To Co-Run, or Not To Co-Run: A Performance Study on Integrated Architectures

Feng Zhang*, Jidong Zhai*, Wenguang Chen*, Bingsheng He† and Shuhao Zhang†
*Department of Computer Science and Technology
Tsinghua University, Beijing, China
Email: feng-zhang12@mails.tsinghua.edu.cn, zhaijidong@tsinghua.edu.cn, cwg@tsinghua.edu.cn
†Nanyang Technological University, Singapore
Email: bshe@ntu.edu.sg, szhang026@e.ntu.edu.sg

***Abstract*—Architecture designers tend to integrate both CPU and GPU on the same chip to deliver energy-efficient designs. To effectively leverage the power of both CPUs and GPUs on integrated architectures, researchers have recently put substantial efforts into co-running a single application on both the CPU and the GPU of such architectures. However, few studies have been performed to analyze a wide range of parallel computation patterns on such architectures. In this paper, we port all programs in Rodinia benchmark suite and co-run these programs on the integrated architecture. We find that co-running results are not always better than running the application on the CPU only or the GPU only. Among the 20 programs, 3 programs can benefit from co-running, 12 programs using GPU only and 2 programs using CPU only achieve the best performance. The remaining 3 programs show no performance preference for different devices. We also characterize the workload and summarize the patterns for the system insights of co-running on integrated architectures.**

I. Introduction

Integrating GPUs with CPUs on the same chip is increasingly common in current processor architectures. In 2011, AMD released its integrated architecture [7], called accelerated processing units (APUs). Subsequently, Intel also released an integrated architecture in the processors of Ivy Bridge and Haswell [1]. 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 PCIe bus in traditional architectures using discrete GPUs.

To effectively leverage the power of both CPUs and GPUs on integrated architectures, researchers have recently focused on co-running the same application on both the CPU and the GPU on such architectures. He et al. [8], [9] employ an integrated architecture to optimize Hash Join, which is an important type of operations in databases. Delorme et al. [6] implement a parallel radix sort using an integrated architecture. Daga et al. [5] show the promising performance of Breadth-First Search using an integrated architecture. Chen et al. [4] accelerate MapReduce on an integrated architecture. Different from the co-run study in our paper, Zhu et al. [10] run different applications simultaneously on both CPU- and GPU-integrated architectures. However, few studies have been performed to analyze a wide range of co-running single applications on such architectures.

In this paper, we explore these problems through adopting a strategy of data partition that can utilize the GPU and CPU resources on integrated architectures. We focus on data-parallel workloads. We have ported Rodinia benchmark suite [3], 20 parallel programs in total, and co-run these programs on an integrated architecture. We rewrite these programs to enable them to co-run on both CPUs and GPUs, and the partition point of CPUs and GPUs can be adjusted in a flexible manner to achieve good load balance. This is a simple yet general approach for implementing co-running, and more optimization through re-designing algorithms are left for future work.

The study has shed light on the future research of co-running the application. More research and optimizations are needed to take advantage of our workload characterization for better performance. We make two main contributions in this work:

- We port the Rodinia benchmark suite. Among the 20 ported programs, 3 programs are co-run-friendly (the program performance of co-running on both CPU and GPU is the best), 12 programs are GPU dominant (the program performance of only running on GPU is the best), 2 programs are CPU dominant (the program performance of only running on CPU is the best), and 3 programs show no performance preference for devices.
- We further characterize the workloads and find that the key indicators are local memory usage, kernel execution time, partition number, and the ratio of computation to memory access. Finally, we provide a summary to assist in choosing devices when using integrated architectures.

The remainder of this paper is organized as follows. Section II reviews the integrated architecture. Section III describes our methodology. Section IV details the experiments and summarizes the characteristics. Section V provides the conclusion.

II. Background

We focus on AMD's integrated architecture, the A-Series APU A10-7850K (codenamed "Kaveri [2]"). We show its structure in Figure 1. The CPU has four cores and each core is referred as a computing unit in OpenCL. The GPU has eight computing units. For APUs, the GPU and CPU are on the same die and share the same memory. The CPU and GPU also share the same memory controller. Therefore, memory bandwidth contention occurs. On the APU, although the CPU

and GPU are on the same chip, they still have different memory bandwidths. The bandwidth provided for GPUs is higher than the bandwidth provided for CPUs.

