FinePar: Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures

Feng Zhang[◇], Bo Wu[⋆], Jidong Zhai[◇], Bingsheng He[+], Wenguang Chen[◇]

[◇]Tsinghua University, China

[⋆]Colorado School of Mines, USA

[+]National University of Singapore, Singapore

feng-zhang12@mails.tsinghua.edu.cn, bwu@mines.edu, zhaijidong@tsinghua.edu.cn, hebs@comp.nus.edu.sg, cwg@tsinghua.edu.cn

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. 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 named 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 8 irregular applications on an AMD integrated architecture and compare it with state-of-the-art partitioning approaches. Results show that FinePar demonstrates better resource utilization and achieves an average of 1.38X speedup over the optimal coarse-grained partitioning method.

1. Introduction

In recent years, GPUs have made big strides in throughput-oriented computing thanks to the massively parallel architecture. 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 [2], 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 [5, 9, 10, 13, 16, 18, 28, 30, 39, 40]. 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[5, 13, 18, 30, 39, 40] only perform coarse-grained workload partitioning, without considering the fine-grained collaboration between the CPU and GPU. For example, Delorme et al. [13] and Pandit et al. [30] break the workload into many jobs. Each job typically operates on adjacent data and is processed by a work-group (in OpenCL [33] 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. [18] addressed more complicated applications and dynamically assigned the jobs to processors through lightweight online profiling. Second, although some studies [9, 10, 16, 28] use fine-grained workload partitioning, they are applied to specific applications only such as hash join in databases [16] and MapReduce [9]. They do not necessarily offer an automatic or general solution to irregular applications such as graph processing.

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.

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.

As case studies, we focus on sparse matrix and graph processing applications. We evaluate FinePar with 8 irregular applications on 6 input matrices and compare it with 4 state-of-the-art workload partitioning methods on an AMD A10-7850K APU. Results show that FinePar demonstrates better resource utilization and achieves an average of 1.38X speedup over the optimal coarse-grained partitioning method. Meanwhile, FinePar is very lightweight, only introducing less than 6% space overhead and 0.2% time overhead.

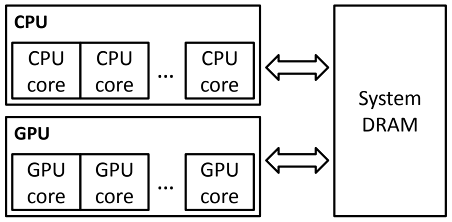


Figure 1. A general view of the integrated architecture.

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 evaluate FinePar on a set of irregular applications and inputs to demonstrate its benefit over state-of-the-art partitioning approaches.

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.

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

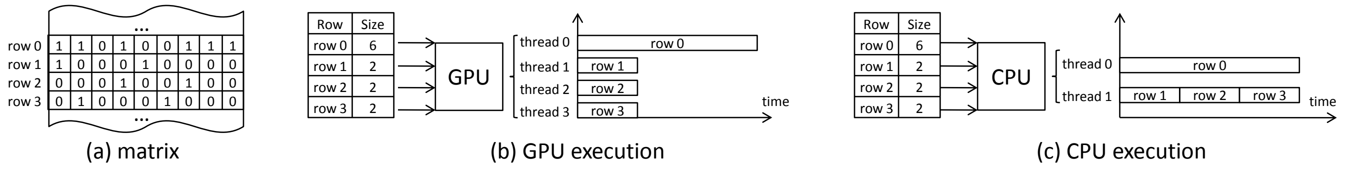
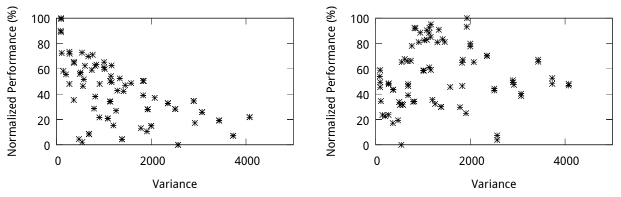


Figure 2. An example to demonstrate the performance features of the CPU and GPU to execute irregular application.



(a) GPU performance trend (b) CPU performance trend

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

of the GPU is typically several times larger than that of the CPU because of the efficiency of SIMD execution.

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 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 lockstep 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.

