Understanding Co-running Behaviors on Integrated CPU/GPU Architectures

Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, Wenguang Chen

Abstract—Architecture designers tend to integrate both CPUs and GPUs on the same chip to deliver energy-efficient designs. It is still an open problem to effectively leverage the advantages of both CPUs and GPUs on integrated architectures. In this work, we port 42 programs in Rodinia, Parboil, and Polybench benchmark suites and analyze the co-running behaviors of these programs on both AMD and Intel integrated architectures. We find that co-running performance is not always better than running the program only with CPUs or GPUs. Among these programs, only 8 programs can benefit from the co-running, while 24 programs only using GPUs and 7 programs only using CPUs achieve the best performance. The remaining 3 programs show little performance preference for different devices. Through extensive workload characterization analysis, we find that architecture differences between CPUs and GPUs and limited shared memory bandwidth are two main factors affecting current co-running performance. Since not all the programs can benefit from integrated architectures, we build an automatic decision-tree-based model to help application developers predict the co-running performance for a given CPU-only or GPU-only program. Results show that our model correctly predicts 14 programs out of 15 for evaluated programs. For a co-run friendly program, we further propose a profiling-based method to predict the optimal workload partition ratio between CPUs and GPUs. Results show that our model can achieve 87.7% of the optimal performance relative to the best partition. The co-running programs acquired with our method outperform the original CPU-only and GPU-only programs by 34.5% and 20.9% respectively.

Index Terms—Heterogeneous Computing, Integrated Architecture, Performance Prediction, Performance Tuning, Workload Characterization.

1 Introduction

Integrating GPUs with CPUs on the same chip is increasingly common in current processor architectures. In 2011, AMD released its integrated architecture [1], called accelerated processing units (APUs). Subsequently, Intel also released integrated architectures in the processors of Ivy Bridge and Haswell [2]. A main advantage of integrated architectures is that both CPUs and GPUs share the same physical memory, which can significantly reduce data transmission requirements through the PCIe bus in traditional architectures using discrete GPUs.

These integrated architectures are new and have many challenging research problems compared with discrete architectures. First, although integrated GPUs (iGPU) can reduce data transmissions from PCIe, the memory access speed of integrated GPUs is considerably lower than the video memory access speed of discrete GPUs (dGPU). Second, because of power concerns, integrated GPUs have less computing capacity than discrete GPUs do. Third, the integrated architectures increase memory access pressure due to unified memory channels. Since the integrated architecture has only been available for a few years, researchers are still exploring how to make full use of integrated architectures.

To effectively leverage the power of both CPUs and GPUs on integrated architectures, researchers have recently focused on co-running a single application using both the CPU and GPU on such architectures. He et al. [3], [4] employed an integrated architecture to optimize Hash Join, which is an important type of operation in databases. Delorme et al. [5] implemented a parallel radix sort using an integrated architecture. Daga et al. [6] showed the promising performance of Breadth-First Search on an integrated architecture. Chen et al. [7] accelerated MapReduce programs on an integrated architecture. Eberhart et al. [8] used the AMD APU to accelerate stencil computations. Different from the co-running study in our paper, Zhu et al. [9] analyzed the performance of executing different applications simultaneously on the integrated architecture.

Despite previous efforts, few studies have been performed to analyze a wide range of co-running applications on such heterogeneous architectures to obtain a comprehensive understanding of varied workload characterizations. A detailed workload characterization of integrated architectures is indispensable for both designing future integrated system and assisting application developers in effective programming.

To this end, we explore a large number of programs through adopting a strategy of data partition that can fully utilize the GPU and CPU resources on integrated architectures. We port all the programs from three varied benchmark suites, Rodinia [10], Parboil [11], and Polybench [12]. There are 42 parallel programs in total, which cover a diversity of topics in parallel computing. We rewrite these programs with the OpenCL framework [13] to enable them to co-run on both CPUs and GPUs, and also provide an automatic partition method to achieve good load balance between CPUs and GPUs. We co-run all of these programs

- *F. Zhang, J. Zhai and W. Chen are with the Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China. Email: feng-zhang12@mails.tsinghua.edu.cn, zhaijidong@tsinghua.edu.cn, cwg@tsinghua.edu.cn.*
- *B. He and S. Zhang are with the School of Computing, National University of Singapore, 119077, Singapore. Email: hebs@comp.nus.edu.sg, shuhao.zhang@sap.com.*

on both AMD and Intel integrated architectures and get several interesting findings different from previous studies.

Among 42 programs we ported to integrated architectures, only 8 programs are co-run friendly (the program performance of co-running on both CPUs and GPUs is the best), 24 programs are GPU dominant (the program performance of only running on GPUs is the best), 7 programs are CPU-dominant (the program performance of only running on CPUs is the best), and 3 programs show no performance preference for different devices. In general, the co-running results on integrated architectures are varied for different types of programs and only a small but non-neglectable portion of programs can benefit from co-running. To get a comprehensive understanding of co-running behaviors on such integrated architectures, we perform a series of workload characterization analysis and obtain a lot of useful findings for future application developers. Specifically, we study the co-running workload characterization on integrated architectures from the following three aspects.

First, to identify the main factors causing the co-running performance degradation, we analyze several factors that may affect the co-running performance. Based on experimental results, we find two main factors affecting the co-running performance. (1) Large architecture differences between GPUs and CPUs on integrated architectures is the main challenge for effective co-running. High-level programming models, such as OpenCL, provide uniform programming interfaces to hide the differences of underlying hardware devices. However, if a co-running program is not aware of such hardware differences, the final performance will be significantly affected. Our study reveals that the usage of local memory and memory access patterns are two key differences between computation on CPUs and GPUs. To achieve better performance on integrated architectures, the co-running programs need to be tuned separately for different kinds of devices. Our tuning results show that such optimizations can get 8.3% performance improvement on average. (2) On current integrated processors, the shared memory bandwidth between GPUs and CPUs is a main bottleneck for the co-running performance. Only partial programs with low memory bandwidth requirements can benefit from integrated processors. If we can remove the memory bandwidth limitation on integrated architectures, there will be more co-run friendly programs.

Second, power is another important aspect that needs to be considered on the integrated architecture. AMD argues that the integrated architecture APU is an important balance of performance and power and can provide much better power-efficiency [14] than discrete architectures. Intel also regards the integrated architecture as a power optimization choice [15]. In this paper, we perform extensive power and energy testing on both AMD and Intel integrated architectures. We analyze the results on both AMD A10-7850K and Intel i7-4770R and compare them with those on a discrete GPU, an NVIDIA K20c. Results show that not all integrated architectures are more energy efficient than discrete architectures. Although integrated architectures normally have relatively low real-time power, they also have low computation performance compared to discrete architectures.

Third, since not all the programs can benefit from integrated architectures, it is necessary to identify which programs are co-run friendly before porting them. To address this problem, we adopt a black-box machine learning method, C4.5 algorithm [16], to build a decision-tree-based prediction model. With this model, we can help users to predict the co-running type for a given GPU-only or CPU-only program before porting it. Experimental results show that our model can correctly predict 14 programs out of 15 for the evaluated benchmark. Furthermore, for a co-run friendly program, we propose a profiling-based method to help users estimate the optimal workload partition ratio between GPUs and CPUs. Results show that our method can achieve 87.7% of the optimal performance relative to the best partition. Overall, the co-running programs acquired with our method outperform the original CPU-only and GPU-only programs by 34.5% and 20.9% respectively.

In summary, our study has shed light on the future of co-running applications. To the best of our knowledge, this is the first work to analyze a wide range of workload characterization on integrated architectures. We have opened the programs ported in this work on github[1] and hope that more researchers can use these programs for related research.

2 Background

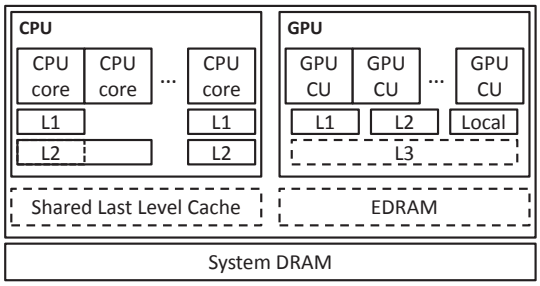


Fig. 1. A high-level view of an integrated CPU/GPU architecture.

Integrated architectures, increasingly popular in recent years, have integrated CPUs with accelerators such as GPUs and FPGAs (Field-Programmable Gate Arrays), on the same die. In this work, we focus on the integrated CPU/GPU architectures, which integrate CPUs and GPUs on the same chip, sharing the physical memory. We perform the workload characterization analysis on two typical integrated architectures: AMD's integrated architecture, A-Series APU A10-7850K (codenamed "Kaveri" [14]), and Intel's integrated architecture, Haswell i7-4770R.

Figure 1 gives a general view for the integrated architecture. For CPUs, they normally have L1 data caches and L1 instruction caches. Some have private L2 caches, such as Intel Haswell i7-4770R, while others have shared L2 caches, such as AMD A10-7850K. For GPUs, L1 caches, L2 caches, and local memory more often have different organization levels. Intel Haswell i7-4770R even has a shared L3 cache in GPUs. Some of Intel integrated architectures have a shared last level cache (LLC) and an embedded DRAM (EDRAM) for both CPUs and GPUs. For both AMD and Intel integrated architectures, CPUs and GPUs share the

1. https://github.com/zhangfengthu/CoRunBench

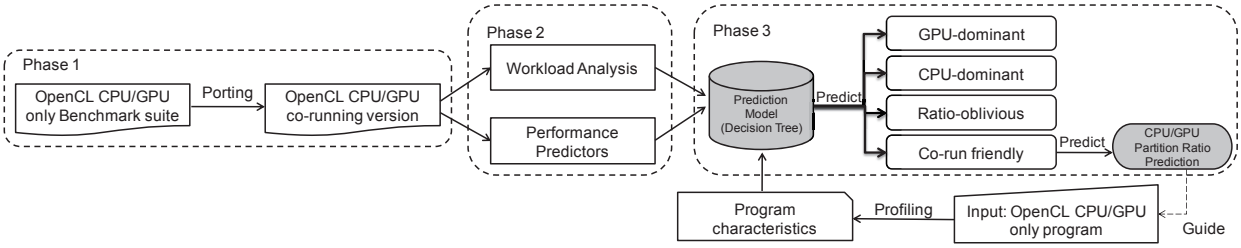


Fig. 2. Co-running program analyzer overview.

same physical memory and the same memory controller. Based on our experiments, although the CPUs and GPUs are on the same chip, they always have different memory bandwidths. The bandwidth provided for GPUs is normally much higher than the bandwidth provided for CPUs.

