Workload-Aware Optimal Power Allocation on Single-Chip Heterogeneous Processors

Jae Young Jang, Hao Wang, Euijin Kwon, Jae W. Lee, and Nam Sung Kim, *Senior Member, IEEE*

Abstract—As technology scales below 32 nm, manufacturers began to integrate both CPU and GPU cores in a single chip, i.e., single-chip heterogeneous processor (SCHP), to improve the throughput of emerging applications. In SCHPs, the CPU and the GPU share the total chip power budget while satisfying their own power constraints, respectively. Consequently, to maximize the overall throughput and/or power efficiency, both power budget and workload should be judiciously allocated to the CPU and the GPU. In this paper, we first demonstrate that optimal allocation of power budget and workload to the CPU and the GPU can provide 13 percent higher throughput than the optimal allocation of workload alone for a single-program workload scenario. Second, we also demonstrate that asymmetric power allocation considering per-program characteristics for a multi-programmed workload scenario can provide 9 percent higher throughput or 24 percent higher power efficiency than the even power allocation per program depending on the optimization objective. Last, we propose effective runtime algorithms that can determine near-optimal or optimal combinations of workload and power budget partitioning for both single- and multi-programmed workload scenarios; the runtime algorithms can achieve 96 and 99 percent of the maximum achievable throughput within 5-8 and 3-5 kernel invocations for single- and multi-programmed workload cases, respectively.

Index Terms—Single-chip heterogeneous processor, GPU, dynamic voltage and frequency scaling, runtime system, multicores

1 Introduction

Technology scaling has reduced the area, delay, and power consumption of CMOS devices, allowing manufacturers to integrate more transistors per chip. With more transistors available for integration, manufacturers have introduced multi- and many-core processors that exploit thread- and application-level parallelism to satisfy the ever-increasing performance demands for emerging applications. With this approach, however, the number of cores per chip is often limited by power and thermal constraints that do not scale with technology scaling [1], [2], [3]. This in turn will eventually limit the maximum performance that can be delivered by future many-core processors. Thus, improving power efficiency has become one of the most critical design goals for high-performance many-core processors.

To maximize power efficiency, single-chip heterogeneous processors (SCHPs), which are comprised of various types of processing elements such as CPUs, GPUs, and accelerators, have been widely adopted by all computing segments [4]. Typically, the parallel portions of compute-intensive workloads execute on the GPU, and the serial portions on the CPU. Besides, CPUs often provide much higher performance than GPUs when executing codes with irregular and complex data and control dependence. Integrating both CPU and GPU cores onto a single chip can greatly reduce the performance and energy penalties incurred by communications between them. This reduction will be more pronounced as the number of CPU and GPU cores placed on a chip continues to increase in the foreseeable future with technology scaling.

Recent studies demonstrate that workload partitioning between the CPU and the GPU can improve the overall throughput or power efficiency of a multi-chip heterogeneous computing system (comprised of discrete CPU and GPU components) [5], [6], [7], [8], [9]. Such workload partitioning can also improve the overall throughput of SCHPs. However, the CPU and GPU must share a chip power budget due to their integration on a single chip, and the CPU or GPU must also satisfy its own power constraint due to thermal and reliability constraints. The relative performance of a CPU or GPU is often proportional to its power consumption (i.e., assigned power), yet the CPU and GPU exhibit different performance and power efficiency depending on the characteristics of the executed workload. Therefore, a joint optimization of both workload and power budget partitioning between the CPU and the GPU can help to increase the overall throughput and/or power efficiency of SCHPs.

Note that in current SCHPs neither the CPU nor the GPU can consume the entire chip power budget. In other words, the power budget of the CPU or GPU alone is always lower than the power budget of the entire SCHP. Hence, assigning the entire workload to either the CPU or GPU is not an effective way to maximize the overall throughput of SCHPs.

This paper proposes workload-aware optimal power allocation schemes on an SCHP with separate voltage/frequency (V/F) domains for the CPU and the GPU. We

- *J.Y. Jang and J.W. Lee are with the College of Information and Communication Engineering, Sungkyunkwan University (SKKU), Suwon 440–746, Korea. E-mail: {rhythm2jay, jaewlee}@skku.edu.*
- *H. Wang and N.S. Kim are with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706. E-mail: {hwang223, nskim3}@wisc.edu.*
- *E. Kwon is with System LSI Buisness, Samsung Electronics, Yongin 446–711, Korea. E-mail: euijin.kwon@samsung.com.*

Manuscript received 11 Nov. 2014; revised 25 May 2015; accepted 23 June 2015. Date of publication 7 July 2015; date of current version 18 May 2016. Recommended for acceptance by Y. Solihin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2453965

first investigate optimal workload and power budget partitioning for *single-programmed* workloads and introduce an effective runtime algorithm that finds a (near-)optimal V/F configuration. Then we extend it to *multi-programmed* workloads assuming the CPU with per-core dynamic V/F scaling (DVFS) and the GPU with two independent V/F domains. Since different programs exhibit non-uniform performance sensitivities to operating frequencies (hence allocated power budget), it is necessary to search for an optimal V/F setting at runtime by considering both workload characteristics and evaluation metrics.