2.2 Understanding the Inefficiency of Coarse-Grained Workload Partitioning

Previous work [5, 18, 30, 39, 40] 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 4 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 4 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).

Figure 4 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 same time. However, fine-grained partitioning improves the overall performance by 1.5X through better utilization of the GPU resources.

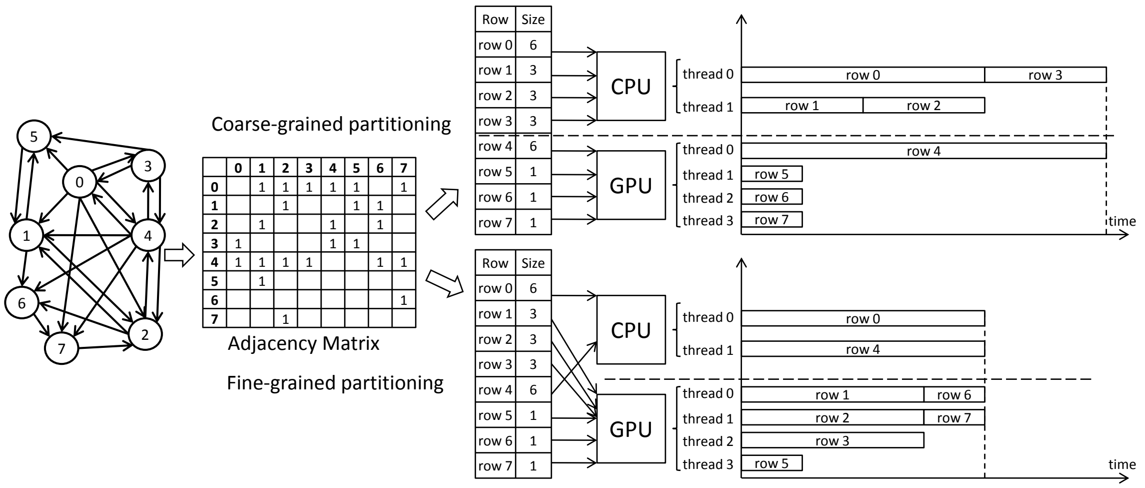


Figure 4. An example to show the benefits from fine-grained partitioning.

3. FinePar Framework

3.1 Overview

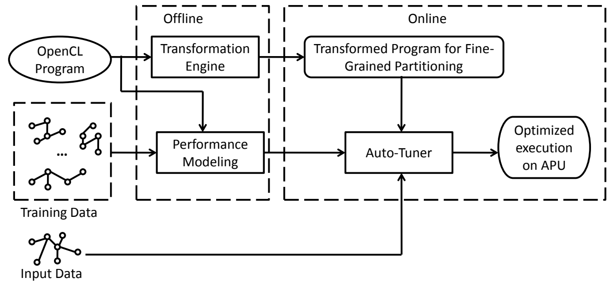


Figure 5. The overview of FinePar.

Figure 5 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.

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 to Enable Fine-Grained Partitioning

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. Figure 6 presents the basic ideas of the transformations using sparse matrix processing as an example. Figure 6 (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 for coarse-grained partitioning, the framework only needs to slightly change the program as shown in Figure 6 (b). FinePar inserts a function $getCoraseGrainedPartitioningThreshold$ (detailed in Section 4.2), 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 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 4, the CPU processes the first 4 rows, so the value for T_c is 4. By adding it to the global ID of all work-items of the GPU kernel, the GPU works on the last four rows.

Figure 6 (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 for each kernel 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. For the example shown in Figure 4, 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.

3.3.1 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), which are explained as follows:

1) **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.

2) **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.

3) **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.

