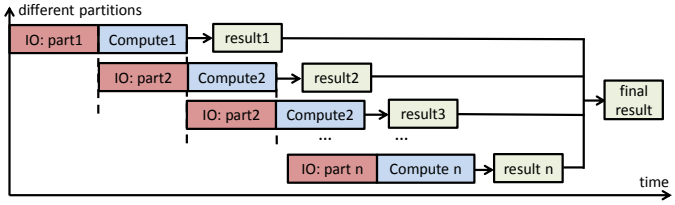


Fig. 11. Pipelined processing for large datasets.

theoretically more powerful than linear regression. Similar to the linear regression model, the MLP model takes as inputs the four features AW, VW, $log(NW)$, $log(SW)$ and predicts the performance in terms of the number of non-zero elements processed per second. We use the same training set as used for the linear regression model and build separate performance models for the CPU and GPU. We show the results in Section 5.4.

Application Scope In general, FinePar is designed for irregular applications, in which the inputs can be represented in the CSR format or adjacency lists with one level of row-pointers. However, the insight in FinePar can be extended to other irregular data types, such as MPI derived datatypes [28], [29], [30]. In a CPU-GPU distributed environment, to fully release the system's power, the idea of FinePar still applies, but challenges such as data communication need to be considered as future work. For well balanced workload, in case that CPUs and GPUs might not deliver the same throughput for each iteration, a dynamic adjustment approach [31] could help. Additionally, FinePar uses a generated dataset to train the model; if users target a higher accuracy or a more complex situation [32], real input data can be added into the training set, which will be evaluated in our future work.

5 Experiment

In this section, we evaluate FinePar using a variety of irregular programs with different types of input matrices. We start by describing our platform and benchmarks.

5.1 Experiment Setup

Platforms We measure the performance of FinePar on two platforms, one with AMD's A-Series APU A10-7850K (code named "Kaveri") [33], which has four cores with four processing threads, and the other with AMD's latest integrated architecture, Ryzen 5 2400G with Radeon RX Vega 11 GPU. We use GCC (version 4.8.2) with O3 optimization level for compilation on A10-7850K. For Ryzen 5 2400G, currently, AMD only provides Windows 10 64-bit driver, so we perform all experiments on it using Visual Studio for compilation.

Benchmarks We select five programs from the GraphBIG benchmark suite [34] , the Rodinia benchmark suite [35], and the SHOC benchmark suite [36]. Breath-First Search (BFS) is from the Rodinia benchmark suite. Connected Component (CC), and Graph Coloring (GC) are from the GraphBIG benchmark suite. Sparse Matrix-Vector Multiplication using Compressed Row Format (SpMV-CSR) and Sparse Matrix-Vector Multiplication using Ellpack Format (SpMV-ELL) are from the SHOC benchmark suite. We also implement three well-known algorithms in OpenCL, Page Rank [37], Hyperlink-Induced Topic Search (HITS) [38], and Random Walk with Restart (RWR) [39], which brings the total number of evaluated benchmarks to eight.

Input Matrices We evaluate FinePar using eight sparse matrices listed in Table 1, which are different from the training set. Specifically, the matrices of *scale20* and *scale21* are generated by the generator of Graph 500 [24]. We use four sparse matrices, *circuit5M*, *eu-2005*, *FullChip*, and *web-BerkStan* from the University of Florida Sparse Matrix Collections [40]. These sparse matrices are widely used in previous studies, such as [41], [42]. Since the ELL format introduces significant space overhead, our platform can only execute SpMV-ELL on *web-BerkStan*. Moreover, we use two large matrices of *uk-2002* and *indochina-2004* to validate the performance for processing large datasets.

TABLE 1
Matrices used in our experiments. Dimension: the dimensions of matrices. NNZ: the number of total non-zero elements. μ: the average number of non-zero elements per row. σ: variance of the number of non-zero elements per row. MAX: the maximum number of non-zero elements per row.

Name	Dimension	NNZ	μ	σ	MAX
scale20	1.05M	31.35M	29.90	258.04	66546
circuit5M	5.56M	59.52M	10.71	1356.62	1290501
eu-2005	0.86M	19.24M	22.30	29.33	6985
scale21	2.10M	63.42M	30.24	300.38	106906
FullChip	2.99M	26.62M	8.91	1806.80	2312481
web-BerkStan	0.69M	7.60M	11.09	16.36	249
uk-2002	18.52M	298.11M	16.10	27.53	2450
indochina-2004	7.41M	194.11M	26.18	215.83	6985

5.2 Performance of FinePar

We compare our method with four state-of-the-art workload partitioning methods on heterogeneous platforms listed in Table 2. The single-device method [17] uses the device from the CPU and GPU that produces better performance. The adaptive method [16] calculates a performance ratio between CPU and GPU through executing partial workload and then partitions the workload using this ratio. The dynamic method [15] uses both GPU and CPU to execute the workload simultaneously, while the GPU executes the workload from the beginning to the end and the CPU executes in the opposite direction, which can achieve a dynamic load balance. Because it is implemented in Pthreads, we only evaluate it on the A10-7850K platform. The coarse-grained oracle method [14] performs workload partitioning from 0 to 100% and selects the best partitioning ratio. We also list the original FinePar [23].