In summary, this paper makes the following contributions:

- We demonstrate that a joint optimization of both workload and power budget partitioning between the CPU and the GPU can lead to considerably higher throughput for single-programmed workloads than optimization of workload partitioning alone with fixed power budget allocations to the CPU and GPU.
- We analyze potential throughput improvement of adaptive, workload-aware power allocation schemes for *multi-programmed* workloads. We find that the optimal V/F settings for the CPU and the GPU vary greatly depending on program characteristics and evaluation metrics.
- We propose an effective runtime algorithm that can determine (near-)optimal workload and power budget partitioning for single-programmed workload within a small number of kernel invocations. The runtime algorithm, which can be integrated with the OpenCL runtime layer, exploits the runtime power efficiency ratio between the CPU and the GPU when executing a given workload.
- We propose and evaluate two runtime algorithms that can maximize throughput and power efficiency, respectively, by determining optimal V/F settings for multiple programs running concurrently.

The rest of the paper is organized as follows. Section 2 presents background on SCHPs. Section 3 describes our experimental methodology. Sections 4 and 5 demonstrate the potential throughput and power efficiency improvements achieved by joint optimization of workload and power budget partitioning for single- and multi-programmed workloads, respectively. Section 6 describes our proposed runtime algorithms and evaluates their effectiveness. Section 7 discusses related work, and Section 8 concludes this paper.

2 Background

OpenCL extends C to provide a language to access and manage heterogeneous computational resources [7]. OpenCL is designed to efficiently support parallel execution on single or multiple programmable processors (e.g., GPUs, CPUs, DSPs, FPGAs) using a data- and task-parallel computing model. OpenCL allows any processors in the computing system to act as peers by abstracting the specifics of the underlying hardware. The software stack of OpenCL is composed of: (i) a platform layer that queries and selects compute devices (e.g., CPUs and GPUs) in the platform, initializes the compute device(s), and creates compute contexts and work-queues; (ii) a runtime layer that manages computational resources and executes compute kernels; and (iii) a compiler that supports a subset of ISO C99 with appropriate language extensions and compiles/builds executable compute kernels online or offline.

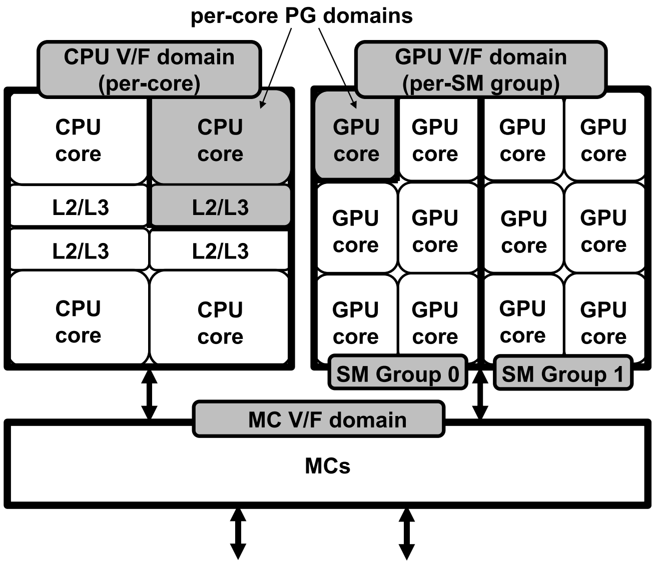

Fig. 1. Block diagram of a SCHP comprised of CPU and GPU cores. The memory controllers (MCs) are shared by both the CPU and GPU cores.

Fig. 1 shows a block diagram of an SCHP, similar to processors from AMD and Intel. For example, AMD's Llano APU integrates four x86 Stars CPU cores and 80 VLIW-5 Radeon GPU cores in a single chip in 32 nm technology [10]. Each CPU core supports out-of-order execution and has a 1 MB L2 cache. Each GPU core is comprised of four stream cores, a special-function stream core, a branch unit, and a register file. The CPU and GPU cores share two DDR3 1,866 memory controllers (MCs) and the chip-level power budget of 100 W [10].

For efficient power management, SCHPs typically support separate voltage/frequency (V/F) domains for the CPU and GPU cores. They also provide power-gating (PG) devices for each CPU and GPU core (or each group of GPU cores). These two features allow an SCHP to dynamically allocate its total power budget between the CPU and the GPU under the chip power constraints [11], [4].

Traditionally, dynamic voltage/frequency scaling (DVFS) has been widely used to optimize performance under a power constraint [12]. However, as multi-/many-core processors running multiple threads and/or applications emerge, dynamically varying the number of on/off (or active/inactive) cores based on the degree of parallelism and/or problem size in the application can also provide opportunities to optimize performance under a power constraint. This is known as dynamic core scaling (DCS) [13], [14]. It has been demonstrated that the DCS can be effectively combined with DVFS (i.e., providing dynamic voltage/frequency/core scaling (DVFCS)) to optimize performance in multi-threaded and multi-programmed applications. This paper extends [13], [14] by applying DVFCS in a single-chip heterogeneous computing environment, in which both the workload and chip power budget are partitioned across the CPU and GPU to maximize the overall throughput.