OpenCL [13] is an open standard computing language for heterogeneous architectures. We briefly introduce the workflow of an OpenCL program on a single device. At first, a command queue is created within a context. A command of executing a specific computing kernel is then enqueued with an OpenCL function *clEnqueueNDRangeKernel()*. After the enqueue operation, another standard OpenCL function called *clFlush()* is invoked to guarantee that those commands have been issued to the associated devices such as CPUs or GPUs. Finally, a blocking operation *clWaitForEvents()* is invoked to wait for all above executions to be completed. In OpenCL, all allocated resources need to be explicitly released after the execution of the computing kernel.

3 METHODOLOGY

In this section, we first give an overview of the co-running analysis. Then, we present how to rewrite a GPU-only or CPU-only program on an integrated architecture. At last, we present the methods used to analyze the workload characterizations and our prediction models.

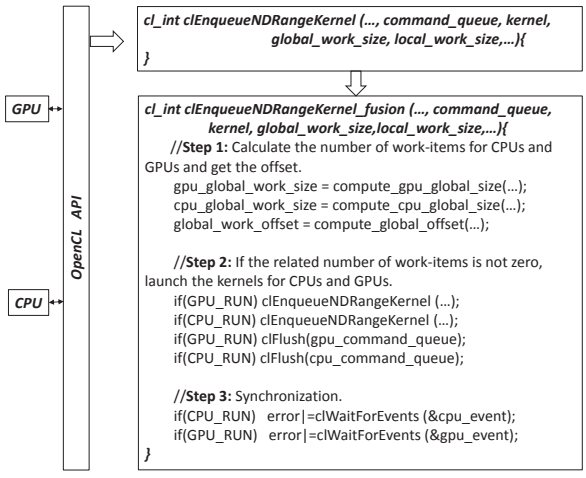


Fig. 3. A pseudo-code of workload partition for integrated architectures.

3.1 System Overview

We provide a program analyzer for co-running analysis on integrated architectures. The analyzer is shown in Figure 2, which consists of three main phases. First, since most of current programs are designed for CPU-only or GPU-only platforms and they cannot be directly executed on integrated architectures, we need to port these programs to make them be able to harness both CPUs' and GPUs' computing simultaneously. Second, to understand the co-running behaviors of these programs, we perform extensive workload characterization analysis and then summarize the main issues causing co-running performance degradation. Finally, based on collected performance data and workload characterization analysis, we build a prediction model to help users to predict the co-running performance. For a co-run friendly program, our model can further predict the optimal partition ratio between CPUs and GPUs. We elaborate each of these phases in the following sections.

3.2 Co-running Implementation

We describe how to co-run a program on an integrated architecture. In this study, we focus on data-parallel programs using OpenCL. The input of our program analyzer is an OpenCL program written for a single CPU or GPU device and the output is a co-running version.

To make the OpenCL program harness the power of both CPUs and GPUs on integrated architectures, we need to carefully partition computing kernels onto both devices. A main challenge is that how to determine the suitable partition ratio between CPUs and GPUs. In this work, we use a profiling-based method to estimate the optimal partition ratio, which will be elaborated in Section 5. We give a description of basic partition strategy for co-running. First, we create two devices and two command queues in a shared context, instead of one device and one command queue. Second, we replace the original invocation *clEnqueueNDRangeKernel()* with a new function *clEnqueueNDRangeKernel_fusion()*. The main purpose of this step is to partition computing kernels onto different devices. During this step, we need to compute the optimal partition ratio, the number of work items for CPUs and GPUs, and their starting numbers. At last, we launch the computing kernels and wait for them to complete. A pseudo-code is listed in Figure 3. In the program, we use a global variable to record the workload partition ratio for CPUs and GPUs. If the variable is 0 or 100%, it means that the program is only executed on one device.

3.3 Workload Characterization Analysis

We divide the co-running programs into two categories: co-run friendly and co-run unfriendly. For the co-run un-

friendly category, we further subdivide it into three categories: GPU-dominant, CPU-dominant, and ratio-oblivious.

Co-run friendly programs: Co-run friendly programs achieve the best performance when they use both CPUs and GPUs together to process the workload.

GPU-dominant programs: GPU-dominant programs achieve the best performance when they only use GPUs to process the workload.

CPU-dominant programs: CPU-dominant programs achieve the best performance when they only use CPUs to process the workload.

Ratio-oblivious programs: Ratio-oblivious programs exhibit no preference for different devices when using different workload partition ratios.

For a given integrated architecture, we define a threshold to quantify each category (in Section 4.2).

To get a comprehensive understanding of co-running behaviors, we perform a series of workload characterization analysis from both performance and power. In terms of performance, we compare the co-running performance with GPU-only and CPU-only executions. For programs that cannot benefit from integrated architectures, we further identify what factors affect the co-running performance. In general, we investigate the main factors that may affect co-running, including the usage of local memory, memory access patterns, the degree of parallelism, shared memory bandwidth, and the overlap ratio between GPU and CPU kernels. We analyze the above factors based on the following considerations. First, there are significant architecture differences between GPUs and CPUs on the integrated architectures, such as the usage of local memory and memory access patterns. It is necessary to quantify the influence of such architecture differences on co-running performance. Second, we also study the new features introduced by the integrated architecture, such as shared memory, and analyze the effect of them on the co-running performance.

Power is another important characteristic for integrated architectures, we use a power meter to analyze the power characteristic for the evaluated programs and also compare both power and energy consumption between the integrated architecture and the discrete architecture.

3.4 Prediction Models

To help application developers to identify what kind of programs can benefit from integrated architectures before porting them, we use a black-box machine learning method to build a decision-tree-based prediction model, which is an automatic model without any users' intervention. We use the C4.5 algorithm to build a decision tree for program classification. Training data used to build the decision tree model is collected according to the above co-running workload characteristic analysis. The co-running performance data of Rodinia and Parboil benchmark suites is used to build the prediction model, and the Polybench benchmark is used to validate the model.

Furthermore, for a co-run friendly program, we propose a profiling-based prediction model, which can help us predict the optimal partition ratio between CPUs and GPUs. The input of our model is the profiling data on both CPUs and GPUs. The predicted partition ratio can be further input to our program analyzer to guide the program co-running implementation as shown in Figure 2. We will elaborate this part in Section 5.1.

4 Co-running Characteristics and Tuning

We analyze co-running characteristics and identify main factors causing performance degradation on integrated architectures. We also try to tune some key parameters according to the characteristics of integrated architectures for better co-running performance. Finally, we analyze power and energy characteristics of integrated architectures.

4.1 Platforms

We select two typical integrated architectures to analyze co-running characteristics, AMD's integrated architecture, A-Series APU A10-7850K (codenamed "Kaveri [14]"), and Intel's integrated architecture, Haswell i7-4770R. For the AMD platform, its operating system is Ubuntu 13.10, and the CPU and GPU peak FLOPS are 118.4 and 737.3 Gflops/s, respectively. For the Intel platform, its operating system is Windows 8.1, and the CPU and GPU peak FLOPS are 204.8 and 832 Gflops/s respectively. The CPU and GPU peak FLOPS are with the performance of 32-bit single precision floats. The memory frequency is 1600 MHz and the maximum bandwidth is 25.6 GB/s for both platforms.