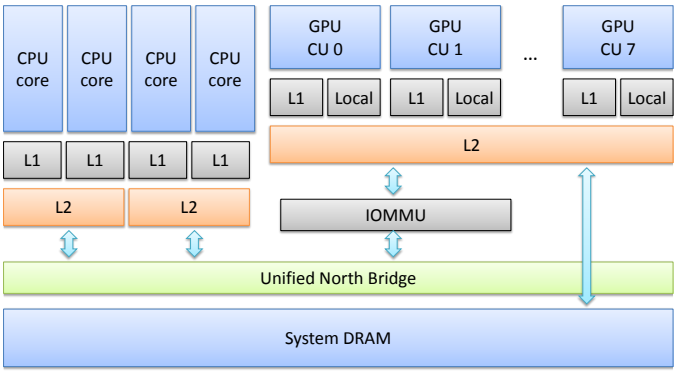


Fig. 1. AMD integrated architecture (AMD A10-7850K)

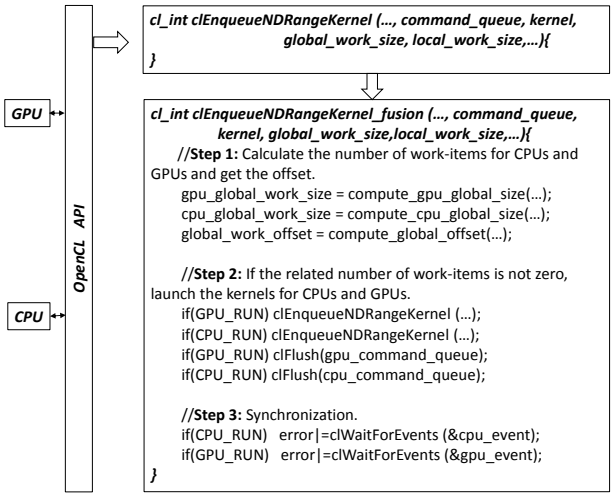


Fig. 2. Replacement for clEnqueueNDRangeKernel.

OpenCL is an open standard computing language and uses a context for managing objects. An OpenCL context can associate with a number of computing devices such as CPUs and GPUs. For each device, we need to create a separate command queue to launch kernels. In OpenCL, each thread is a work item, and a certain number of work items are organized as a workgroup. A work group can run on only one computing unit. The work items within a group can communicate through local memory and global memory, and they are synchronized through barrier operations. Work items from different workgroups cannot communicate directly, and we cannot synchronize them through barrier operations. This flexibility allows us to separate part of workgroups to CPUs if the original program only uses GPUs. Similar ideas are used in [6], [8].

III. Methodology

In the integrated architecture, we distribute workloads to both CPUs and GPUs. Most current benchmarks are designed for CPU-only or GPU-only, and few benchmarks are for co-run. In this study, we focus on data-parallel programs that are written in OpenCL.

A. Implementation

To write an OpenCL program, we first need to create a context, and create command queues within the context. Second, we use the function clEnqueueNDRangeKernel() to enqueue a command to execute a kernel on a device. After the enqueue operation, we use the function clFlush() to guarantee those OpenCL commands have been issued to the associated CPU or GPU. Third, we use the function clWaitForEvents() to wait for executions to complete. After the execution of the kernel, we need to release OpenCL resources.

In our implementation, we first change the code and create two devices and two command queues in a shared context instead of one device and one command queue. Second, we implement a new function, clEnqueueNDRangeKernel_fusion(), instead of the original clEnqueueNDRangeKernel() function. Our pseudo-code is presented in Figure 2. In Step 1, we calculate the work item numbers for CPUs and GPUs, and their starting number. In Step 2, we launch kernels for CPUs and GPUs. In Step 3, we use two queues and two events to perform the synchronization. Third, we release two devices and two queues. We define a global offset to determine the workload proportion for CPUs and GPUs. If the global offset is 0 or 100 %, we only execute the program on one device. Moreover, an OpenCL program can have more than one kernel. We select the main kernel contributing to the most fraction of the execution time for preventing inter-kernel impact of co-run, which facilitates characteristics analysis.

B. Workload Characterizations

We divide the tested programs into two categories: co-run friendly and co-run unfriendly. We then subdivide the co-run unfriendly category into three subcategories: GPU dominant, CPU dominant, and performance similar.