TABLE 1
Key Configuration Parameters for Our Single-Chip Heterogeneous Processor

# of CPU cores	4	# of GPU SMs / V/F domains	12 / 2
CPU freq / volt	1.67-3.44 GHz/0.72-0.99 V	GPU freq / volt	350-710 MHz/0.72-0.99 V
CPU core fetch / issue / retire	4/4/4	GPU # of registers per SM	16,384
CPU IL1 and DL1	64 KB/2-way/64 B 2 cycles	GPU # of threads per SM	1,024
CPU L2 per core	1 MB/16-way/64 B 20 cycles	GPU # of CTAs per SM	8
CPU store buffer	16 per core	GPU L1$ per SM	32 KB
CPU core BTB	8 K/4-Way per core	GPU SIMD Width	8
CPU branch mis-pred. penalty	14 cycles	GPU Warp Size	32
CPU MSHR	20 per core	GPU branch divergence	Immediate post dominator
Memory freq	1,866 MHz (DDR3)	# of MC / scheduling policy	2/FR-FCFS

3 Experimental Methodology

3.1 Baseline Processor Configuration

In this study, the CPU and GPU are integrated into a single chip and share common MCs to service memory access requests from both the CPU and GPU. Because this leads to memory access contentions, it is critical to build a detailed cycle-level simulator to accurately model the effects of the shared main memory bandwidth between the CPU and GPU, as well as the memory access contentions. Our integrated simulator infrastructure has been developed based on widely-used simulators: gem5 [15] and GPGPU-Sim [16] for modeling the CPU and GPU, respectively. Wang et al. provide more details about the simulator [7].

To model the interactions at the MCs, we use a shared memory programming model in the Linux OS. In our study, the CPU and GPU execute data-parallel threads from the same compute kernel, but on statically partitioned datasets. As in the current AMD Llano APU architecture [10], the CPU and GPU caches do not maintain an explicit coherence protocol. Defining a GPU memory hole in the memory map of the CPU, and having all the data that the GPU may access lie in the (physically mapped) memory hole, eliminates the need for coherence mechanisms; while the CPU cores have separate L1 and L2 caches that are coherent with main memory, the GPU cores have private L1 caches that only support a weak consistency model. The memory access scheduler at the MC adopts First-Ready First-Come-First-Served (FR-FCFS) policy [17] with per-bank front-end buffers

We configure our simulator to model an SCHP with four CPU cores. We set the number of GPU SMs based on a recent discrete GPU; NVIDIA's GT260M, which has a similar peak throughput as the GPUs integrated in Intel's and AMD's SCHPs. The GT260M has 12 SMs with 396GFLOPs of throughput and consumes 38 W at the nominal operating V/F [18]. Table 1 summarizes the key configuration parameters.

We assume that the total chip power budget allowed for both the CPU and GPU is 100 W. To model CPU and GPU power consumption, we first apply the configuration parameters from Table 1 to the CPU and GPU power models from [19] and [20], respectively. This allows us to calculate the ratio between dynamic and leakage power of CPU cores and GPU SMs at nominal VDD (0.9 V). Following the modeling methodology presented in [13], we obtain the frequency and leakage scaling factors by estimating the delay of a 24-stage FO4 inverter chain and a dummy leakage circuit that includes a large number of NOT (50 percent), NAND (30 percent), and NOR (20 percent) gates using SPICE and a 32 nm technology model [21] at different VDD levels.

With (i) the ratio between dynamic and leakage power at the nominal VDD, (ii) frequency and leakage scaling factors as functions of voltage, and (iii) the known power consumption of the CPU and GPU at the nominal VDD, we can calculate the power consumption of the CPU and GPU as a function of V/F and the number of active CPU cores and GPU SMs; Table 2 tabulates the frequency and power consumption of a CPU core and a GPU SM for different voltages.

Considering such an SCHP under power constraints, we establish four initial evaluation configurations for the single programmed workload: (i) GPU-oriented, (ii) CPU-oriented, (iii) GPU-only, and (iv) CPU-only. Configurations (i) and (ii) allocate more power budget to the GPU and CPU, respectively, while configurations (iii) and (iv) use only either the GPU or the CPU in the SCHP. In configuration (i), three CPU cores at 0.9V/2.9 GHz and 12 SMs at 0.9V/600 MHz are active and consume 60 and 40 W, respectively. In configuration (ii), 4 CPU cores at 0.9 V/2.9 GHz and 6 SMs at 0.9 V/600 MHz are active and consume 80 and 20 W, respectively. In configuration (iii) only 12 SMs are active and consume 40 W 0.9 V/600 MHz. In configuration (iv), 4 CPU cores are active and consume 80 W at 0.9 V/2.9 GHz. The maximum power consumption allowed is 80 W for the CPU and 63 W for the GPU. This occurs by running the CPU/GPU at 0.99 V, the maximum voltage allowed under thermal and reliability constraints.