4) **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.

```
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

Figure 6. Code transformation for coarse-grained and fine-grained partitioning.

3.3.2 Addressing Substantial Differences in 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 count the number of non-zero elements whose column is no more than one eighth of the width of the matrix away from the diagonal. We divide the total number of non-zero elements in the matrix by the sum of such numbers for all rows. 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.

3.3.3 Building and Training 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 [26] to generate all the training data. The generator has 5 parameters: S, A, B, C, and D. The scale parameter S controls the size of the generated graph, which has 2^S vertices and $2^{(S+4)}$ edges. The other 4 parameters control the distribution of non-zero elements in the adjacency matrix that represents the generated graph. We refer the readers to [7] for the detailed meaning of these parameters, but note that the sum of the 4 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 8 possible values, the training process needs 1280 runs.

3.4 Online Tuning

Given the input data, the goal of online tuning is to select the threshold (T_f in Figure 6) 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 [31] 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, We empirically choose 64 for the $step$ parameter, which performs well in the experiments.

4. 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.

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

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

4.1 Experiment Setup

Platform We perform all of our experiments on AMD's A-Series APU A10-7850K (code named "Kaveri") [6]. This chip has a CPU with 4 processor cores operating at a frequency of 3.7 GHz, as well as an integrated GPU with 8 computing units. The peak performance of the GPU is 737.28 GFlops/s while the peak performance of the CPU is 118.4 GFlops/s. The system is equipped with 32 GB memory. The operating system is Ubuntu 14.04.1 LTS Linux (kernel version 3.13.0-32-generic). We use GCC (version 4.8.2) with O3 optimization level for compilation.

Benchmarks We select 5 programs from the GraphBIG benchmark suite [27], the Rodinia benchmark suite [8], and the SHOC benchmark suite [11]. 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. Since AMD's current integrated architectures do not support atomic operations between the CPU and GPU, we only choose programs without using atomic operations. We also implement 3 well-known algorithms in OpenCL, Page Rank [29], Hyperlink-Induced Topic Search (HITS) [19], and Random Walk with Restart (RWR) [35], which brings the total number of evaluated benchmarks to 8.

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
in-2004	1.38M	16.92M	12.23	37.23	7753
FullChip	2.99M	26.62M	8.91	1806.80	2312481
web-BerkStan	0.69M	7.60M	11.09	16.36	249

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.

Input Matrices We evaluate FinePar using 6 sparse matrices listed in Table 1, which are different from the training set. Specifically, the matrix of *scale20* is generated by the generator of Graph 500 [26]. We use 3 large sparse matrices from [34], *circuit5M*, *eu-2005*, and *in-2004*. These sparse matrices are widely used in previous studies, such as [4, 15, 25, 36, 38], which are available in the University of Florida Sparse Matrix Collections [12]. We also use a matrix, *FullChip*, from the same collection. Since the ELL format introduces significant space overhead, our platform can only execute SpMV-ELL with a small web graph, *web-BerkStan*, which is available from [12, 22].

4.2 Performance of FinePar

Method	Descriptions
Single-Device [5]	Choose CPU or GPU that yields the best performance
Adaptive [18]	Partition workload based on online profiling
Dynamic [30]	Both GPU and CPU execute the workload from opposite directions
Coarse-grained Oracle [40]	Coarse-grained workload partitioning with optimal load balance
FinePar	Fine-grained workload partitioning based on Linear Regression

Table 2. Summary of different partitioning methods.