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

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

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

Performance similar programs: Performance similar programs are programs that exhibit no preference for devices using different proportional distributions.

We provide the classification results in Table I. To avoid the effect of random errors, we define the threshold for each category. We define the $CorunIndicator$ and $DeviceChoosingIndicator$ as follows. Programs are considered co-run-friendly only when $CorunIndicator$ is less than 0.85. For co-run unfriendly programs, the programs with $DeviceChoosingIndicator$ less than -0.4 are considered

CPU-dominant, the programs with $DeviceChoosingIndicator$ greater than 0.4 are considered GPU-dominant, while others are performance similar. We choose the boundary values artificially.

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

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

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

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

t_{cpu_i} is the time when the CPU process $i\%$ workloads. $min(t_{cpu\_i}, t_{cpu\_j})$ is the minimum value of t_{cpu_i} and t_{cpu_j}. $min(t_{cpu\_i}, ..., t_{cpu\_j})$ is the minimum value from t_{cpu_i} to t_{cpu_j}. The i and j can be a number from 0 to 100.

IV. Experimental Evaluation

A. Co-run Results

We evaluate the Rodinia benchmark suite [3] according to our classification and conduct experiments on A10-7850K. We vary the CPU workload portion from 0 to 100% at an interval of 10%. We list the input size for each program in Table I. We also list the name of the main kernel for each program. Certain programs have more than one dominant function. We then choose the kernel according to the significance in the program. The Rodinia benchmark suite divides the running time into different stages. For example, Leukocyte provides six stages: device initialization, data allocation, data copy in, kernel running, data copy out, and resource release. The kernel running stage takes more than 90% of the time. Because we co-run CPUs and GPUs to parallelize the kernel, we are only concerned about the kernel stage of the programs. We refer to the kernel stage as the computation stage because certain programs have more than one kernel stage and contain a small amount of data transmission between different kernels. The co-run results are shown in Figure 3.

B. Performance Characteristics Profile

We find that the key indicators are (1) local memory usage, (2) kernel execution time, (3) partition number (where partition number refers to the total thread number divided by the thread number equals the partition number on the selected dimension), and (5) the ratio of computation to memory access. In addition, insufficient thread number also influences co-running results. The characteristics of the four types of programs are given below.

1) Co-run-friendly programs (KM, HW, and GE): Co-run-friendly programs have sufficient kernel execution time and do not use local memory. The partition number of co-run-friendly programs is not excessive. The program performance on GPUs is similar to the performance on CPUs.

2) GPU-dominant programs (PF, HS, BP, LC, SRAD, SC, CFD, PTHF, MD, BT, DWT, and LU): GPU-dominant programs usually have well-structured parallelism, which can fully utilize GPU performance. Moreover, if the kernel time is short or the partition number is high, which gives the CPU limited performance space, the program is GPU-dominant. Local memory is friendlier to GPUs than to CPUs; therefore, programs that use local memory intensively are likely to be GPU dominant.

3) CPU-dominant programs (NN, and MC): If programs have low vector compute to memory ratios or the average kernel time is low, the programs will run poorly on GPUs. These programs will be CPU dominant.

4) Performance similar programs (NW, BFS, and HS): For performance similar programs, programs running on GPUs perform well but do not represent the best configuration. This type of program typically has substantial partition numbers or inadequate threads in a workgroup.

As an example, we show the profiling results for the KM, which is a co-run friendly program. The average kernel time is 24 ms that is sufficient and it does not use local memory. The partition number is 1930 that is considered reasonable. Moreover, the execution time for CPU only and GPU only are 2.1 sec and 2.8 sec, which are similar.

V. Conclusion

In this paper, we analyze the characteristics of co-running a single application on both the CPU and the GPU of an integrated architecture. On AMD's APU A10-7850K, we demonstrate that 1) co-running a single application may not always deliver a better performance than running it on a single processor due to memory contention, 2) careful design and optimizations are required for future research of co-running a single application on the integrated architecture.

VI. Acknowledgement

This work is supported by the National High Technology Research and Development Program of China (2012AA010901), Natural Science Foundation of China project (61472201), and a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

References

[1] 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.

[2] D. Bouvier and B. Sander. Applying AMDs Kaveri APU for Heterogeneous Computing. In *Hot Chips*, 2014.