TABLE 2
Voltage versus Frequency and Power Consumption of a CPU Core and GPU SM

Voltage (V)	Freq (GHz)		Power (Watts)	
	CPU	GPU	CPU	GPU
0.72	1.67	0.35	7.1	1.2
0.75	1.89	0.39	8.7	1.4
0.78	2.09	0.43	10.4	1.7
0.81	2.31	0.48	12.4	2.1
0.84	2.51	0.52	14.6	2.4
0.87	2.70	0.56	17.1	2.8
0.90	2.90	0.60	20.0	3.3
0.93	3.07	0.64	23.2	3.9
0.96	3.26	0.68	27.0	4.5
0.99	3.44	0.71	31.4	5.2

TABLE 3
The Full Name of Each Benchmark, its Corresponding Abbreviation, and its Input Data Size

Benchmark	Abbrev.	Input Data Size
Back Propagation	BPR	131,072 input nodes
Breadth-First Search	BFS	1,000,000 nodes
CFD Solver	CFD	200k elements
Heart Wall Tracking	HWT	609 × 590 pixels / frame
HotSpot	HOT	1,024 × 1,024 data points
Leukocyte Tracking	LKT	480 × 640 pixels / frame
LU Decomposition	LUD	512 × 512 data points
Needleman-Wunsch	NW	2,048 × 2,048 data points
K-means	KMN	204,800 points, 34 features
SRAD	SRAD	2,048 × 2,048 data points

In this paper, we assume that the power consumption of other on-chip components (e.g., I/O, MCs, and other peripheral components), which are connected to different power domains, is constant regardless of V/F and the number of active CPU cores and GPU SMs; we only consider the power consumption of the CPU and GPU after excluding this fixed power component for the total chip power budget and consumption.

3.2 Evaluated Workloads

We use 10 benchmarks (Back Propagation, Breadth-First Search, CFD Solver, Heart Wall Tracking, HotSpot, Leukocyte Tracking, LU Decomposition, Needleman-Wunsch, K-means, and SRAD) from Rodinia, a benchmark suite for heterogeneous computing covering a wide range of parallel communication patterns and synchronization techniques [22]. The abbreviation for each benchmark and its input data sizes are described in Table 3.

Each benchmark is either an application executing one or several parallel kernels many times, or a kernel that is called iteratively in real applications. For example, CFD Solver, which is the first case, executes two kernels 2,000 times by default; and HotSpot, which corresponds to the second case, updates the temperature profile for one small time step, and should be called iteratively to estimate the processor temperature based on a simulated power trace for a certain period of time. Because the parallel code portions of the program dominate the total execution time, we focus on improving the throughput of the parallel code portions, or kernels. These parallel kernels are written in CUDA or OpenCL for the GPU and in OpenMP for the CPU.

4 Optimizing Single-Programmed Workloads

4.1 The Need to Use Both the CPU and GPU

In an SCHP, a proper partitioning of the single programmed workload between the CPU and the GPU can improve overall throughput. There are two reasons for this. First, the performance of the CPU (or GPU) varies based on its power consumption when DVFS is employed. Second, the CPU (or GPU) alone cannot consume the total chip power budget due to its thermal and/or maximum voltage constraints.

Consequently, the CPU and GPU should operate in parallel to consume the entire chip power budget and thus maximize overall application throughput. To adjust the power allocation between the CPU and the GPU within the chip power budget, we explore two options: (i) dynamic core scaling enabled by per-core and per-SM PG devices and (ii) dynamic voltage/frequency scaling enabled by two separate V/F domains; one for the CPU and one for the GPU.

Under such conditions, instead of assigning all the work (i.e., threads) to the GPU, whose maximum throughput is limited by its power constraint (i.e., 63 W in our baseline), we can assign some work to the CPU such that the remaining chip power budget (i.e., 37 W out of 100 W in our baseline) can be fully used to maximize the overall throughput. For the DCS option, all 12 GPU SMs are active and consume 40 W while only three CPU cores operating at the nominal V/F consume 60 W. Alternatively, four CPU cores can consume 60 W at lower V/F, leaving 40 W for the GPU.

4.2 Power Allocation through DCS

Fig. 2 plots the maximum throughput of different SCHP configurations for all the benchmarks we examine. For CPU- and GPU-oriented configurations, we simulated 16 workload partitioning points (i.e., from 6.25 to 100 percent of the workload assigned to the GPU) for each benchmark, and pick the percentage value that maximizes throughput. To facilitate exploration of huge configuration space, we make the simplifying assumption that the workload can be partitioned at a thread (or loop iteration) granularity. Even if we partition the workload at a coarser granularity (e.g., thread blocks), its impact will be marginal, as we assume a large input data size with many thread blocks to fully saturate the hardware resources. Table 4 tabulates the optimal percentage values for all the benchmarks.