Figure 12 and 13 show the performance results for different partitioning methods. We use single-device as the baseline. Speedup is defined as the baseline's execution time divided by the corresponding method's execution time. In general, FinePar achieves consistent performance improvement for most of the evaluated programs and is much better than the other partitioning methods. The average performance speedup of FinePar is 1.67X over the single-device method, and 1.08X over the original FinePar version. For the *FullChip* matrix, the performance speedup is up to 2.40X on A10-7850K platform. For the coarse-grained oracle method, the average speedup is 1.07X. FinePar achieves an average of 1.6X speedup over the coarse-grained oracle method.

TABLE 2
Summary of different partitioning methods.

Method	Descriptions
Single-Device [17]	Choose CPU or GPU that yields the best performance
Adaptive [16]	Partition workload based on online profiling
Dynamic [15]	Both GPU and CPU execute the workload from opposite directions
Coarse-Grained Oracle [14]	Coarse-grained workload partitioning with optimal load balance
FinePar (Original) [23]	The original fine-grained workload partitioning method
FinePar	FinePar with all optimizations including the more fine-grained partitioning, dynamic adaptation, and pipelining in Section 4

From Figure 12 and 13, we can see that FinePar achieves performance improvement over the optimal single device method in most cases. The adaptive method calculates a partitioning ratio with a light-weight sampling method, but this ratio sometimes cannot reflect the most balanced partitioning point for some inputs, such as *scale20* and *circuit5M*. The coarse-grained oracle method presents the upper limit of the adaptive results. However, for most irregular inputs, it only brings very little performance improvement over the single device method. For the dynamic method, it can achieve good load balance for most programs, but it incurs large runtime overhead for checking whether CPU and GPU execute to the same point.

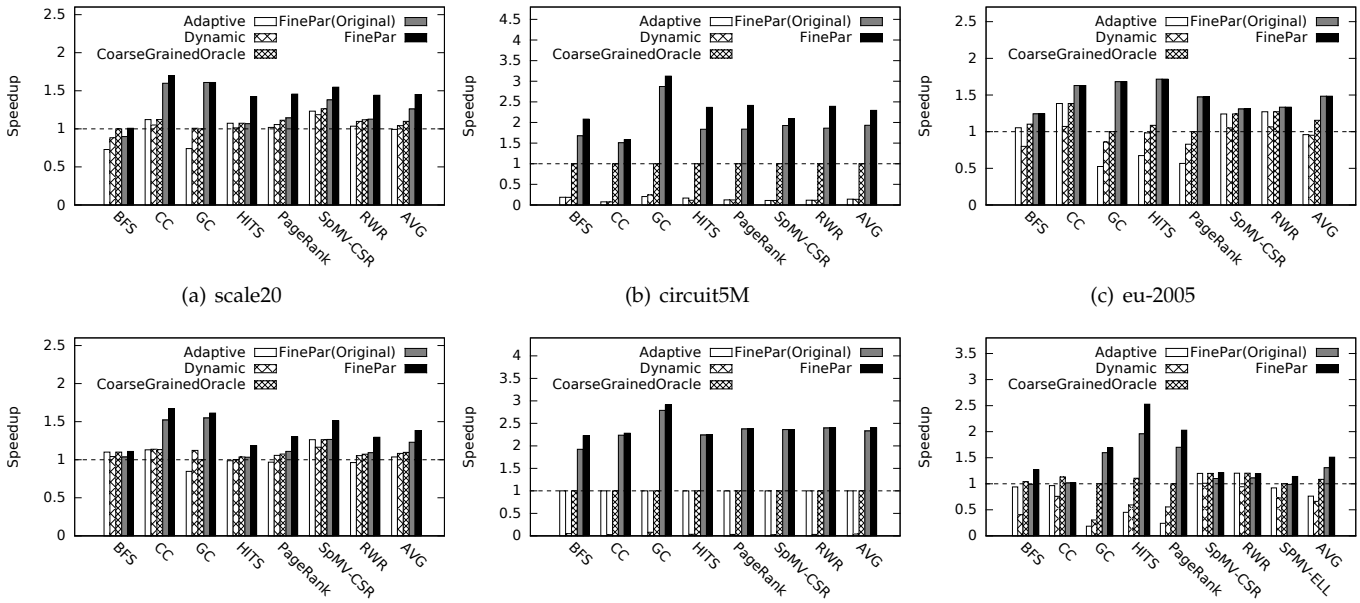
We show the performance of processing large datasets in Figure 14. FinePar uses the partitioning technique to process these datasets, which is explained in Section 4.3. The average speedup is 1.3X, and the performance on both large datasets is similar. Compared to the baseline of using a single device, FinePar still presents clear performance benefits. Compared to the performance on medium-sized datasets, FinePar produces moderate performance speedup because of unbalanced workload partitioning, especially for dynamic workloads with long-tailed distribution dataset.