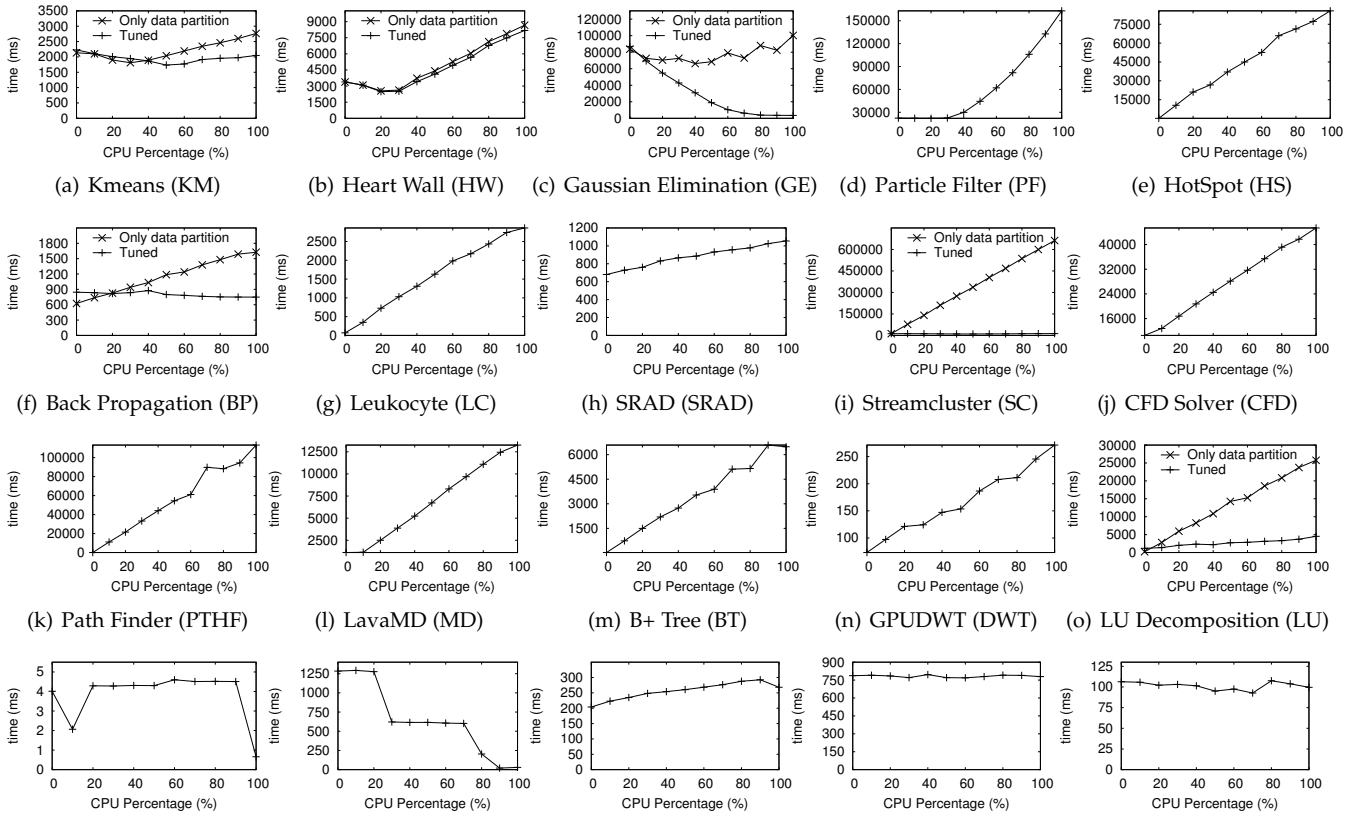
Note that although Intel processors integrate both CPUs and GPUs on the same chip and also have a shared last level cache (LLC) between them, cache coherence for two devices within the same cache line cannot be guaranteed in current OpenCL implementations [17]. Moreover, on the Intel integrated processor, only CPU devices support double precision floating point computation. Our experimental results show that only ATAX, CORR, COVAR, and GESUMMV of Polybench can be correctly executed for co-running on the Intel platform. Therefore, we only co-run partial programs of Polybench on the Intel platform. The processor i7-4770R used in the experiment is the Intel 4th generation [18] which only supports OpenCL 1.2. The latter Intel chips supporting OpenCL 2.0 provide fine-grained shared memory access and are able to guarantee cache coherence [19]. For the AMD platform, when both CPUs and GPUs write to the same location, the coherence cannot be guaranteed, even using atomic operations. Therefore, for the Parboil benchmark, only 7 programs can be correctly co-run on the AMD platform. This limitation is because the current implementation of atomic operations is only valid on a single device. In later versions that both CPUs and GPUs support OpenCL 2.0, this issue can be solved.

4.2 Overall Co-Running Results

To effectively distinguish the different categories of evaluated programs, we define two metrics, $CorunIndicator$ and $DeviceChoosingIndicator$. $CorunIndicator$ can help us distinguish co-run friendly and unfriendly programs while $DeviceChoosingIndicator$ can help us further identify CPU-dominant and GPU-dominant programs from the co-run unfriendly programs. To reduce the effect of random errors, a threshold is given for each metric. In this work, we set the threshold of $CorunIndicator$ to 0.85 and

TABLE 1
Co-running classification results of Rodinia benchmarks on A10-7850K.

Applications	Main Kernel Name	Input Data Set	*Co-run Indicator*	*Device Choosing Indicator*	Category
Leukocyte (LC)	IMGVF_kernel	testfile.avi	1.00	40.56	GPU-dominant
Heart Wall (HW)	kernel_gpu_opencl	number of frame 20	0.76	1.56	Co-run friendly
CFD Solver (CFD)	compute_flux	fvcorr.domn.097K	1.00	3.31	GPU-dominant
LU Decomposition (LU)	lud_internal	s 2048	1.00	113.34	GPU-dominant
HotSpot (HS)	hotspot	512 2 1000	1.00	506.87	GPU-dominant
Back Propagation (BP)	bpnn_adjust_weights_ocl	524288	1.00	1.60	GPU-dominant
Needleman-Wunsch (NW)	nw_kernel2	4096 10	1.00	0.31	Ratio-oblivious
Kmeans (KM)	kmeans_kernel_c	kdd_cup	0.84	0.29	Co-run friendly
Breadth-First Search (BFS)	BFS_1	graph1MW 6.txt	0.93	0.07	Ratio-oblivious
SRAD (SRAD)	srad_kernel	100 0.5 502 458	1.00	0.55	GPU-dominant
Streamcluster (SC)	pgain_kernel	10 20 256 65536 65536 1000	1.00	53.68	GPU-dominant
Particle Filter (PF)	find_index_kernel	x 128 y 128 z 10 np 400000	0.99	6.29	GPU-dominant
Path Finder (PTHF)	dynproc_kernel	100000 100 20	1.00	2217.29	GPU-dominant
Gaussian Elimination (GE)	Fan2	s 1024	0.79	0.20	Co-run friendly
k-Nearest Neighbors (NN)	NearestNeighbor	r 5 lat 30 lng 90	1.00	5.08	CPU-dominant
LavaMD (MD)	kernel_gpu_opencl	boxes1d 20	1.00	10.85	GPU-dominant
Myocyte (MC)	kernel_gpu_opencl	time 100	0.93	8.59	CPU-dominant
B+ Tree (BT)	findRangeK	j 65536 10000	1.00	256.43	GPU-dominant
GPUDWT (DWT)	cl_fdwt53Kernel	rgb.bmp d 1024x1024 f 5 l 3	1.00	2.71	GPU-dominant
Hybrid Sort (HYS)	bucketsort	r	0.98	0.01	Ratio-oblivious



(a) Kmeans (KM) (b) Heart Wall (HW) (c) Gaussian Elimination (GE) (d) Particle Filter (PF) (e) HotSpot (HS)

(f) Back Propagation (BP) (g) Leukocyte (LC) (h) SRAD (SRAD) (i) Streamcluster (SC) (j) CFD Solver (CFD)

(k) Path Finder (PTHF) (l) LavaMD (MD) (m) B+ Tree (BT) (n) GPUDWT (DWT) (o) LU Decomposition (LU)

(p) k-Nearest Neighbors (NN) (q) Myocyte (MC) (r) Needleman-Wunsch (NW) (s) Hybrid Sort (HYS) (t) Breadth-First Search (BFS)

Fig. 4. Co-running results for the Rodinia benchmarks for different partition ratios. When we only perform data partition for these programs (the default lines), (a, b, c) are co-run friendly programs, (d, e, f, g, h, i, j, k, l, m, n, o) are GPU-dominant programs, (p, q) are CPU-dominant programs, and (r, s, t) are ratio-oblivious programs. After performing optimization, the co-running performance of some programs can be improved (the lines labeled with "Tuned").

$DeviceChoosingIndicator$ to 0.4, which can be adjusted for other platforms. These metrics are defined as follows.

$$CorunIndicator = \frac{min\_all}{min\_atipodes} \quad (1)$$

$$DeviceChoosingIndicator = \frac{|t_{cpu\_100} - t_{cpu\_0}|}{min\_atipodes} \quad (2)$$

$$min\_all = min(t_{cpu\_0}, ..., t_{cpu\_100}) \quad (3)$$

$$min\_atipodes = min(t_{cpu\_0}, t_{cpu\_100}) \quad (4)$$

t_{cpu_i} denotes the execution time when CPUs process i% of the workload while GPUs process the remaining workload. $min(t_{cpu\_i}, \ldots, t_{cpu\_j})$ denotes the minimum value from t_{cpu_i} to t_{cpu_j}. $min(t_{cpu\_i}, t_{cpu\_j})$ denotes the minimum value between t_{cpu_i} and t_{cpu_j}.

Due to space limitation, we only show the classification results of the Rodinia benchmark suite [10] on the AMD integrated architecture in Table 1. The classifying metrics and input data set for each program are also listed in this table. The detailed co-running results of Rodinia benchmark are shown in Figure 4. We change the load partition ratio from 0 to 100% with an interval of 10%.

For the co-run friendly programs, the curve of execution time for different partition ratios is a concave curve. There is an optimal partition ratio for each program. For the GPU-dominant and CPU-dominant programs, the curve of execution time is normally a monotonic curve. For the ratio-oblivious programs, it normally presents a horizontal line. Among these programs, there are 3 co-run friendly programs, 12 GPU-dominant programs, 2 CPU-dominant programs, and 3 ratio-oblivious programs. Note that these are the classification results when we only perform data partition between CPUs and GPUs without optimizations. In Section 4.3.1, we will further present optimization results for these programs. Overall, co-running does not bring too much performance improvement for most of the programs. In contrast, sometimes co-running causes significant performance degradation for many programs.

We summarize the main characteristics for four types of programs when we only perform data partitioning.

Co-run friendly programs (KM, HW, and GE): Co-run friendly programs normally have low memory bandwidth requirements compared to co-run unfriendly programs. From the aspect of programming models, co-run friendly programs usually use little local memory and tend to have long kernel execution times.

GPU-dominant programs (PF, HS, BP, LC, SRAD, SC, CFD, PTHF, MD, BT, DWT, and LU): GPU-dominant programs usually have a high degree of parallelism and can fully exert the computation power of GPU devices. From the aspect of programming models, these programs normally use a large amount of local memory to reduce global memory access.

CPU-dominant programs (NN and MC): Since the peak performance of CPUs is much lower than that of GPUs on current integrated architectures, few programs present CPU-dominant behavior. CPU-dominant programs have low degree of parallelism and use little local memory. The kernel execution time of these programs is relatively short.