The throughputs in Fig. 2 are normalized to that of the GPU-only configuration in which all the CPU cores are

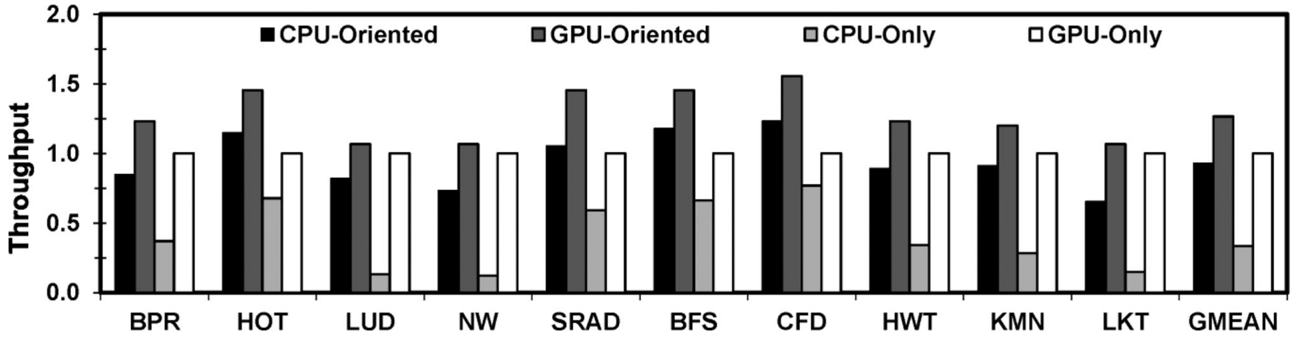

Fig. 2. Throughput of different SCHP configurations with fixed power budgets. Throughput is normalized to GPU-only.

TABLE 4
Percentage of Workload Assigned to GPU to Optimize the Throughput of Each Configuration

Benchmark	CPU-oriented	GPU-oriented	Benchmark	CPU-oriented	GPU-oriented
BPR	56.3%	81.3%	LKT	81.3%	93.8%
BFS	43.8%	68.8%	LUD	87.5%	93.8%
CFD	37.5%	62.5%	NW	87.5%	93.8%
HWT	62.5%	81.3%	KMN	68.8%	81.3%
HOT	43.8%	68.8%	SRAD	43.8%	68.8%

disabled and all GPU SMs are active. On average (using the geometric mean), the GPU-oriented configuration, which uses the full power budget of the SCHP, exhibits 27 percent higher throughput than the GPU-only configuration. In comparison, the CPU-oriented configuration, which also uses the full power budget of the heterogeneous processor, has 7 percent lower throughput than the GPU-only configuration. The CPU- and GPU-oriented configurations show 177 and 278 percent higher throughput than the CPU-only configuration, respectively. This supports our earlier argument: to maximize the throughput of an SCHP, the workload should be partitioned between the CPU and the GPU and executed using both the CPU and GPU.

4.3 Power Budget Partitioning Using DVFCS

Figs. 3a and 3b plot the relative execution time versus the percentage of work assigned to the GPU for HotSpot and Heartwall, respectively. These benchmarks are chosen because the optimal percentage of work assigned to the GPU of the remaining benchmarks is between those of these two benchmarks. In addition to the CPU- and GPU-oriented configurations, we apply DVFS and DCS simultaneously (i.e., DVFCS). This allows us to explore more fine-grain power allocations between the CPU and the GPU depending on the workload characteristics.

For DVFCS at each workload partitioning point, we initially perform an exhaustive search for all possible combinations of (i) V/F of the CPU and GPU and (ii) the number of active CPU cores and GPU SMs, while satisfying the chip, CPU, and GPU power constraints. In HotSpot, the highest throughput is achieved with 68.75 percent of the work assigned to 12 active GPU SMs operating at 0.96 V/680 MHz and consuming 54.1 W. The rest of the work (31.25 percent) is assigned to four active CPU cores operating at 0.78 V/2.09 GHz and consuming 41.5 W. In Heartwall, the highest throughput is achieved with 87.5 percent of the work assigned to 12 active GPU SMs operating at 0.99 V/710 MHz and consuming 62.8 W. The rest of the work (12.5 percent) is assigned to four active CPU cores operating at 0.75 V/1.89 GHz and consuming 34.2 W.

In both HotSpot and Heartwall, the minimum execution time (i.e., maximum throughput) is achieved when both the CPU and GPU finish the assigned work approximately at the same time because the total execution time of an SCHP is the maximum of the CPU and GPU execution times. In other words, one finishing the given work much earlier than the other leads to worse performance since all the available resources and power budget are not fully and efficiently utilized. The optimal DVFCS configurations provide