5.3 Result Analysis

In general, our method partitions an irregular workload into two parts, the relatively regular part allocated to the GPU and the more irregular part allocated to the CPU. By considering the architectural differences between two devices, we can effectively improve the performance of irregular programs. In this section, we give detailed analysis about our fine-grained partitioning.

5.3.1 Analysis for Transformation

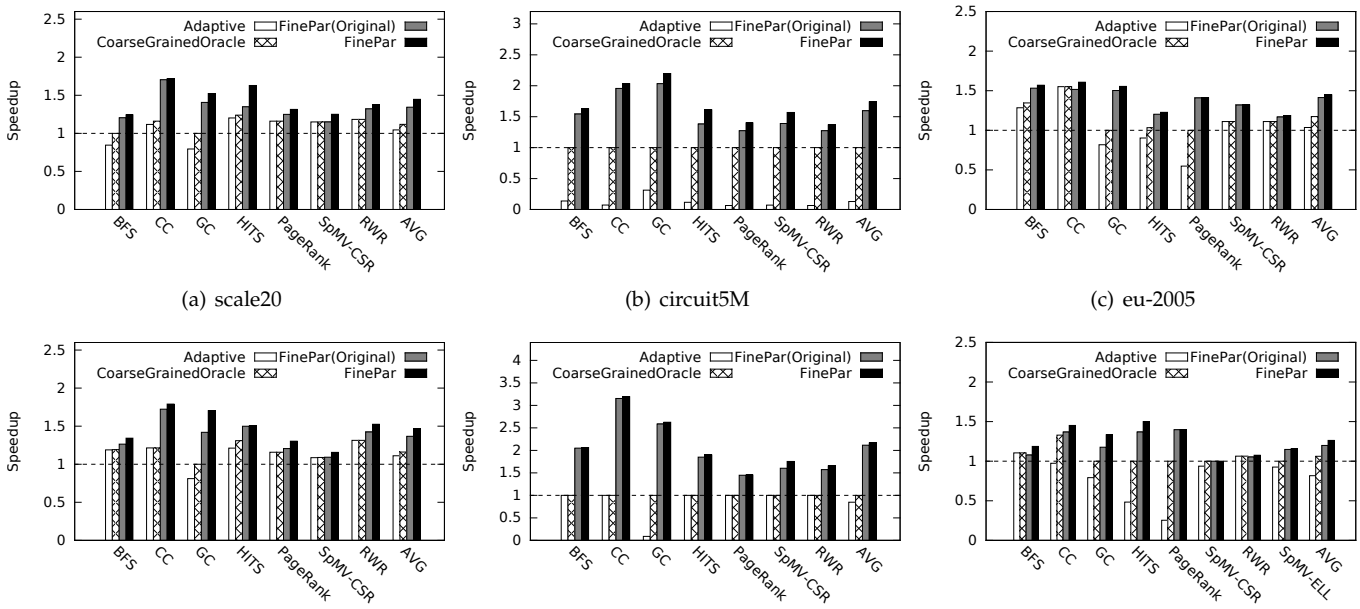
Our method largely benefits from mitigating the irregularity of the GPU workload. Table 3 shows the changes after performing fine-grained partitioning in FinePar for different matrices. We use the variance of the number of non-zero elements per row to describe the matrix irregularity. The



(a) scale20 (b) circuit5M (c) eu-2005

(d) scale21 (e) FullChip (f) web-BerkStan

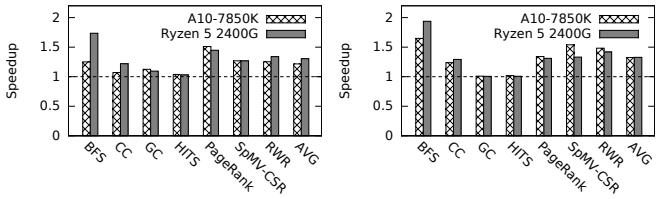
Fig. 12. Performance results of different partitioning methods on A10-7850K. The baseline is the optimal single device result, GPU- or CPU- only version.