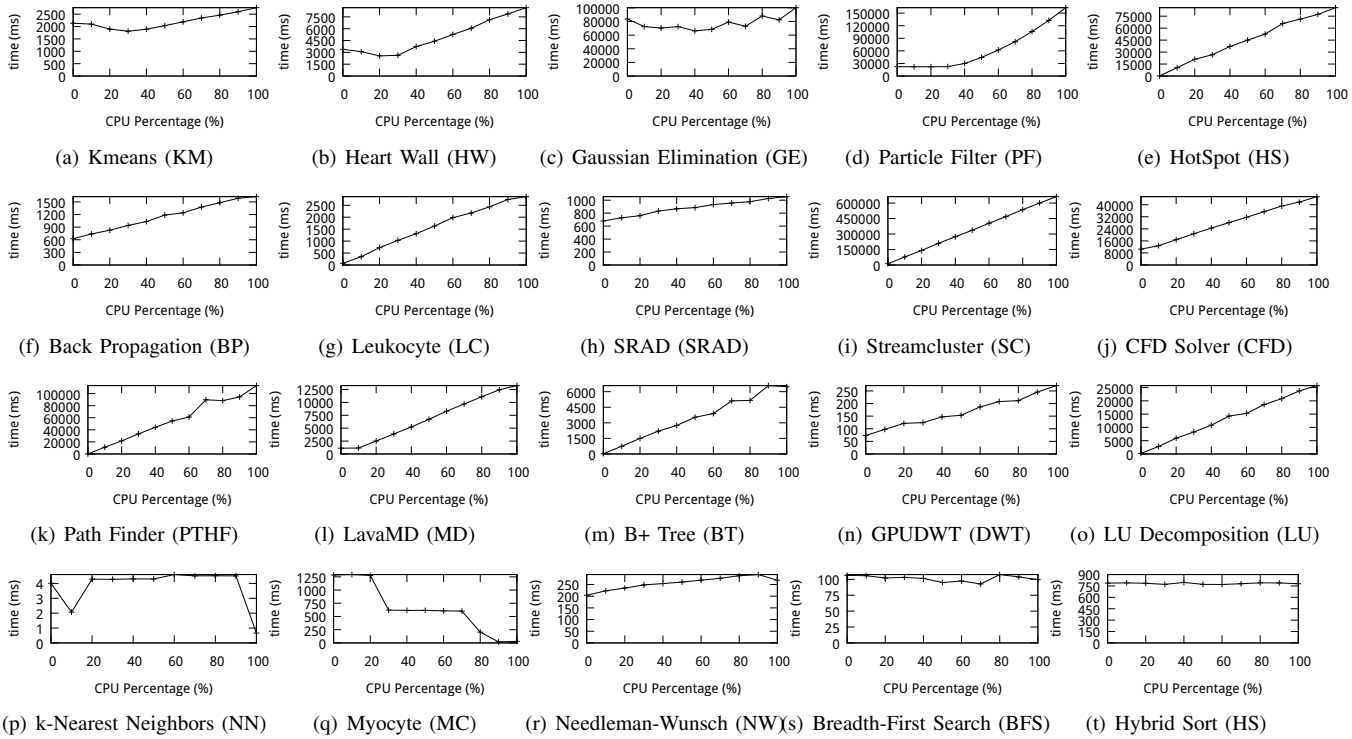
[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[4] 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.

[5] 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.

TABLE I
EXPERIMENT INPUT.

Applications	Main Kernel	Input	Co-run Indicator	Device Choosing Indicator	Type
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	Performance similar
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	Performance similar
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 (HS)	bucketsort	r	0.98	0.01	Performance similar



(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) Breadth-First Search (BFS) (t) Hybrid Sort (HS)

Fig. 3. Execution time for the Rodinia benchmark suite. Co-run-friendly programs are (a, b, c), GPU-dominant programs are (d, e, f, g, h, i, j, k, l, m, n, o), CPU-dominant programs are (p, q), and performance similar programs are (r, s, t).

[6] 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.

[7] D. Foley, M. Steinman, A. Branover, G. Smaus, A. Asaro, S. Punyamurtula, and L. Bajic. AMD's 'Llano'Fusion APU. In *Hot Chips*, volume 23, 2011.

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

[9] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, 2014.

[10] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang. Understanding Co-Run Degradations on Integrated Heterogeneous Processors. In *The 27th International Workshop on Languages and Compilers for Parallel Computing*, page 15, 2014.