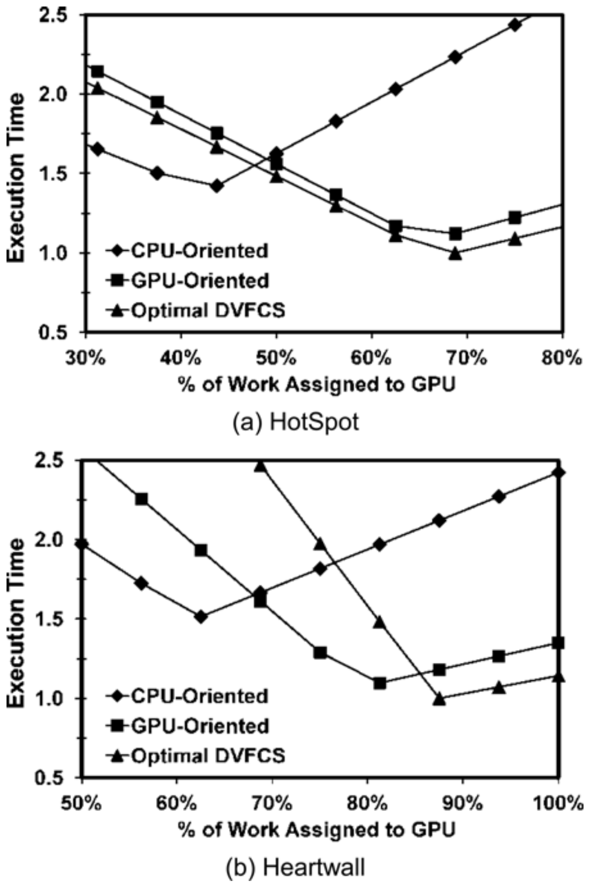

Fig. 3. Execution time versus percentage of work assigned to the GPU.

13 and 54 percent higher throughput than the GPU- and CPU-oriented configurations, respectively, while using lower V/F for the CPU and allocating more power to the GPU to make it run faster than either the CPU- or GPU-oriented configurations. Depending on how efficiently the CPU and GPU can each process a given workload, the optimal configuration assigns more work (and thus more power) to either the CPU or GPU. Note that the "optimal" DVFCS configuration may perform worse than either CPU- or GPU-oriented configuration if the workload partition is suboptimal. This clearly demonstrates that proper partitioning of both workload and power budget based on workload characteristics is a key to maximize the overall throughput.

Fig. 4 shows the throughput improvement with optimal DVFCS for all the benchmarks we examine. The throughput is normalized to the GPU-oriented configuration. Compared to the results of the optimal workload partitioning under the fixed power budget for the CPU and GPU (i.e., the CPU- and GPU-oriented configurations), the joint optimization of workload and power budget partitioning delivers up to 19 percent (and on average 13 percent) higher throughput than the GPU-oriented configuration, and up to 94 percent (and on average and 54 percent) higher throughput than the CPU-oriented configuration. Table 5 summarizes the optimal percentage of work assigned to the GPU and the optimal processor configurations (i.e., the number of active CPU cores and GPU SMs, and the operating V/F and power consumption of the CPU and GPU) for all the benchmarks. In columns (2) to (5) of Table 5, the numbers in parenthesis are the value for the GPU.

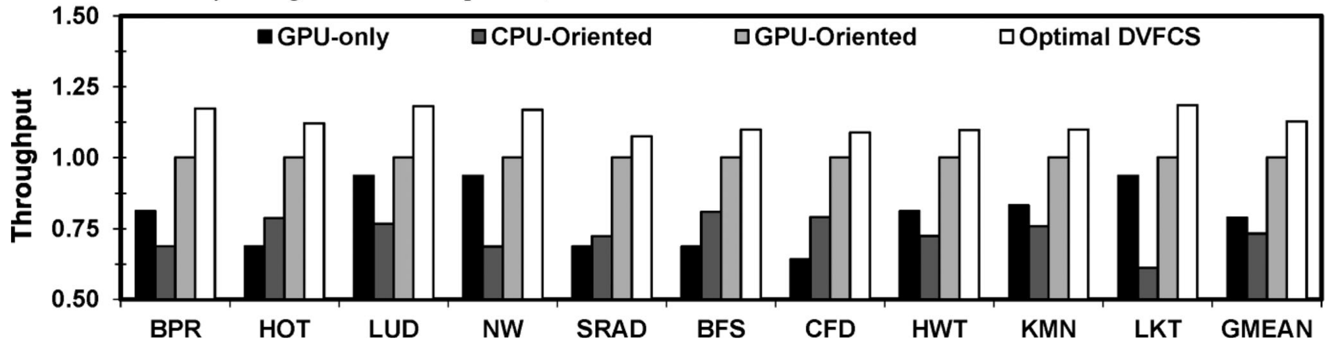

Fig. 4. Throughput improvement with fixed power budgets and optimal DVFCS. Throughput is normalized to GPU-Oriented.

4.4 Insight into Joint Optimization on Workload and Power Budget Partitioning Using DVFCS

To gain insight into developing an effective runtime DVFCS algorithm that can determine the optimal configurations (i.e., the optimal percentage of work to assign to the GPU, number of active CPU cores and GPU SMs, and their V/F values), we plot the power efficiency ratio of the CPU to the GPU for the different configurations used to generate Fig. 4. We define power efficiency at the optimal workload partitioning point, where both the CPU and GPU exhibit the same execution time for a given workload, as follows:

$$\frac{P_{EFF\_CPU}}{P_{EFF\_GPU}} = \frac{W_{CPU}/P_{CPU}(V_{CPU}, F_{CPU}, N_{CPU})}{W_{GPU}/P_{GPU}(V_{GPU}, F_{GPU}, N_{GPU})} \quad (1)$$

where P_{EFF_CPU} and P_{EFF_GPU} are the power efficiency of the CPU and GPU; W_{CPU} and W_{GPU} are the percentage of work assigned to the CPU and GPU; and P_{CPU} and P_{GPU} are the power consumption of the CPU and GPU as a function of the number of active CPU cores and GPU SMs (N_{CPU} and N_{GPU}) and their V/F values (V_{CPU} /F_{CPU} and V_{GPU} /F_{GPU}).