Ratio-oblivious programs (NW, BFS, and HYS): For NW and BFS, since the memory bandwidth becomes the main bottleneck, the performance for different workload partition ratios does not change obviously. For HYS, there are 7 kernels and the main kernel only accounts for a small fraction of total execution time.

TABLE 2
Optimization strategies adopted when porting GPU-only programs into integrated architectures.

	HW	KM	SC	GE	COVAR	SYR2K	CORR	SYRK
Removing local memory usage			✓					
Tuning data memory layout		✓	✓		✓		✓	
Loop tiling		✓						✓
Adjusting the degree of parallelism	✓	✓		✓				✓
Common subexpression elimination						✓		

4.3 Performance Degradation Analysis and Tuning

Theoretically, without other limits, if we can exert the computation power of both GPUs and CPUs, the co-running execution should outperform the GPU-only or CPU-only execution. To get a comprehensive understanding of co-running performance degradation, we analyze several main factors that may affect the co-running performance from the following points.

(1) *Architecture difference between GPUs and CPUs.* There are significant architecture differences between GPUs and CPUs. First, GPUs provide local memory that programmers need to explicitly manage. Second, GPUs and CPUs adopt different execution models that are in favor of different data memory layout. GPUs have a great number of computing cores, which can launch many light-weight threads simultaneously to hide the latency of global memory access, but CPUs have relatively large caches, which are beneficial for programs with good locality. Therefore, if the co-running programs are not aware of such architecture differences, the final performance will be affected to some degree.

(2) *New features of integrated architectures.* The integrated architectures introduce many new features compared with the traditional discrete architectures. First, integrated architectures share the memory bandwidth between GPUs and CPUs. It is interesting to quantify the effect of memory bandwidth on the co-running performance. Second, since computation kernels on GPUs and CPUs are launched asynchronously, it is necessary to identify the overlap ratios between GPU and CPU kernels.

4.3.1 Architecture Difference between GPUs and CPUs

Finding (1): There are significant architecture differences between GPUs and CPUs on the integrated architecture, such as local memory usage and memory access patterns. If a co-running program is not aware of such difference, the final performance will be compromised. Performance tuning targeting different kinds of devices is necessary to achieve better performance.

When porting GPU-only or CPU-only programs into integrated architectures, we find that the architecture difference between two types of devices is a main factor affecting the co-running performance. We list the main differences below.

(1) *Local memory usage.* On GPU devices, there is a small-size and high-speed local memory on the chip and programmers can reduce global memory access by keeping frequently accessed data in the local memory. The OpenCL programming model also provides corresponding interfaces to use the local memory. However, CPUs do not have physical local memory and the local memory is emulated by programming libraries. If the programs largely use the local memory on CPUs, they will introduce redundant memory access. Hence, the local memory usage on CPUs can cause some performance degradation.

(2) *Memory Access Patterns.* Since GPUs and CPUs adopt different thread execution models, there is great difference in memory access patterns between the two devices. GPU programs normally use a large number of threads to hide the latency of global memory access, while CPU programs normally use number-of-core threads and highly rely on

caches to improve the performance of memory access. Increasing the program's locality is beneficial to the performance on CPUs.

To address the problem above, we need to optimize the computation kernels for each device. Specifically, when porting a GPU-only program to integrated architectures, we explore the following optimization strategies for CPUs: (1) removing local memory usage. (2) increasing the program's locality, such as tuning the data memory layout and tiling large loops. (3) adjusting the degree of parallelism according to the number of processor cores. When porting a CPU-only program, we explore the following strategies for GPUs: (1) storing the frequently accessed data in the local memory. (2) tuning the memory access patterns. (3) adjusting the degree of program parallelism to exert the computation power of GPU devices, such as tuning the workgroup size. Moreover, we also do some common optimizations for both devices, such as common subexpression elimination. In the future, we can explore more optimizations on integrated architectures.

```
//(a) kernel for GPU (Original Version)
__kernel void pgain_kernel(..., __global float *coord_d, __local float *coord_s, ...){
  ...
  if(local_id == 0){
    ...
    for(int i=0; i<dim; i++)
      coord_s[i] = coord_d[i*num + x];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  ...
  for(int i=0; i<dim; i++)
    x_cost += ( coord_d[(i*num)+thread_id] -coord_s[i]) *
              ( coord_d[(i*num)+thread_id] -coord_s[i]);
  ....
}
-----------------------------------------------------------------
//(b) kernel for CPU
__kernel void pgain_kernel_forcpu (..., __global float *coord_d_cpu, ...){
  ....
  for(int i=0; i<dim; i++)
    x_cost += ( coord_d_cpu [i+thread_id*dim] -coord_d_cpu [i + x*dim]) *
              ( coord_d_cpu [i+thread_id*dim] -coord_d_cpu [i + x*dim]);
  ...
}
```

Fig. 5. Removing the local memory usage and tuning the data memory layout for SC. (a) the GPU kernel. (b) the optimized kernel for CPUs.

We perform the above optimization strategies for all the evaluated programs, Table 2 only lists the main programs benefiting largely from these optimizations. Although some of these optimizations have been explored on discrete GPUs, we analyze their benefits on integrated architectures in this paper. Results show that some programs can get significant performance improvement with these optimizations. We give two examples to demonstrate our optimization strategies for improving the co-running performance.

SC is a program in the Rodinia benchmark. Figure 5(a) is the original GPU kernel and Figure 5(b) is the optimized kernel for CPUs. We do the following optimizations. First, we remove the local memory usage, *coord_s*, which can cause redundant memory access on CPUs. In Figure 5(b), the kernel directly accesses the global memory and does not use the keyword *__local* explicitly. Second, we tune the data memory layout. We create a new buffer *coord_d_cpu*, which is the transposed array of *coord_d*. After transformation, the memory access becomes continuous and the program's locality is largely improved on CPUs. In contrast, different threads within the same *wavefront* on GPUs can achieve the best performance when they access adjacent memory locations, so we still keep the original computation kernel on GPUs. With this method, both CPUs and GPUs access the input memory buffer with good locality.

The tuned results of SC are shown in Figure 4(i). In Figure 4(i), the line labeled with *Only data partition* represents the performance of only performing data partition, while the line labeled with *Tuned* shows the optimized performance. After optimization, SC becomes a co-run friendly program, where the execution time at ratio 0 is 12227.3 ms and the execution time at ratio 100% is 12553.4 ms. The best performance is 10033.8 ms where the workload partition ratio is 50%.

```
//(a) kernel for GPU (Original Version)
__kernel void Fan2(...){
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    if (globalIdx < size-1-t && globalIdy < size-t) {
        a_dev[size*(globalIdx+1+t)+(globalIdy+t)] -=
                    m_dev[size*(globalIdx+1+t)+t] * a_dev[size*t+(globalIdy+t)];
                    ...
    } ...
}
//(b) kernel for CPU
__kernel void Fan2(...){
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    int globalIdysize = get_global_size(1);
    int globalIdystart = globalIdy/globalIdysize*size;
    int globalIdyend = (globalIdy+1)/globalIdysize*size;
    for(globalIdy = globalIdystart; globalIdy<globalIdyend ;globalIdy++)
        if (globalIdx < size-1-t && globalIdy < size-t) {
            a_dev[size*(globalIdx+1+t)+(globalIdy+t)] -=
                    m_dev[size*(globalIdx+1+t)+t] *a_dev[size*t+(globalIdy+t)];
                    ...
    } ...
}
```

Locality improved

Fig. 6. Adjusting the degree of parallelism to improve the locality for GE. (a) the GPU kernel. (b) the optimized kernel for CPUs.

We also present how to adjust the degree of parallelism for GE to improve the program's locality in Figure 6. Figure 6(a) is the original GPU kernel. The original GPU kernel has two dimensions and we change one dimension into a loop in the CPU kernel. As shown in Figure 6(b), the memory access for *a_dev* within the same thread becomes continuous after transformation. The performance results of GE after optimization is shown in Figure 4(c) (labeled with "Tuned"). Note that GE becomes a CPU-friendly program after tuning.

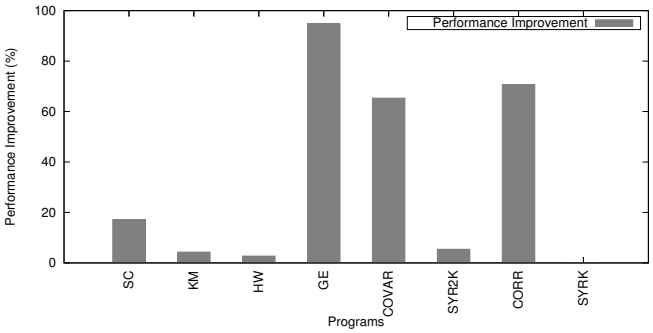


Fig. 7. Co-running performance improvement after performing all the optimizing strategies.

Figure 7 shows the final co-running performance improvement applying the above optimization strategies. We only list the programs with co-running performance improvement. The average performance improvement for these programs is 8.3%. We use the geometric mean to calculate the average performance improvement to mitigate the influence of a few extreme cases in the testing results.

4.3.2 Memory Bandwidth Limitation

Finding (2): Co-run friendly programs usually have low memory bandwidth utilization. Memory bandwidth limitation is currently a main factor causing the co-running performance degradation. There will be more co-run friendly programs if we can remove the memory bandwidth limitation on the integrated architecture.