(a) scale20 (b) circuit5M (c) eu-2005

(d) scale21 (e) FullChip (f) web-BerkStan

Fig. 13. Performance results of different partitioning methodson on Ryzen 5 2400G. The baseline is the optimal single device result, GPU- or CPU-only version.



(a) uk-2002 (b) indochina-2004

Fig. 14. Performance results for large datasets. The baseline is the optimal single device result, GPU- or CPU- only version.

Matrix Variance column represents the variance of the original matrix before partitioning. The *GPU Variance* and *CPU Variance* columns represent the variances for the GPU workload and the CPU workload, respectively, after partitioning. The *GPU/CPU Workload Ratio* column shows the size of the GPU workload divided by the size of the CPU workload.

TABLE 3
Mitigating the irregularity of the input matrices by FinePar.

Name	Matrix Variance	GPU Variance	CPU Variance	GPU/CPU Workload Ratio
scale20	258.04	56.57	2496.13	1.36
circuit5M	1356.62	0.50	5416.37	0.78
eu-2005	29.33	15.38	71.18	2.92
scale21	300.38	35.41	2230.48	0.71
FullChip	1806.80	2.74	26307.03	3.80
web-BerkStan	16.36	6.73	29.49	1.64

After transformation by FinePar, the variance for the GPU workload is significantly reduced, while the variance

for the CPU workload is increased. For instance, for the matrices of *circuit5M* and *FullChip*, the original matrices have very high irregularity with variances of 1356.62 and 1806.80, respectively, but after fine-grained partitioning, the irregularity of GPU workloads has been significantly reduced (with the variances of 0.50 and 2.74). In contrast, the traditional coarse-grained partitioning does not realize such input irregularity and only considers the load balance. Moreover, the *GPU/CPU Workload Ratio* column shows that the workload partitioning ratios vary greatly across inputs.

From the aspect of matrix variance, we classify the performance results in Figure 12 into three categories. First, the matrices of *circuit5M* and *FullChip* have the largest irregularity and their irregularity can be significantly decreased after fine-grained partitioning. Our method can get very high performance improvements for these inputs. Second, the matrices of *eu-2005* and *web-BerkStan* have the moderate irregularity and there is no significant irregularity difference between CPU and GPU after the fined-grained partitioning. However, our method can still produce moderate performance improvement for these inputs. Third, for *scale20* and *scale21*, their irregularity is uniformly distributed in the whole matrix, so it is difficult to greatly reduce their irregularity. For example, Table 3 shows that the GPU workload of *scale20* still has a variance of 56.57 after fine-grained partitioning. Consequently, the performance improvement is limited for this matrix.

5.3.2 Performance Profiling

We also use performance counters in GPU to analyze the micro-architecture-level performance behaviors of the different partitioning approaches. Figure 15 shows the improvement on GPU utilization over the GPU-only approach as the baseline. The improvement is defined as the utilization of the compared approach divided by that of the GPU-only approach. Each bar represents the average improvement of all inputs for the corresponding benchmark. The bar height of one means there is no improvement or degradation. All the performance data is collected by AMD CodeXL. CodeXL crashes when collecting the performance data for BFS-Dynamic and HITS, which is hence removed from the graph. For the coarse-grained oracle approach, all the workload of GC and BFS is dispatched to the CPU, so there is no data on the GPU side. We observe that all the three coarse-grained approaches have similar GPU utilization as the GPU-only approach does. The reason is that those approaches only concern the load balance between the CPU and GPU, but do not change the load balance across GPU threads in the same work group. FinePar substantially improves GPU utilization over the other approaches. The average improvement over coarse-grained oracle is 10.1X. Additionally, FinePar achieves 4.4X speedup over the original version, which implies that the optimizations in Section 4 are effective.

Due to space limitation, we only show the results for *scale20*, but the other input matrices have similar trends. Since our fine-grained partitioning method can significantly mitigate the input irregularity for the GPU workload, load balance for GPU threads is improved, which leads to better GPU core utilization. Moreover, it also enhances memory bandwidth utilization because more active GPU threads can issue memory requests together.

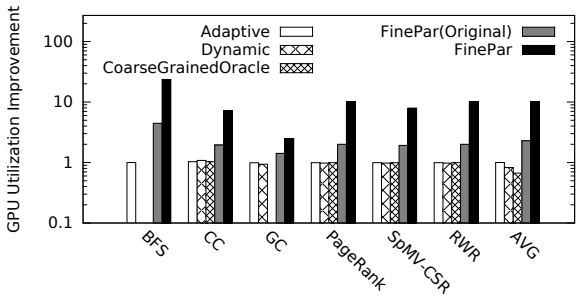


Fig. 15. Improvement on GPU core utilization (scale20).