As we can observe from the results shown in Fig. 5, the power efficiency ratios are very low for the CPU- and GPU-oriented configurations; on average 0.14 for both configurations. Such a low ratio indicates that the CPU is using its power less efficiently than the GPU, and therefore more power should be shifted to GPU so that the SCHP can use the limited power budget more efficiently to improve overall throughput. As the CPU operates at lower V/F to allocate more power to the GPU, it becomes more power-efficient. Note that power efficiency increases as the operating voltage decreases. This is because power is reduced super-linearly while frequency (and the performance of compute-bound applications) is degraded close to linearly.

On the other hand, as the GPU shifts to a higher V/F operating point by exploiting the power transferred from the CPU, the power efficiency of the GPU decreases. After the DVFCS optimization, we observe that the power efficiency ratios significantly improve. We note that throughput can be maximized by allocating more power from a less power-efficient device to a more power-efficient device. This increases the power efficiency ratio; the average power efficiency ratio with the optimal DVFCS is increased from 0.14 to 0.34. Theoretically, the optimal DVFCS technique should perfectly balance the power efficiency between the CPU and GPU, but (i) the limited V/F levels due to the minimum operating V/F constraint for the CPU and the maximum operating V/F constraint for the GPU, (ii) discrete V/F levels, and (iii) discrete workload partitioning points prevent the optimal DVFCS from reaching the perfectly balanced point (i.e., a power efficiency ratio of 1.0) in Fig. 5.

The key point is that the CPU and GPU have different performance and power efficiency characteristics that scale differently based on workload characteristics when varying V/F levels, the number of CPU cores, and the number of GPU SMs. This provides the opportunity to improve the overall throughput of power-constrained SCHPs through DVFCS. Furthermore, comparing the power efficiency of the CPU and GPU provides a promising basis for an effective runtime DVFCS algorithm for workload and power budget partitioning that does not need to collect instruction-level statistics.

TABLE 5
Optimal (1) Percentage of Work Assigned to the GPU, (2) Number of CPU Cores (GPU SMs), (3) Voltage in Volts, (4) Frequency in GHz, and (5) Power Consumption of the CPU (GPU) in Watts

	(1)	(2)	(3)	(4)	(5)
BPR	81.3	4(12)	0.78(0.96)	2.09(0.68)	41(54)
BFS	68.8	4(12)	0.78(0.96)	2.09(0.68)	41(54)
CFD	68.8	4(12)	0.75(0.99)	1.89(0.71)	34(63)
HWT	87.5	3(12)	0.75(0.99)	1.89(0.71)	26(63)
HOT	68.8	4(12)	0.78(0.96)	2.09(0.68)	41(54)
LKT	93.8	3(12)	0.78(0.99)	2.09(0.71)	31(63)
LUD	93.8	2(12)	0.87(0.99)	2.70(0.71)	34(63)
NW	93.8	3(12)	0.78(0.99)	2.09(0.71)	31(63)
KMN	87.5	3(12)	0.78(0.99)	2.09(0.71)	31(63)
SRAD	75.0	3(12)	0.81(0.99)	2.31(0.71)	37(63)

5 Optimizing Multi-Programmed Workloads

Recently, GPU performance has improved more rapidly than CPU performance on SCHPs. For example, Apple's A8X system-on-a-chip (SoC) has 2.5 times GPU performance compared to its predecessor (A7), while improving CPU performance only by a factor of 1.4 [23]. This trend is likely to continue due to growing performance demands from data-parallel applications on both mobile and desktop platforms. To efficiently utilize GPU resources, future GPUs will support device fission (i.e., dividing one device into one or more sub-devices to run multiple kernels concurrently) [24]. This section investigates optimal workload partitioning for multi-programmed workloads.

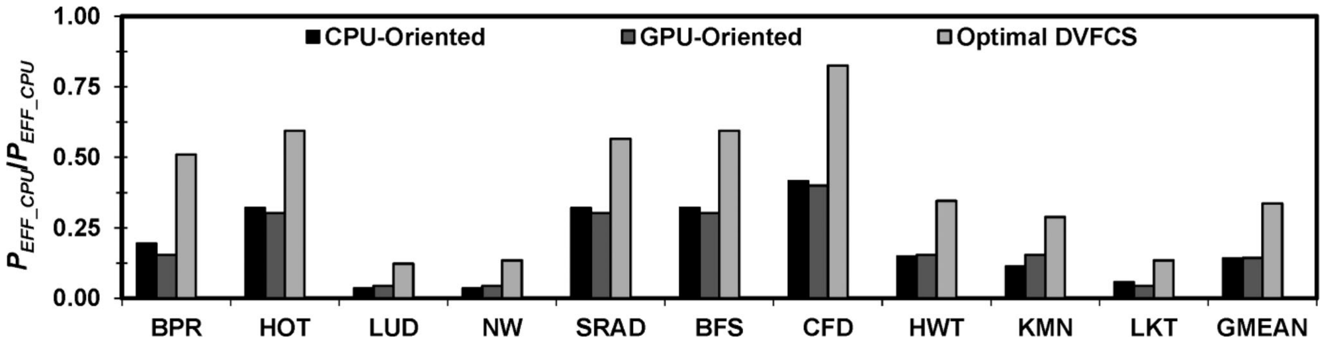

Fig. 5. Power efficiency ratio of the CPU to the GPU for different processor configurations.