Memory bandwidth is another important factor that may affect the co-running performance. Although previous researchers observed the phenomenon that the performance of many programs is bound by limited memory bandwidth [20], we have made a similar observation on integrated architectures. CodeXL provides the metrics of *FetchSize* and *WriteSize* to quantify the total numbers of kilobytes fetched from and written to the global memory, respectively. We define *bandwidth utilization* for the global memory to present the bandwidth utilization of the whole system. *time* denotes the total time of executing a kernel in millisecond.

$$bandwidth\ utilization = \frac{FetchSize + WriteSize}{time} \quad (5)$$

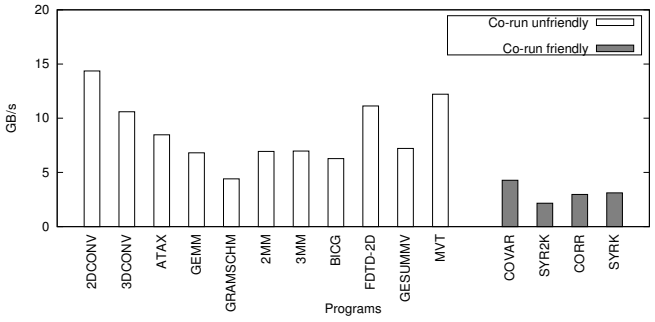


Fig. 8. The relationship between the memory bandwidth utilization and co-running types.

In Figure 8, we list the memory bandwidth utilization and the program classification types. We find that the co-run friendly programs usually have low memory bandwidth requirements while the co-run unfriendly programs behave in an opposite way. Therefore, memory bandwidth has a great relationship with the co-running performance. In order to quantify the effect of memory bandwidth on the co-running performance, we present a theoretical performance improvement when we remove the memory bandwidth limitation as follows.

We use the following method to remove memory bandwidth contention and obtain a theoretical co-running performance improvement. First, we add a barrier operation between the executions of GPUs and CPUs so that both GPUs and CPUs process their workloads separately and the program can utilize the whole memory bandwidth. Second, we calculate the execution time spent on CPUs, denoted as $time_{CPU}$, and the execution time spent on GPUs, denoted as $time_{GPU}$. We use the maximum value of $time_{CPU}$ and $time_{GPU}$ as the theoretical co-running time without the memory bandwidth contention. Third, we calculate the theoretical co-running time for each workload partition ratio, and choose the optimal co-running time from them. Finally, we use the following formula to calculate the theoretical performance improvement. $time_{single\_dev}$ denotes the best execution time on CPUs or GPUs. We replace the $time_{co\_run}$ with the optimal theoretical co-running time and the optimal tested co-running time to calculate the theoretical and actual co-running improvement respectively.

$$PerformanceImprovement = \frac{time_{single\_dev} - time_{co\_run}}{time_{single\_dev}} \quad (6)$$

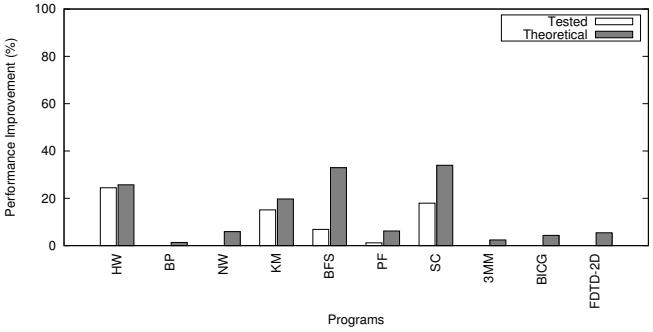


Fig. 9. Theoretical performance improvement for the co-running programs after partially removing the memory bandwidth limitation compared with actual performance improvement.

Figure 9 shows the theoretical and actual performance improvement for co-running. We can get two interesting findings from these data. First, there will be more co-run friendly programs (BP, NW, BFS, PF, 3MM, BICG, and FDTD-2D) if we remove the memory bandwidth limitation (Note that our method cannot completely remove current memory bandwidth limitation while our method only lets the CPUs or GPUs utilize all of the available memory bandwidth). Second, the co-run friendly programs can also benefit from removing the memory bandwidth limitation (HW, KM, and SC). The maximum performance improvement of co-running can achieve about 40%.

4.3.3 Overlap of Co-Running Programs

Finding (3): Most programs on the integrated architecture have high overlap ratios between GPUs and CPUs. Co-run friendly programs always have long kernel execution time and high overlap ratios. Programs with low kernel execution time tend to have low overlap ratios.

Since the GPU command queue and CPU command queue are separate queues and the computation kernels are launched asynchronously on the integrated architecture, it is necessary to quantify the overlap ratios between GPU and CPU kernels. The interleaved state of computation kernels between CPUs and GPUs can affect the co-running performance if the computation kernels cannot be

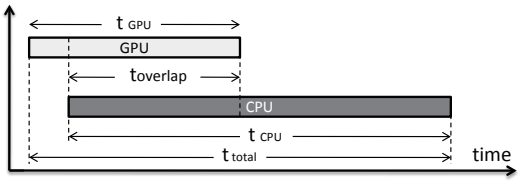


Fig. 10. The execution model on the integrated architecture.

fully overlapped between two queues. Figure 10 shows the execution model on the integrated architecture. t_{GPU} denotes the execution time of the GPU and t_{CPU} denotes the execution time of the CPU. $t_{overlap}$ denotes the overlapping time interval between GPUs and CPUs. t_{total} denotes the total execution time. We use performance events on the integrated architecture to measure the start and end time of computation kernels and then calculate these values. We define $OverlapRatio$ as follows.

$$OverlapRatio = \frac{t_{overlap}}{min(t_{GPU}, t_{CPU})} \quad (7)$$

If $OverlapRatio$ is close to 100%, it means that the overlap issue is not serious on the integrated architecture.

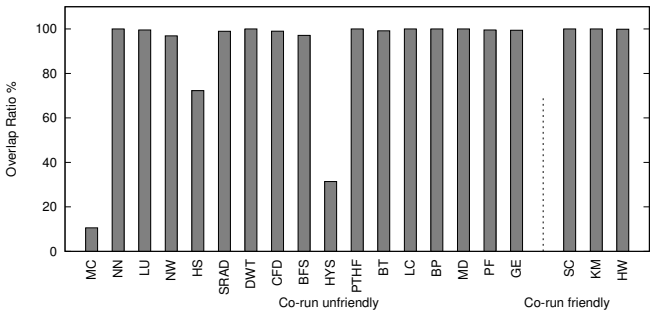


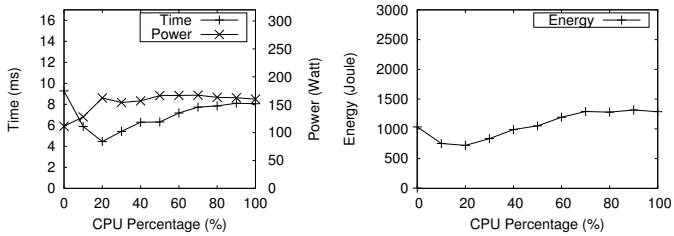
Fig. 11. The overlap ratios between GPU and CPU kernels for the co-running programs.

We list the overlap ratios in Figure 11. We use 50% workload partition ratio in our analysis. For most programs, $OverlapRatio$ is nearly 100% and only 3 programs have low overlap ratios. We analyze the programs with low overlap ratios and find that they also have short kernel execution time (MC is 0.01ms, HS is 0.28ms, HYS is 12.4ms). If the kernel execution time is too short, the co-running overhead is relatively obvious, which can further affect the co-running performance. For co-run friendly programs, they always have very high overlap ratios.

4.4 Power and Energy Characteristics

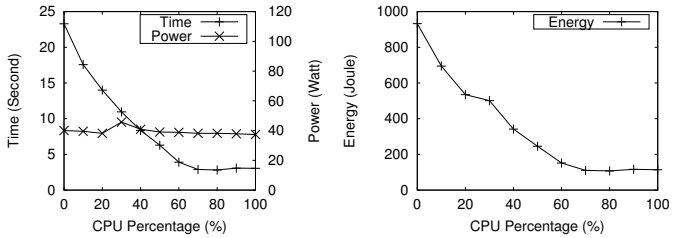
Finding (4): Not all integrated architectures are more energy-efficient than discrete architectures. Although integrated architectures normally have relatively low real-time power, they also have low computation performance compared to discrete architectures. For some programs, they still consume more energy than discrete architectures.

Power is an important metric for evaluating integrated architectures. Both AMD and Intel have special designs for optimizing the power on integrated architectures. We



(a) Execution time and real-time power (b) Accumulated energy

Fig. 12. The real-time power and accumulated energy for COVAR on the AMD integrated architecture.



(a) Execution time and real-time power (b) Accumulated energy

Fig. 13. The real-time power and accumulated energy for COVAR on the Intel integrated architecture.

analyze both the power and energy consumption on AMD and Intel integrated architectures. We use an external power meter WT210 to measure the real-time power and calculate the total energy consumption. The measured power is for the whole machine.

On the AMD A10-7850K platform, the power at idle state is 56W. We use COVAR of Polybench as an example to demonstrate the variance of power and energy consumption with the workload partition ratios. Figure 12(a) shows the execution time and real-time power for different workload partition ratios, and Figure 12(b) shows the energy consumption for different workload partition ratios. COVAR is a co-run friendly program. Results show that the power and energy consumption changes with the CPU/GPU workload, but the change rate is insignificant. The energy curve is mainly determined by the total execution time.