5.4 Accuracy of Performance Models

Table 4 shows the trained parameters of the performance models from our offline training. We use a statistical metric, called coefficient of determination [43], to analyze the accuracy the predicted performance model. The values of *r2* range from 0 to 1. The larger this value is, the better the predicted result is. For the Type 1 matrices, the values of *r2* are close to 1, which means that our performance model has very high accuracy. For the Type 2 matrices, the values of *r2* are not very large, because the Type 2 matrices have a great diversity of sparsity patterns and it is hard to provide an accurate performance model for prediction.

TABLE 4
Estimated parameters of performance models. Type 1: non-zero elements around the diagonal. Type 2: non-zero elements not around the diagonal. *r2* is the coefficient of determination.

Type	Device	C1	C2	C3	C4	C5	r2
Type 1	CPU	-0.05	0.03	-283.83	605.24	-2165.00	0.81
	GPU	19.03	-5.11	2752.44	244.77	-16924.91	0.93
Type 2	CPU	-0.05	0.07	14.42	13.26	122.77	0.50
	GPU	-2.24	0.88	413.11	79.54	-2661.51	0.69

To understand the importance of the features in the models for different devices and types of matrices, we provide their correlation coefficients in Table 5. We list main findings below. (1) The average workload for a work-item (AW) is critical for the GPU, because the GPU has a large number of hardware threads which are more sensitive to the average workload for a work-item. (2) The variance of workload (VW) is also much more important for the GPU. This is because the GPU uses the lockstep execution model as mentioned in Section 2. (3) The number of work-items (NW) is more important for the GPU, because work-items are mapped to hardware threads and the GPU performance is more dependent on available parallelism. (4) The workload size (SW) is much more important for the CPU, because the CPU has very few hardware threads compared with the GPU and its performance is more dependent on the input workload size.

TABLE 5
Correlation coefficient of the features in the performance model.

Type	Device	AW	VW	NW	SW
Type 1	GPU	0.88	0.52	0.85	0.15
	CPU	0.08	0.18	0.77	0.49
Type 2	GPU	0.64	0.75	0.75	0.28
	CPU	0.33	0.03	0.50	0.45

To demonstrate the effectiveness of our performance model, we enumerate all possible fine-grained partitioning thresholds and obtain the maximum performance improvement for each input matrix across all benchmarks. Figure 16 shows the comparison between the improvement by FinePar

and the optimal performance improvement FinePar can achieve (named *Oracle*). For most of the input matrices, FinePar demonstrates very high consistency with the optimal performance. The optimal performance improvement for the proposed fine-grained partitioning approach is 29% on average while FinePar achieves an average of 27% performance improvement.

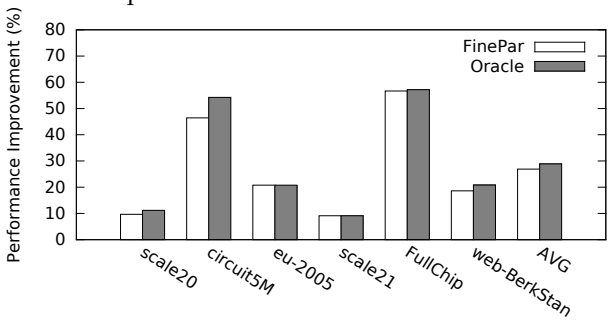


Fig. 16. Performance improvement between FinePar and the optimal partition.

Figure 17 shows the performance comparison between the linear regression model and MLP. Each bar represents the performance speedup when the corresponding model is used for a particular input and program. On average, MLP only provides 2.6% extra performance benefit, but has longer training time. We hence use the linear model as the default model in FinePar, but users can also choose to use MLP if the longer training time is not a concern. Please note that we use the MLP in Scikit-learn [44] in our implementation. We use the set of tuned parameters with the best performance in training set.

5.5 Performance Overhead Analysis

5.5.1 Time Overhead

Before kernel computation, the evaluated programs perform I/O operations and data initialization. Our fine-grained partitioning method adds runtime overhead to this pre-kernel processing phase from two aspects. First, it randomly samples a number of rows from the input matrix to estimate its type. Second, it chooses a suitable performance model and searches for the partitioning threshold.

Table 6 shows the runtime overhead of the original FinePar and the optimized version compared to the preprocessing time of single device version for each program for the matrix *web-BerkStan*. The other matrices have similar performance overhead. We observe that the overhead of the original FinePar only accounts for less than 0.2% of the preprocessing time of single device version, while the optimized version has less than 3% occupancy overhead, which is negligible.

TABLE 6
The time overhead of the fine-grained partitioning method.