We compare our method with 4 state-of-the-art workload partitioning methods on heterogeneous platforms listed in Table 2. The Single-Device method [5] uses the device from the CPU and GPU that produces better performance. The adaptive method [18] calculates a performance ratio between CPU and GPU through executing partial workload and then partitions the workload using this ratio. The dynamic method [30] 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. The coarse-grained oracle method [39, 40] performs workload partitioning from 0 to 100% and selects the best partitioning ratio.

Figure 7 shows 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 is 1.53X over the single-device method. For the *FullChip* matrix, the performance speedup is up to 2.31X. The average speedup is 0.87X for the adaptive method, 0.70X for the dynamic method, and 1.11X for the coarse-grained oracle method. FinePar achieves an average of 1.38X speedup over the coarse-grained oracle method.

In Figure 7, we can see that not all the partitioning methods can achieve consistent performance improvement over the optimal single device method. 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

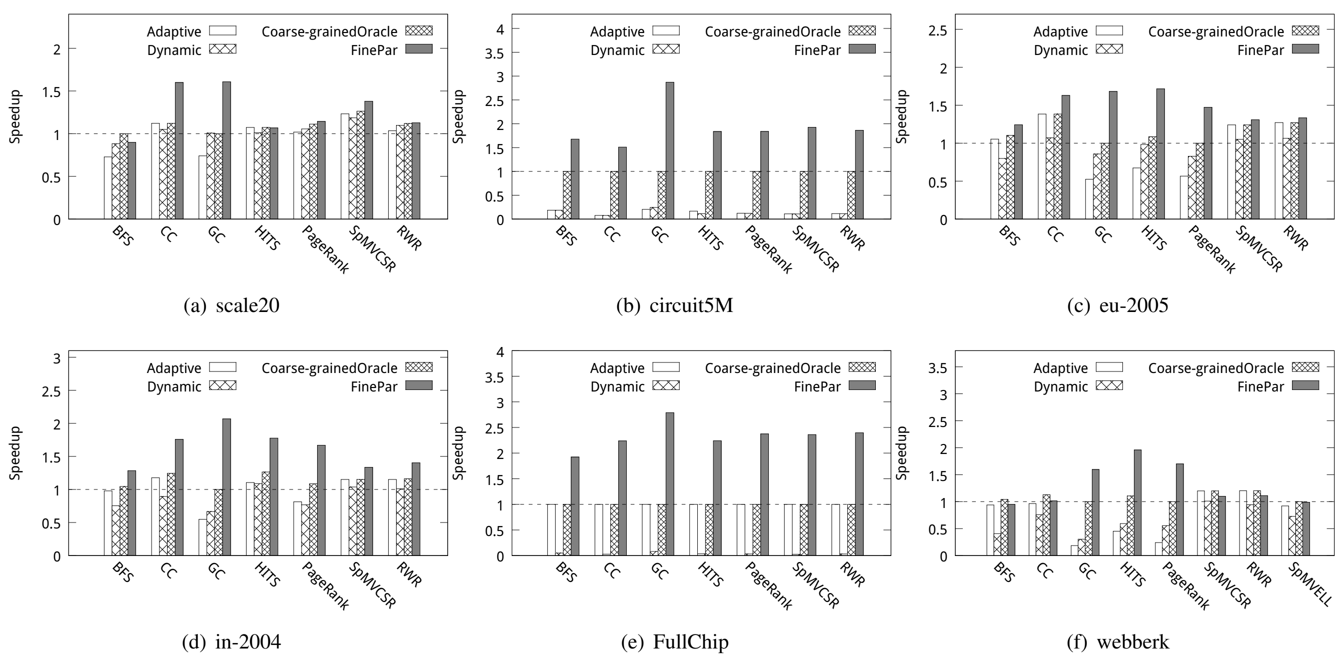


Figure 7. Performance results of different partitioning methods. The base line is the optimal single device result, GPU- or CPU- only version.

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.

In particular, our method has performance degradation for BFS with *scale20* and *webberk*. This is because for BFS, the workloads across iterations change dramatically depending on the graph topology. Currently, our performance model cannot handle such dynamic behavior.

4.3 Result Analysis of FinePar

In general, FinePar 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.

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
in-2004	37.23	13.28	175.70	2.93
FullChip	1806.80	2.74	26307.03	3.80
web-BerkStan	16.36	6.73	29.49	1.64

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

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 *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.

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 7 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. FinePar can get very high performance improvements for these inputs. Second, the matrices of *eu-2005*, *web-BerkStan*, and *in-2004* have the moderate irregularity and there is no significant irregularity difference between CPU and GPU after the fined-grained partitioning. However, FinePar can also produce moderate performance improvement for these inputs. Third, for *scale20*, its irregularity is uniformly distributed in the whole matrix, so it is difficult to greatly reduce its irregularity. Table 3 shows that the GPU workload still has a variance of 56.57 after fine-grained partitioning. Consequently, the performance improvement is limited for this matrix.

4.4 Accuracy of Performance Models

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

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.

Table 4 shows the trained parameters of the performance models from our offline training. We use a statistical metric, called coefficient of determination [3], 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.

To understand the importance of the features in the models for different devices and types of matrices, we provide their correlation coefficients [1] 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.

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 8 shows the comparison between the improvement by FinePar and the optimal performance improvement (named *Oracle*).

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

Table 5. Correlation coefficient of the features in the performance model of FinePar.

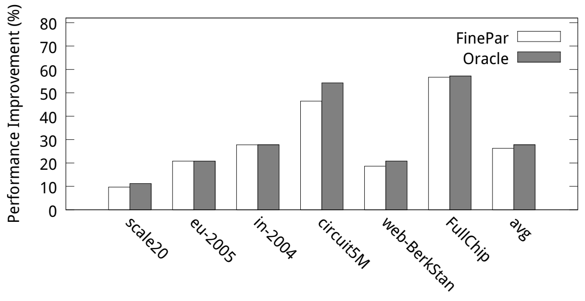


Figure 8. Performance improvement between FinePar and the optimal partition.

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 32% on average while FinePar achieves an average of 30% performance improvement.

4.5 Performance Overhead Analysis

4.5.1 Time Overhead

Before kernel computation, the evaluated programs perform I/O operations and data initialization. FinePar 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 FinePar compared to the original pre-kernel processing time for each program for the matrix *web-BerkStan*. The other matrices have similar performance overhead. We observe that FinePar's overhead only accounts for less than 0.2% of the pre-kernel processing time, which is negligible.

Program	I/O(%)	Initialization(%)	FinePar(%)
BFS	74.67	25.25	0.08
ConnectedComp	31.27	68.70	0.03
GraphColoring	58.91	41.05	0.04
HITS	52.36	47.51	0.13
PageRank	55.22	44.73	0.05
SpMV-CSR	75.00	24.90	0.10
RWR	71.78	28.08	0.14
SpMV-ELL	15.53	84.44	0.03