On the Intel i7-4770R platform, the power at idle state is 17W, which is very power efficient. The results of power and energy consumption on i7-4770R are similar to the results on A10-7850K. The energy curve is also determined by the program execution time. The co-running result of COVAR is shown in Figure 13. On the Intel platform, the power variance of different workload partition ratios is also insignificant. On the Intel platform, although the peak performance of the CPU is lower than that of the GPU, results show that COVAR shows much better performance on the CPU device than the GPU. The more workload is allocated to the CPU, the less energy is consumed on the Intel platform. Moreover, as mentioned before, the cache coherence between CPUs and GPUs cannot be guaranteed, so only ATAX, CORR, COVAR, and GESUMMV can be correctly co-run. For BICG and MVT of Polybench, since the problem sizes are too large for the Intel platform, we do

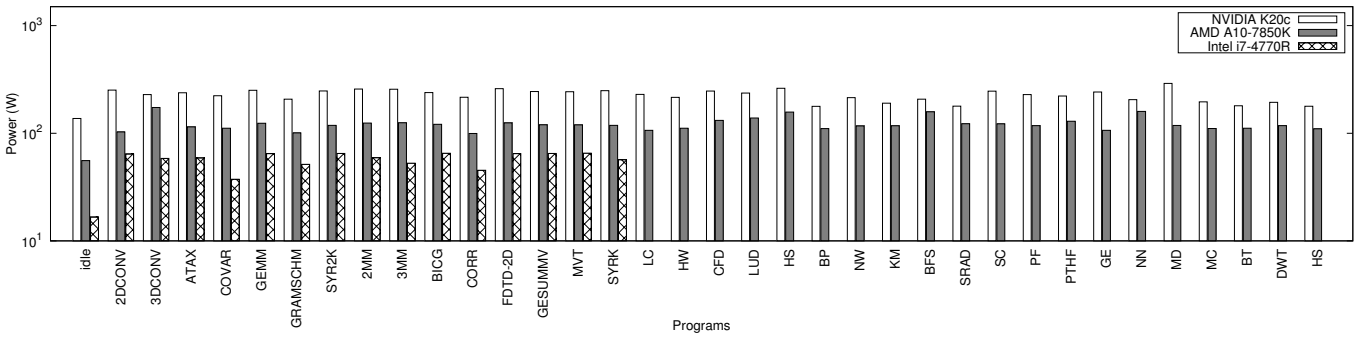


Fig. 14. Power comparison on AMD integrated architecture, Intel integrated architecture, and NVIDIA discrete architecture.

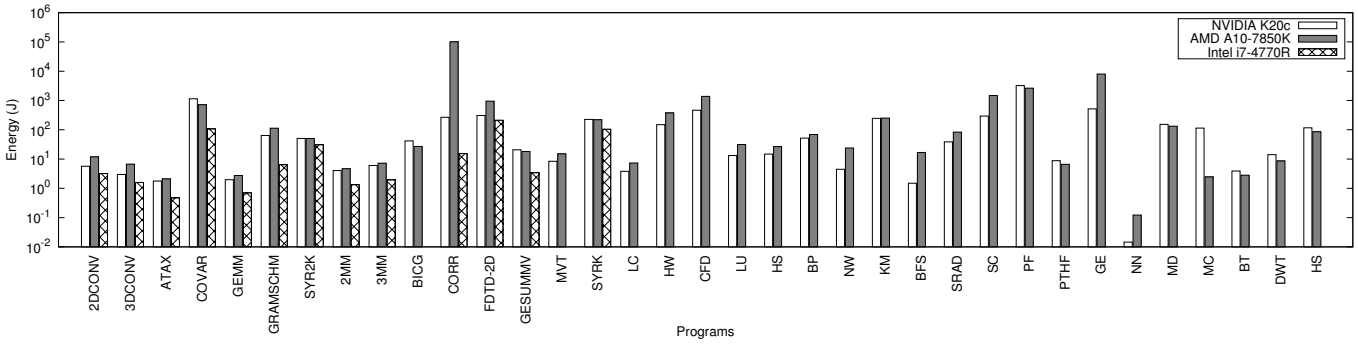


Fig. 15. Accumulated energy consumption on AMD integrated architecture, Intel integrated architecture, and NVIDIA discrete architecture.

not execute them on the Intel platform.

Due to space limitation, we only list the results of the Rodinia and Polybench. We list the minimum power for the AMD and Intel platforms and also compare them with a discrete GPU, NVIDIA K20c in Figure 14. On the integrated architectures, we record the minimum power state from different workload partition ratios. The power of K20c is much higher than the power on integrated architectures. The Intel integrated architecture has the lowest real-time power. The average power for evaluated programs on A10-7850K, i7-4770R, and K20c are 121.8W, 58.9W, and 227.2W respectively.

Accumulated energy consumption is a main metric for end users. We present the energy consumption results of the AMD and Intel platforms and also list the energy consumption results of NVIDIA K20c in Figure 15. The Intel platform consumes the least energy to complete all the programs. For the K20c, though it has the highest real-time power, as it has the shortest execution time, the total energy consumption is less than that of the AMD platform. For the AMD platform, though it has much lower power than that of K20c, the execution time is much longer than that of K20c. Therefore, it consumes the most energy on average.

5 Prediction Models

Since not all the programs can benefit from integrated architectures, it is necessary to identify which program is co-run friendly before porting it. To address this problem, we adopt a black-box machine learning method to build a decision-tree-based performance prediction model. With this model, we can predict the co-running type for a given GPU-only or CPU-only program. Furthermore, for a co-run friendly program, we present a profiling-based prediction model to help users to estimate the optimal partition ratio. Our method is platform-independent. To perform the prediction for a given platform, we need to first collect a training set on it and obtain a performance prediction model, and then use the prediction model to predict other programs on the same platform. For other platform, we need to collect the training set again on that platform to obtain a new prediction model. We only validate the prediction model on the AMD platform in the experiment because the Intel platform has a cache coherence issue. Our model cannot precisely predict the absolute performance, but can predict whether a program is co-run friendly or not. A more general performance prediction model on CPU/GPU integrated platforms will be studied in our future work.

5.1 Predicting Program Co-running Types

In this work, we use the C4.5 algorithm [16] to automatically build a decision-tree-based prediction model, as C4.5 is a popular and mature technique, which has been widely used in the domain of data mining [21]. However, our method is not limited to a particular machine learning model. Like other classic machine learning methods, C4.5 builds a prediction model from a set of training data. Specifically, C4.5 builds a decision tree in a top-down manner. For each node of the decision tree, C4.5 determines the most effective predictor from the input training set, using the concept of information entropy. A metric called gain ratio is calculated to be used as specific splitting criterion. A predictor with the highest value of gain ratio is chosen as a splitting node

at each step. The C4.5 algorithm then recurs on a smaller subset until a predefined criteria is met.

5.1.1 Training Data Selection

We use both Rodinia and Parboil benchmarks as the training set. The Rodinia benchmarks cover a wide range of computation patterns and the Parboil benchmarks include a set of throughput computing applications. We use 27 programs from both benchmarks, including 20 programs from the Rodinia and 7 programs from the Parboil. We choose these programs as the training set because they cover the main parallel patterns and can make the generated prediction model more reliable.

We select a variety of predictors based on the aforementioned workload characteristics analysis. There are 47 predictors for each program. Some predictors can be directly obtained from performance profiling tools, such as *FetchSize* and *WriteSize*, while others can be obtained through simple calculation, such as *Read Bandwidth* (*FetchSize*/*time*). It should be noted that all these predictors can be acquired from CPU-only or GPU-only executions. We classify these predictors into 3 categories. (1) *GPU-side predictors:* the predictors are acquired from the GPU device. For example, *WriteSize* denotes the total data written to global memory. (2) *CPU-side predictors:* the predictors are acquired from the CPU device, such as *IPC*. (3) *Relative predictors:* the predictors are calculated using both GPU and CPU metrics, such as $TimeRatio_{CPU/GPU}$ (the execution time ratio between two devices), which reflects relative performance difference between two devices.

5.1.2 Prediction Model Analysis

TABLE 3
Top ten predictors calculated by C4.5 while selecting the root node of the decision tree.

Predictor	Type	Gain Ratio	Descriptions
$\text{TimeRatio}_{C/G}$	Relative	0.68	$\text{TimeRatio}_{CPU/GPU}$, tcpu_100/tcpu_0
WriteSize	GPU	0.38	The total kilobytes written to memory
TNuminWG	GPU	0.37	The number of threads in a workgroup
VALUBusy	GPU	0.32	The GPUTime percentage when VALUs are processed
Bandwidth	GPU	0.31	The memory bandwidth for read and write
ThreadNum	GPU	0.26	The total number of threads
AvgKTime	GPU	0.26	The average kernel time in GPU devices
LocalUse	GPU	0.25	Whether the local memory is used
IPC	CPU	0.23	Instructions per cycle in CPU devices
SALUBusy	GPU	0.19	The GPUTime percentage when SALUs are processed

During the construction of the decision tree model, the C4.5 algorithm will calculate the gain ratio for each predictor and select a predictor with the highest value of the gain ratio at each step. It then generates a ranking list for the predictors based on the gain ratio values. Table 3 shows the top ten predictors and their descriptions when C4.5 selects the root node of the decision tree. It shows that all these predictors have a strong relationship with the factors for performance degradation described in Section 4.3. For example, the predictor of $LocalUse$ indicates whether the local memory is used in the program. *WriteSize* and *Bandwidth* describe the program memory usage. The predictors of *TNuminWG* and *ThreadNum* present the degree of parallelism in the program. $TimeRatio_{CPU/GPU}$ has the highest value of gain ratio, which reflects the relative performance difference between two devices.

The decision tree built by C4.5 is shown in Figure 16. The decision tree includes both internal nodes (*Nodes*) and leaf nodes (*Terminals*). For the internal nodes, the predictor with the highest gain ratio is selected, and the specific splitting value is also automatically calculated by C4.5 to guide decision making. For the leaf nodes, the final predicted types are listed. We find that not all the leaf nodes have 100% accuracy for the training set, such as *Terminal 1*. The exception program in *Terminal 1* is HW. In *Node 3* of the decision tree, *FetchSize* is selected as the predictor. This is because the gain ratio of *FetchSize* is 0.736 while the metric of $TimeRatio_{CPU/GPU}$ decreases to 0.195 in this node. This also indicates that the memory access is an important factor for current integrated architectures.