	I/O(%)		Initialization(%)		Occupancy(%)	
Program	FinePar (original)	FinePar	FinePar (original)	FinePar	FinePar (original)	FinePar
BFS	74.67	74.66	25.25	25.16	0.08	0.18
ConnectedComp	31.27	31.5	68.70	68.41	0.03	0.09
GraphColoring	58.91	58.79	41.05	41.07	0.04	0.14
HITS	52.36	51.04	47.51	47.04	0.13	1.92
PageRank	55.22	54.58	44.73	44.24	0.05	1.18
SpMV-CSR	75.00	73.37	24.90	26.44	0.10	0.19
RWR	71.78	70.18	28.08	27.81	0.14	2.01
SpMV-ELL	15.53	15.25	84.44	84.62	0.03	0.13

5.5.2 Space Overhead

The APU has two separate main memory data paths to the CPU and GPU. To accurately evaluate and compare different partitioning approaches, we allocate two copies for read-only data for the two data paths to reduce the interference due to bus contention. For a graph with n vertices and m edges. The needed storage space for read-only data is:

$$Size_{total} = (m + n) \times sizeof(int) \times 2 \quad (3)$$

The original FinePar creates a bit vector of n bits, named *gpuRowMap*, to inform the GPU the rows it should process. The bit vector reduces storage space and improves memory coalescing for the GPU. Since the GPU typically processes much more rows than the CPU does, it only uses a regular integer array of size n for the CPU. Hence, the space overhead incurred by the original FinePar is:

$$Size_{FinePar(original)} = n/8 + n \times sizeof(int) \quad (4)$$

For the optimized version, FinePar also uses the bit vector. FinePar allows each device to launch multiple kernels, and it creates one vector for each kernel. Therefore, the size of bit vector relates to the number of launched kernels, g. To avoid IO overhead, CPU also reads the bit vector. The space overhead incurred by FinePar is:

$$Size_{FinePar} = n/8 \times g \quad (5)$$

Table 7 shows the extra space overhead introduced by the original FinePar and the optimized version. The column named "Size (MB)" shows the original size for each matrix. The columns in "Extra Allocation (MB)" show the size incurred by both versions. The last two columns show the space overhead of both versions normalized to the original matrix size. Because the optimized version does not use the integer array for CPU, its space overhead is lower than that of the original version. For all the inputs, our method introduces little space overhead (less than 6%).

TABLE 7
The space overhead of the fine-grained partitioning method.

		Extra Allocation(MB)		Space Overhead(%)	
Matrix	Size (MB)	FinePar (original)	FinePar	FinePar (original)	FinePar
scale20	259	4	3	1.54	1.01
circuit5M	521	23	16	4.41	3.07
eu-2005	161	4	2	2.48	1.34
scale21	524	10	6	1.91	1.05
FullChip	237	12	8	5.06	3.47
web-BerkStan	66	3	2	4.55	2.60

6 Related Work

Heterogeneous architectures pose new optimization opportunities for knowledge and data engineering applications, thanks to the high parallelism and throughput. He et al. [45] used GPUs to accelerate SimRank computation. Shi et al. [46] accelerated graph processing on GPUs. Lin et al. [47] applied GPUs to accelerate the identification process of network motifs. There have been other recent studies on using GPU or FPGA to accelerate data processing operations. Serra et al. [48] designed a GPU-based Monte Carlo algorithm that significantly reduces the long running time. Zhou et al. [49] provided a hardware-accelerated solution for hierarchical index-based merge join. As for this work, FinePar shows the possibility of using both the CPU and the GPU to further accelerate these applications on integrated architectures.

Recently, heterogeneous CPU-GPU architectures have been used in optimizing irregular applications and multi-dimensional data processing. Vilches et al. [50] developed

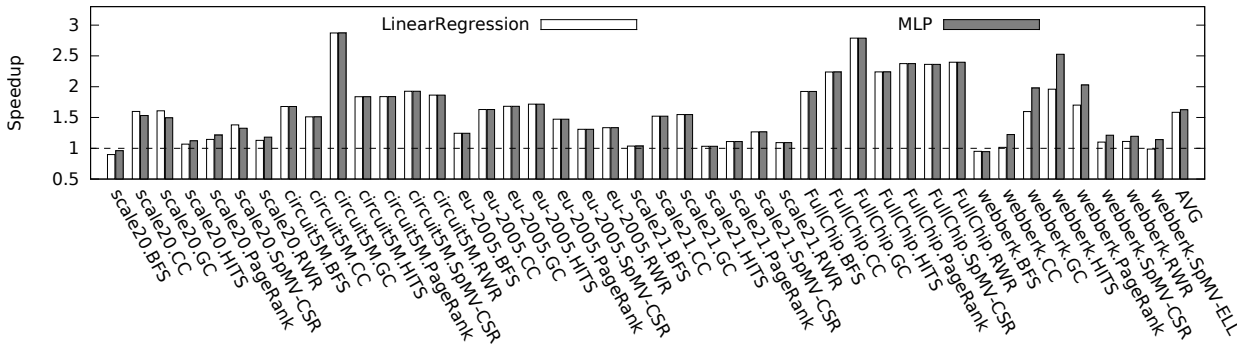


Fig. 17. Comparison between Linear Regression and MLP.