Table 6. The time overhead of FinePar.

4.5.2 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:

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

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, FinePar only uses a regular integer array of size n for the CPU. Hence, the space overhead incurred by FinePar is:

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

Matrix	Original (Bytes)	Extra Allocation (Bytes)	Space Overhead(%)
scale20	259M	4M	1.67
circuit5M	521M	23M	4.40
eu-2005	161M	4M	2.21
in-2004	146M	6M	3.90
web-BerkStan	66M	3M	5.17
FullChip	237M	12M	5.20

Table 7. The space overhead of FinePar.

Table 7 shows the extra space overhead introduced by FinePar. The column named "Original (Bytes)" shows the original size for each matrix. The column named "Extra Allocation (Bytes)" shows the size incurred by FinePar. The last column shows FinePar's space overhead normalized to the original size. For all the inputs, our method introduces little space overhead (less than 6%).

5. Related Work

5.1 Workload Partitioning

Heterogeneous architectures introduce non-trivial challenges for workload partitioning due to the drastically different architectures between the CPU and GPU. Lee et al. [21] proposed SKMD (Single Kernel Multiple Devices), which transparently orchestrates multiple devices to execute the same OpenCL kernel on systems with discrete GPUs. The partitioner of SKMD considered data transfer overhead and performance variation across devices. Pandit et al. [30] proposed a dynamic workload partitioning approach to balancing the workload between the CPU and the discrete GPU. Huchant et al. [17] proposed compiler and runtime techniques to automatically partition work groups of the same OpenCL program to multiple CPUs and GPUs. Their evaluation on an N-body application with two heterogeneous systems demonstrated good load balance across devices. Those approaches all leveraged some form of coarse-grained partitioning, which was shown to be ineffective for irregular applications on integrated architectures.

5.2 Optimizing Irregular Applications

Most recent efforts in optimizing irregular applications in heterogeneous architectures were focused on GPUs. Several new storage formats were proposed to optimize sparse matrix applications [4, 32, 34, 38]. Sedaghati et al. [32] characterized sparsity features of many sparse matrices and built a model to automatically select the most suitable format during runtime. Ashari et al. [4] represented graph algorithms in SpMV and optimized load balance among GPU threads by reordering the rows in the sparse matrix. In comparison, FinePar targets integrated architectures with both CPU and GPU, and hence considers the architecture differences.

5.3 Performance Modeling

There exists some work on building performance models for sparse matrix applications. Li et al. [24] proposed probabilistic models for SpMV. Li et al. [23] used a machine learning-based model to auto-tune matrix format for SpMV. Some machine learning based models were proposed for optimization. Leather et al. [20] provided automatic feature generation for optimizing compilation. Wang el al. [37] provided a machine learning based prediction for better mapping. Those performance models focused on one device, while FinePar builds performance models for both the CPU and GPU that take into consideration architecture differences and input features. Gharaibeh et al. [14] proposed performance models to partition graph workload for the CPU and GPU for load balance, but did not consider architecture differences.

6. Conclusion

In this paper, we identified the pitfall of coarse-grained workload partitioning for irregular applications on integrated architectures. To deal with the problem, we developed a compilation and runtime system named FinePar to achieve fine-grained partitioning. FinePar automatically transforms the input irregular kernel to use both the CPU and GPU, and builds irregularity-aware performance models for partitioning the workload during runtime through auto-tuning. Experimental results on six graphs and eight applications demonstrated 1.38X performance speedup over the optimal coarse-grained partitioning.

Acknowledgements