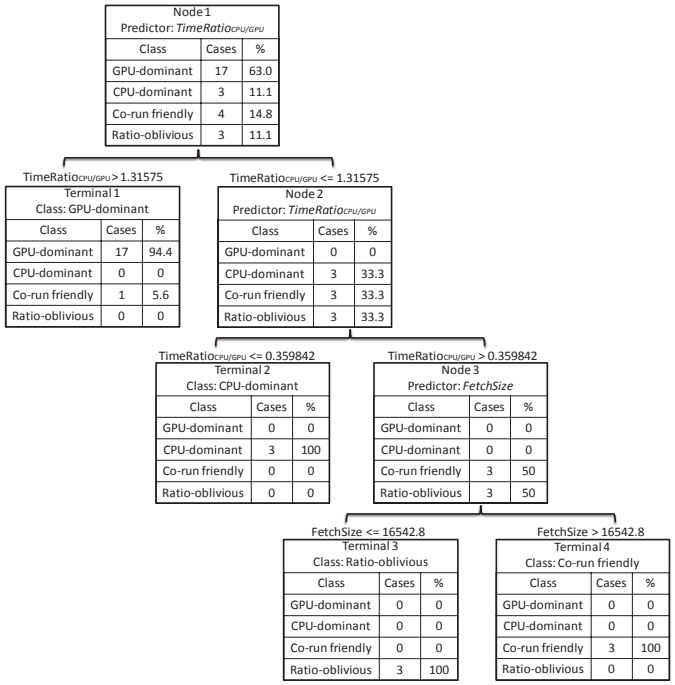


Fig. 16. The decision tree built with C4.5 algorithm.

5.1.3 Validation of Prediction Model

TABLE 4
Prediction results of co-running types for Polybench with our model. The check mark means the number of correct predictions.

Categories	(a)	(b)	(c)	(d)
(a) GPU-dominant	8 (✓)			
(b) CPU-dominant		3 (✓)		
(c) Co-run friendly	1 (×)		3 (✓)	
(d) Ratio-oblivious				

We use Polybench to demonstrate the accuracy of our decision-tree-based prediction model. Polybench [12] is also a popular heterogeneous computing benchmark suite and includes OpenCL implementations. It covers three main co-running program types.

We list the prediction results in Table 4. We can correctly predict 14 programs out of 15 programs in Polybench. For each program, we also list the values of those performance predictors used in the decision tree, the input data size, and the main computing kernels in Table 5. Experimental results show that the value of $TimeRatio_{CPU/GPU}$ is very close to

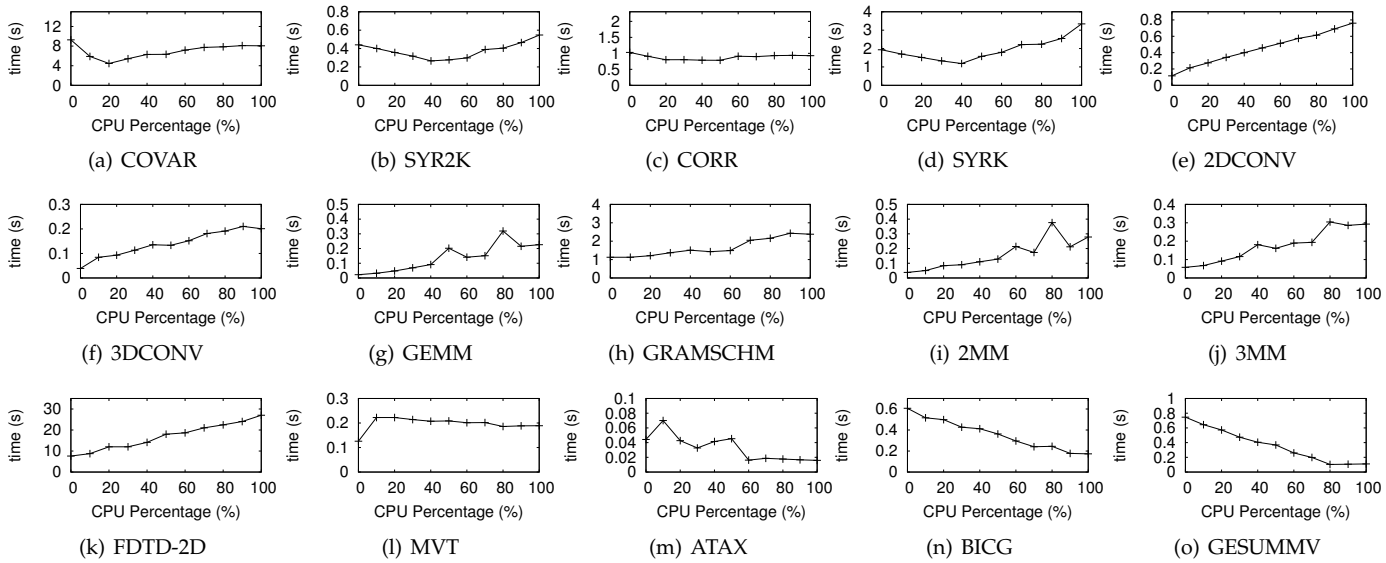


Fig. 17. Co-running results for the Polybench benchmarks for different partition ratios. (a, b, c, d) are co-run friendly programs. (e, f, g, h, i, j, k, l) are GPU-dominant programs, and (m, n, o) are CPU-dominant programs.

TABLE 5
The values of performance predictors used in our decision tree model and the detailed prediction results for the Polybench benchmarks.

Programs	Kernel Name	Input Size	$\text{TimeRatio}_{\text{CPU/GPU}}$	FetchSize	Co-running Types	Predicted Types
2DCONV	Convolution2D_kernel	(8192, 8192)	6.5685	306318.9	GPU-dominant	GPU-dominant
3DCONV	Convolution3D_kernel	(256, 256, 256)	5.1853	769.2	GPU-dominant	GPU-dominant
ATAX	atax_kernel1	(4096, 4096)	0.3593	268020.8	CPU-dominant	CPU-dominant
COVAR	covar_kernel	(2048, 2048)	0.8686	32064573.5	Co-run friendly	Co-run friendly
GEMM	gemm	(512, 512, 512)	10.1808	65303.5	GPU-dominant	GPU-dominant
GRAMSCHM	gramschmidt_kernel3	(512, 512)	2.1118	4160.9	GPU-dominant	GPU-dominant
SYR2K	syr2k_kernel	(1024, 1024)	1.2429	1237085.9	Co-run friendly	Co-run friendly
2MM	mm2_kernel1	(512, 512, 512, 512)	7.3806	64745.8	GPU-dominant	GPU-dominant
3MM	mm3_kernel1	(512, 512, 512, 512, 512)	5.0579	64713.6	GPU-dominant	GPU-dominant
BICG	bicgKernel1	(16384, 16384)	0.2853	3072194.9	CPU-dominant	CPU-dominant
CORR	corr_kernel	(1024, 1024)	0.9008	3384485.7	Co-run friendly	Co-run friendly
FDTD-2D	fdtd_kernel3	(500, 2048, 2048)	3.5508	49183.4	GPU-dominant	GPU-dominant
GESUMMV	gesummv_kernel	(8192)	0.1480	5478489.8	CPU-dominant	CPU-dominant
MVT	mvt_kernel1	(8192)	1.5029	908085.2	GPU-dominant	GPU-dominant
SYRK	syrk_kernel	(2048, 2048)	1.7281	4724026.4	Co-run friendly	GPU-dominant

1 in COVAR, SYR2K, and CORR, which are typical co-run friendly programs.

There is one exception program with our prediction model, SYRK. The inaccurate prediction result is because the prediction accuracy is mainly subject to the input training set. Although we have selected a variety of programs as the training set, there is still some performance characteristic space that the training set does not cover.

Figure 17 lists the execution time for the programs of Polybench for different partition ratios. Experiments show that the co-running results of Polybench are also diverse. In summary, there are 4 co-run friendly programs, 8 GPU-dominant programs, and 3 CPU-dominant programs.

5.2 Prediction of Workload Partition Ratio

For the co-run friendly programs, we further propose a simple profiling-based prediction model to help application developers to determine the optimal workload partition ratio, rather than trying different partition ratios. The prediction method for workload partition ratio is a key supplement of the decision-tree model, which can help users to automatically determine an optimal partition point for a co-run friendly program. Our method only uses two basic metrics measured on GPU and CPU devices. First, we measure the execution time, $time_{GPU}$, when we assign 100% workload to GPUs, and the execution time $time_{CPU}$ when we assign 100% workload to CPUs. The reciprocal of the above execution times represents the compute capacity for each device. After obtaining $time_{GPU}$ and $time_{CPU}$, we partition the entire workload according to the following equation. $Partition_{CPU}$ workload proportions are assigned to CPUs, while the remaining $(1 - Partition_{CPU})$ workload is assigned to GPUs. $Partition_{CPU}$ is calculated as follows.

$$Partition_{CPU} = \frac{time_{GPU}}{time_{GPU} + time_{CPU}} \quad (8)$$

We list the prediction results for co-run friendly programs in Table 6. In these programs, HW, KM, and SC are from the Rodinia benchmark. COVAR, SYR2K, CORR are from the Polybench, and SPMV is from the Parboil benchmark. The column of *Predicted co-run* presents the execution time with the predicted partition ratio, and the column of *Optimal co-run* presents the execution time of

the optimal partition ratio acquired with exhaustive tests. *Single dev* means the optimal performance on a single device (CPU or GPU). Results show that our prediction model almost achieves the optimal performance for most programs. In summary, the co-running programs acquired with our model outperform the original CPU-only and GPU-only programs by 34.5% and 20.9% respectively. Our method can achieve 87.7% of the optimal partition performance.