a novel adaptive partitioning algorithm for parallel loops to find the appropriate chunk size for GPUs and CPUs. Navarro et al. [51] also studied the partitioning strategy for parallel loops, specially for irregular applications on heterogeneous CPU-GPU architectures. Sakai et al. [52] proposed a novel decomposition method that can execute single-GPU code on multi-GPU systems. Zhang et al. [53] provided an adaptive BFS algorithm on integrated architectures. In contrast, the core idea of FinePar is to reduce the workload irregularity for GPUs, and the partitioning of FinePar is reflected in the mapping of data to devices: irregular data are assigned to CPU threads while the rest of relatively regular data are left to GPU threads. Thus, we adopt different strategies and the application scenarios are not the same.

7 Conclusion

In this paper, we identified the pitfall of coarse-grained workload partitioning for irregular applications on integrated architectures. We pointed out that even if coarse-grained partitioning achieves ideal load balance between the CPU and GPU, the integrated architecture may still get under-utilized. To deal with the problem, we developed a system software named FinePar to achieve fine-grained partitioning. FinePar considers architectural differences of the CPU and GPU, and builds irregularity-aware performance models for partitioning the workload through auto-tuning. Experimental results of eight applications demonstrated 1.6X performance speedup over the optimal coarse-grained partitioning.

Acknowledgments

This work has been partly supported by the National Key R&D Program of China (Grant No. 2016YFB0200100) and National Natural Science Foundation of China (Grant No. 61732014, 61722208, 61802412); it is also supported in part by NSF grant CCF-1823005 and an NSF CAREER Award (CNS-1750760). Bingsheng's work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore.

References

[1] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*. ACM, 2008, pp. 511–524.

[2] J. Paul, J. He, and B. He, "GPL: A GPU-based pipelined query processing engine," in *SIGMOD*. ACM, 2016, pp. 1935–1950.

[3] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGGRAPH*. ACM, 2005, p. 206.

[4] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM TODS*, vol. 34, no. 4, p. 21, 2009.

[5] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "GPUQP: query co-processing using graphics processors," in *SIGMOD*. ACM, 2007, pp. 1061–1063.

[6] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[7] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs," *PVLDB*, vol. 11, no. 1, pp. 107–120, 2017.

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," in *PPoPP*. New York, NY, USA: ACM, 2016, pp. 11:1–11:12.

[9] J. Zhong and B. He, "Parallel graph processing on graphics processors made easy," *PVLDB*, vol. 6, no. 12, pp. 1270–1273, 2013.

[10] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: implications for graph mining," *PVLDB*, vol. 4, no. 4, pp. 231–242, 2011.

[11] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *TPDS*, vol. 25, no. 6, pp. 1543–1552, 2014.

[12] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *PACT*. IEEE, 2011, pp. 78–88.

[13] "The Compute Architecture of Intel Processor Graphics Gen7.5," https://software.intel.com, 2014.

[14] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated CPU/GPU architectures," *TPDS*, vol. 28, no. 3, pp. 905–918, 2017.

[15] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices," in *CGO*. ACM, 2014, p. 273.

[16] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *PACT*. ACM, 2014, pp. 151–162.

[17] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai, "Efficient mapping of irregular C++ applications to integrated GPUs," in *CGO*. ACM, 2014, p. 33.

[18] M. Daga, M. Nutter, and M. Meswani, "Efficient breadth-first search on a heterogeneous processor," in *Big Data*. IEEE, 2014, pp. 373–382.

[19] M. C. Delorme, T. S. Abdelrahman, and C. Zhao, "Parallel radix sort on the AMD fusion accelerated processing unit," in *ICPP*. IEEE, 2013, pp. 339–348.

[20] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *SC*. IEEE Computer Society Press, 2012, p. 25.

[21] J. He, M. Lu, and B. He, "Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture," *PVLDB*, vol. 6, no. 10, pp. 889–900, 2013.

[22] K. Nilakant and E. Yoneki, "On the Efficacy of APUs for Heterogeneous Graph Computation," in *Proc. 4th Workshop on Systems for Future Multicore Architectures (SFMA), Amsterdam, Netherlands*, 2014, pp. 2–7.

[23] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "FinePar: irregularity-aware fine-grained workload partitioning on integrated architectures," in *CGO*. IEEE Press, 2017, pp. 27–38.

[24] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, 2010.

[25] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[26] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[27] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, and classification," *IEEE Transactions on Neural Networks*, vol. 3, no. 5, pp. 683–697, 1992.