This work is partially supported by National Key Research and Development Program of China 2016YFB0200100, NSFC Project 61472201, Tsinghua University Initiative Scientific Research Program, Huawei Innovation Research Program (HIRP), Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology. Bingsheng's research is partly supported by an MoE AcRF Tier 1 grant (T1 251RES1610) from Singapore. Bo's research is partly supported by the National Science Foundation (NSF) under Grant No. 1464216 and No. 1618912. Jidong Zhai is the corresponding author (Email: zhaijidong@tsinghua.edu.cn).

References

[1] CORREL function. `https://support.office.com/en-us/article/`.

[2] The Compute Architecture of Intel Processor Graphics Gen7.5. `https://software.intel.com/sites/default/files/managed/4f/e0/Compute_Architecture_of_Intel_Processor_Graphics_Gen7dot5_Aug4_2014.pdf`.

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

[4] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 781–792. IEEE, 2014.

[5] 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 *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 33. ACM, 2014.

[6] D. Bouvier and B. Sander. Applying AMDs Kaveri APU for heterogeneous computing. In *Hot Chips: A Symposium on High Performance Chips (HC26)*, 2014.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[8] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[9] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012.

[10] M. Daga, M. Nutter, and M. Meswani. Efficient breadth-first search on a heterogeneous processor. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 373–382. IEEE, 2014.

[11] 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*, pages 63–74. ACM, 2010.

[12] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[13] M. C. Delorme, T. S. Abdelrahman, and C. Zhao. Parallel radix sort on the AMD fusion accelerated processing unit. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 339–348. IEEE, 2013.

[14] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, 2012. ISBN 978-1-4503-1182-3.

[15] J. L. Greathouse and M. Daga. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 769–780. IEEE, 2014.

[16] J. He, M. Lu, and B. He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *VLDB'13*, 6 (10):889–900, 2013.

[17] P. Huchant, M. C. Counilh, and D. Barthou. Automatic opencl task adaptation for heterogeneous architectures. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, pages 684–696, 2016.

[18] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 151–162. ACM, 2014.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[20] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 81–91. IEEE, 2009.

[21] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, 2013. ISBN 978-1-4799-1021-2.

[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[23] J. Li, G. Tan, M. Chen, and N. Sun. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[24] K. Li, W. Yang, and K. Li. Performance analysis and optimization for spmv on GPU using probabilistic modeling. *IEEE Trans. Parallel Distrib. Syst.*, 26(1):196–205, 2015.

[25] W. Liu and B. Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

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

[27] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. ACM, 2015.

[28] 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*, pages 2–7, 2014.

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

[30] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.

[31] S. Russell, P. Norvig, and A. Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[32] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 99–108, 2015. ISBN 978-1-4503-3559-1.

[33] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[34] B.-Y. Su and K. Keutzer. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 353–364. ACM, 2012.

[35] H. Tong, C. Faloutsos, and J.-Y. Pan. Random walk with restart: fast solutions and applications. *Knowledge and Information Systems*, 14(3):327–346, 2008.

[36] H. Wang, W. Liu, K. Hou, and W.-c. Feng. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing*, page 33. ACM, 2016.

[37] Z. Wang, G. Tournavitis, B. Franke, and M. F. O'boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):2, 2014.

[38] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaSpMV: Yet another SpMV framework on GPUs. In *ACM SIGPLAN Notices*, volume 49, pages 107–118. ACM, 2014.

[39] F. Zhang, J. Zhai, W. Chen, B. He, and S. Zhang. To Co-Run, or Not To Co-Run: A Performance Study on Integrated Architectures. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 89–92. IEEE, 2015.

[40] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Trans. Parallel Distrib. Syst.*, pages 1–1, 2016. doi: 10.1109/tpds.2016.2586074. URL `http://dx.doi.org/10.1109/TPDS.2016.2586074`.