TABLE 6
Performance comparison between co-running with our predicted partition ratio and that with the optimal partition ratio. The execution times for CPU-only and GPU-only versions are also listed (in second).

Name	Predicted co-run	Optimal co-run	CPU only	GPU only	single dev	CPU improvement	GPU improvement
HW	2.61	2.55	8.67	3.38	3.38	69.9%	22.7%
KM	1.94	1.81	2.76	2.13	2.13	29.7%	8.9%
SC	10.03	10.03	12.23	12.55	12.23	18.0%	18.0%
COVAR	6.62	4.46	8.06	9.28	8.06	17.8%	28.6%
SYR2K	0.27	0.27	0.55	0.44	0.44	50.5%	38.5%
CORR	0.82	0.78	0.93	1.03	0.93	12.0%	20.7%
SYRK	1.23	1.19	3.34	1.93	1.93	63.1%	36.2%
SPMV	0.17	0.13	0.20	0.16	0.16	15.0%	-6.3%
avg						34.5%	20.9%

6 Related Work

There are several studies focusing on integrated architectures. Spafford et al. [22] evaluated the trade-offs in APU. Zhu et al. [9] studied co-running performance degradation on integrated architectures. However, their work focused on co-running different programs on such architectures, which is different from ours. Daga et al. [23] characterized the efficacy of the AMD APU. Lee et al. [24] provided a full performance characterization of the Llano APU. Barik et al. [25] mapped applications to the GPUs on Intel integrated architectures. Kaleem et al. [26] proposed adaptive scheduling methods for integrated architectures. Our study is mainly focusing on a wide range of co-running single applications and obtains some key performance characteristics on integrated architectures. Part of this work is published in a conference paper [27], which only provided the Rodinia co-running results. We have added experiments on more platforms and provide workload characteristic analysis, prediction model, and power analysis in this paper.

Some researchers recently tried to apply integrated architectures to some applications. Doerksen et al. [28] optimized 0-1 knapsack and Gaussian Elimination. Daga et al. [29] accelerated B+ Tree. Hetherington et al. [30] studied Key-Value Store. Gu et al. [31] implemented a deep neural network. However, these studies only used the GPUs of the integrated architectures to process computing kernels, which is different from ours. Researchers also tried to find suitable applications to use both GPUs and CPUs together. Chen et al. [7] ported MapReduce programs. Delorme et al. [5] co-ran Radix Sort. Zhang et al. [32] developed portable query processing across heterogeneous processors. He et al. [3], [4] employed an integrated architecture to optimize Hash Join. Moreover, researchers used CPUs and GPUs to perform different tasks. Eberhart et al. [8] studied stencil computation and used CPUs to process the diverging parts. Daga et al. [6] performed Top-Down algorithm on CPUs while Bottom-Up algorithm on GPUs for BFS.

Researchers have also studied the co-processing relationship between GPUs and CPUs on discrete architectures. Grewe et al. [33] proposed a portable partitioning scheme on a discrete heterogeneous CPU-GPU platform. O'Boyle et al. [34] developed an efficient compiler to generate OpenCL code from OpenMP code. Grewe also presented a detailed task partitioning method in a previous study [35]. As the reliability of microprocessors [36], [37], [38], [39] has been a concerning issue in many applications, the system reliability on integrated architecture is also needed.

7 Conclusion and Future Work

In this paper, we port 42 heterogeneous programs on integrated architectures for co-running. We perform a series of workload characterization analysis to understand the co-running behaviors. We find that architecture differences between GPUs and CPUs and limited shared memory bandwidth are two main factors affecting current co-running performance. Based on our workload characteristic analysis, we build a decision-tree-based performance prediction model to help users predict the co-running type before porting the program. Results show that our model achieves very high prediction accuracy. For a co-run friendly program, our model can further estimate the optimal workload partition ratio between GPUs and CPUs and results show that our model can achieve 87.7% of the optimal performance regarding to the best partition. This study starts with some popular benchmarks such as the Rodinia and Parboil benchmark suites, and studies their computation and memory behavior on integrated architectures. This paper focuses on the data parallelism analysis. We will study fine-grained co-running optimizations in our future work.

8 ACKNOWLEDGE

The authors sincerely thank the anonymous reviewers for their valuable comments and suggestions. This work has been partly supported by the Chinese 863 project 2012AA010901, NSFC project 61472201, Tsinghua University Initiative Scientific Research Program, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Huawei Innovation Research Program in China, and a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore. Jidong Zhai is the corresponding author.

References

[1] D. Foley, M. Steinman, A. Branover, G. Smaus, A. Asaro, S. Punyamurtula, and L. Bajic, "AMD's 'Llano'Fusion APU," in *Hot Chips*, 2011.
[2] "The Compute Architecture of Intel Processor Graphics Gen7.5," https://software.intel.com.
[3] J. He, M. Lu, and B. He, "Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture," *VLDB'13*, vol. 6, no. 10, pp. 889–900, 2013.
[4] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *VLDB'14*, vol. 8, no. 4, pp. 329–340, 2014.
[5] M. C. Delorme, T. S. Abdelrahman, and C. Zhao, "Parallel Radix Sort on the AMD Fusion Accelerated Processing Unit," in *ICPP'13*. IEEE, 2013, pp. 339–348.
[6] M. Daga, M. Nutter, and M. Meswani, "Efficient breadth-first search on a heterogeneous processor," in *Big Data'14*. IEEE, 2014, pp. 373–382.
[7] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *SC'12*. IEEE Computer Society Press, 2012, pp. 25:1–25:11.

[8] P. Eberhart, I. Said, P. Fortin, and H. Calandra, "Hybrid strategy for stencil computations on the APU," in *International Workshop on High-Performance Stencil Computations*, 2014, pp. 43–49.
[9] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, "Understanding co-run degradations on integrated heterogeneous processors," in *LCPC'14*. Springer, 2014, pp. 82–97.
[10] 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'09*. IEEE, 2009, pp. 44–54.
[11] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.
[12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *InPar'12*. IEEE, 2012, pp. 1–10.
[13] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.
[14] D. Bouvier and B. Sander, "Applying AMDs Kaveri APU for Heterogeneous Computing," in *Hot Chips*, 2014.
[15] "Leave the Graphics Card Out, Beautiful is Built In," http://www.intel.com.
[16] P. Winston, "Learning by building identification trees," *Artificial intelligence*, pp. 423–442, 1992.
[17] "Intel Forums," https://software.intel.com/en-us/comment/1827746#comment-1827746.
[18] "4th Generation Intel Core i7 Processors," http://ark.intel.com.
[19] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann, 2015.
[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[21] J. R. Quinlan, *C4.5: programs for machine learning*. Elsevier, 2014.
[22] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures," in *CF'12*. ACM, 2012, pp. 103–112.
[23] M. Daga, A. M. Aji, and W.-c. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *SAAHPC'11*. IEEE, 2011, pp. 141–149.
[24] K. Lee, H. Lin, and W.-c. Feng, "Performance characterization of data-intensive kernels on AMD Fusion architectures," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 175–184, 2013.
[25] 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'14*. ACM, 2014, pp. 33:33–33:43.
[26] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *PACT'14*. ACM, 2014, pp. 151–162.
[27] 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 *MASCOTS'15*. IEEE, 2015.
[28] M. Doerksen, S. Solomon, and P. Thulasiraman, "Designing APU Oriented Scientific Computing Applications in OpenCL," in *HPCC'11*. IEEE, 2011, pp. 587–592.
[29] M. Daga and M. Nutter, "Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU," in *SCC'12 Proceedings of the 2012 SC Companion*. IEEE, 2012, pp. 240–247.
[30] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *ISPASS'12*. IEEE, 2012, pp. 88–98.
[31] J. Gu, M. Zhu, Z. Zhou, F. Zhang, Z. Lin, Q. Zhang, and M. Breternitz, "Implementation and evaluation of deep neural networks (DNN) on mainstream heterogeneous systems," in *APSYS'14*. ACM, 2014, pp. 12:1–12:7.
[32] S. Zhang, J. He, B. He, and M. Lu, "OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures," *VLDB'13*, vol. 6, no. 12, pp. 1374–1377, 2013.
[33] D. Grewe and M. F. OBoyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Compiler Construction*. Springer, 2011, pp. 286–305.
[34] M. F. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *CGO'13*. IEEE Computer Society, 2013, pp. 1–10.
[35] D. Grewe, "Mapping parallel programs to heterogeneous multi-core systems," 2014.
[36] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *MICRO'13*. ACM, 2013, pp. 457–467.
[37] C. Liu, K. Ouyang, X. Chu, H. Liu, and Y.-W. Leung, "R-memcached: A reliable in-memory cache for big key-value stores," *Tsinghua Science and Technology*, vol. 20, no. 6, pp. 560–573, 2015.
[38] T. Liu, C.-C. Chen, W. Kim, and L. Milor, "Comprehensive reliability and aging analysis on SRAMs within microprocessor systems," *Microelectronics Reliability*, vol. 55, no. 9, pp. 1290–1296, 2015.
[39] D. Zhang, Y. Liu, S. Li, T. Wu, and H. Yang, "Simultaneous accelerator parallelization and point-to-point interconnect insertion for bus-based embedded SoCs," *Tsinghua Science and Technology*, vol. 20, no. 6, pp. 644–660, 2015.

Feng Zhang is a Ph.D candidate in Department of Computer Science and Technology, Tsinghua University. He received the bachelor degree from Xidian University in 2012. His major research interests include high performance computing, heterogeneous computing, 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 assistant 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.

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.

Shuhao Zhang is currently a Ph.D candidate in School of Computing, National University of Singapore, and a research associate at SAP Research&Innovation Singapore. He received the bachelor degree from Nanyang Technological University Singapore in 2014. His major research interests include High Performance Computing, Streaming Data Processing, Parallel and Distributed 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.