[28] W. Gropp, T. Hoefler, R. Thakur, and J. L. Träff, "Performance expectations and guidelines for MPI derived datatypes," in *European MPI Users' Group Meeting*. Springer, 2011, pp. 150–159.

[29] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 308–316.

[30] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. K. Panda, "HAND: A hybrid approach to accelerate non-contiguous data movement using MPI datatypes on GPU clusters," in *ICPP*. IEEE, 2014, pp. 221–230.

[31] R. Shi, S. Potluri, K. Hamidouche, X. Lu, K. Tomko, and D. K. Panda, "A scalable and portable approach to accelerate hybrid HPL on heterogeneous CPU-GPU clusters," in *International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[32] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," *TPDS*, 2019.

[33] D. Bouvier and B. Sander, "Applying AMDs Kaveri APU for heterogeneous computing," in *Hot Chips: A Symposium on High Performance Chips*, 2014.

[34] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *SC*. ACM, 2015, p. 69.

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. IEEE, 2009, pp. 44–54.

[36] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the web," 1999.

[38] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[39] H. Tong, C. Faloutsos, and J.-Y. Pan, "Random walk with restart: fast solutions and applications," *Knowledge and Information Systems*, vol. 14, no. 3, pp. 327–346, 2008.

[40] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *TOMS*, vol. 38, no. 1, p. 1, 2011.

[41] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *ICS*. ACM, 2012, pp. 353–364.

[42] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *ICS*. ACM, 2015, pp. 339–350.

[43] L. S. Aiken, S. G. West, and S. C. Pitts, "Multiple linear regression," *Handbook of psychology*, 2003.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[45] G. He, C. Li, H. Chen, X. Du, and H. Feng, "Using graphics processors for high performance SimRank computation," *TKDE*, vol. 24, no. 9, pp. 1711–1725, 2012.

[46] X. Shi, X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin, "Frog: Asynchronous graph processing on gpu with hybrid coloring model," *TKDE*, vol. 30, no. 1, pp. 29–42, 2018.

[47] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A GPU approach," *TKDE*, vol. 29, no. 3, pp. 513–528, 2017.

[48] E. Serra and F. Spezzano, "An effective gpu-based approach to probabilistic query confidence computation," *TKDE*, vol. 27, no. 1, pp. 17–31, 2015.

[49] Z. Zhou, C. Yu, S. Nutanong, Y. Cui, C. Fu, and C. J. Xue, "A hardware-accelerated solution for hierarchical index-based merge-join," *TKDE*, vol. 31, no. 1, pp. 91–104, 2019.
heterogeneous CPU-GPU chips," *Procedia Computer Science*, vol. 51, pp. 140–149, 2015.

[50] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, and M. Garzarán, "Adaptive partitioning for irregular applications on

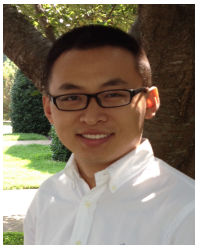
[51] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, and R. Asenjo, "Heterogeneous parallel_for Template for CPU–GPU Chips," *International Journal of Parallel Programming*, vol. 47, no. 2, pp. 213–233, 2019.

[52] R. Sakai, F. Ino, and K. Hagihara, "Towards automating multi-dimensional data decomposition for executing a single-GPU code on a multi-GPU system," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016, pp. 408–414.

[53] F. Zhang, H. Lin, J. Zhai, J. Cheng, D. Xiang, J. Li, Y. Chai, and X. Du, "An adaptive breadth-first search algorithm on integrated architectures," *The Journal of Supercomputing*, vol. 74, no. 11, pp. 6135–6155, 2018.

Feng Zhang received the bachelor degree from Xidian University in 2012, and the PhD degree in computer science from Tsinghua University in 2017. He is an assistant professor in DEKE Lab and School of Information, Renmin University of China. His major research interests include database systems, and parallel and distributed systems.

Jidong Zhai received the BS degree in computer science from University of Electronic Science and Technology of China in 2003, and PhD degree in computer science from Tsinghua University in 2010. He is an associate professor in Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis and modeling of parallel applications.



Bo Wu received the BS degree in information and computational sciences and the MS degree in computer science from Central South University in Changsha, China. He received the PhD degree from the College of William and Mary . He is currently an assistant professor of computer science at Colorado School of Mines in Golden. His research focuses on GPU computing, graph analytics, heterogeneous memory systems, and compiler optimization.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000 respectively. He was the CTO of Opportunity International Inc. from 2000 to 2002. Since January 2003, he joined Tsinghua University. He is a professor and associate head in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing and programming model.

Xiaoyong Du obtained the B.S. degree from Hangzhou University, Zhengjiang, China, in 1983, the M.E. degree from Renmin University of China, Beijing, China, in 1988, and the Ph.D. degree from Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

View publication stats