5.1 Methodology

Platform. As a simple realization of device fission, we adopt a static partitioning method of the GPU into two groups of GPU cores (i.e., SMs) and assign an independent V/F domain for each group. Specifically, we assumed an SCHP has four CPU cores and two GPU SM groups, each of which has six SMs. Also, to maximize power efficiency, we assume each CPU core is supported by its own V/F domain. For the rest of this paper, we will use this SCHP as our baseline.

To run a multi-programmed workload, we create two GPGPU-Sim instances and a gem5 instance. Each GPGPU-Sim instance corresponds to an SM group with a separate set of request and response queues to communicate with gem5 for memory accesses. By controlling ticks for each GPGPU-Sim instance, we can model per-SM group V/F domains.

Power model. For multi-programmed workloads, we calculate the power consumption of the CPU and the GPU as a function of V/F as in Section 3.1 Baseline Processor Configuration. Assuming two programs running concurrently, the baseline configuration allocates hardware and power resources equally to both programs. With the total power budget of 100 W, one program run on six GPU SMs operating in tandem at 0.93 V/0.64 GHz (consuming 23.4 W), and two CPU cores at 0.81 V/2.31 GHz (consuming 24.8 W). The total power budget is almost equally allocated for both the CPU and the GPU with 49.6 W for four CPU cores and 46.8 W for 12 GPU SMs. Hence, each program can use 48.2 W, and the total power consumption to 96.4 W.

Dividing the GPU into two V/F domains allows one GPU SM group to run at higher V/F than 0.99 V/0.71 GHz (i.e., the maximum V/F in Table 1), if the other SM group runs at lower V/F. For example, if an SM group consumes 23.4 W at 0.93 V/0.64 GHz, 39.6 W is available for the other SM group. Without exceeding the entire GPU power budget of 63 W, the latter can safely consume 39.3 W at 1.05 V/0.80 GHz. Thus, we increase the maximum V/F for a SM group from 0.99 V/0.71 GHz to 1.05 V/0.80 GHz. Now 12 V/F levels are available for the GPU with maximum power consumption of 6.55 W for each SM.

Workloads. To create multi-programmed workloads with two programs, we choose six benchmarks (Breadth-First Search, Heartwall, HotSpot, K-means, Needleman-Wunsch, and SRAD) with three compute-bound ones (Breadth-First Search, Heartwall, and SRAD) and three memory-bound ones (K-means, Hotspot, and Needleman-Wunsch). Then, we create 15 multi-programmed workloads (1) by taking one from each group (nine workloads) and (2) by taking distinct two within each group (six workloads). As a throughput measure we use (loop) iterations per second (IPS) to calculate the combined throughput for a program running across CPU and GPU. As a power efficiency measure we use IPS per Watt. For both metrics we report figures normalized to the baseline configuration.

5.2 Workload-Aware Asymetric Power Allocation for Multi-Programmed Workloads

Multiple V/F domains enable independent operating frequency settings for CPU cores and GPU SM groups in an SCHP. This introduces an optimization problem of allocating the chip power budget across processing elements while maximizing the overall throughput or power efficiency. An optimal power allocation depends on the characteristics of a given set of programs since different programs have non-uniform performance sensitivities to changes in allocated power budget (i.e., the number and types of processing elements and their V/F setting). It is also affected by the evaluation metric (e.g., throughput versus throughput/Watt).

In a multi-programmed environment the chip-level power budget should be allocated to programs running concurrently. By exploiting different performance sensitivity to allocated power budget, we can find optimal, possibly asymmetric, V/F settings that can maximize overall throughput or throughput/Watt. In general, the throughput of a compute-bound program increases more rapidly than a memory-bound program as the allocated power budget increases. Thus, increasing power allocation for the CPU and/or GPU cores that execute the memory-bound program does not benefit as much as the compute-bound program since the shared memory system becomes a throughput bottleneck.

Once per-program power budget is determined, there is another level of power budget allocation between CPU cores and GPU SMs running the same program. If a program has complex data dependences and control flows, it is likely to run more efficiently on the CPU, hence more power budget should be allocated to the CPU; if a program has abundant data-level parallelism, more power budget should be allocated to the GPU. As in the single-programmed case, even if a program is suitable for either a CPU or a GPU, neither of them can consume the entire chip power budget, due to thermal and reliability constraints and thus distributing the workload on both the CPU and the GPU leads to higher throughput. Hence, it is highly desirable to dynamically allocate the chip power budget by considering the characteristics of individual programs.

5.3 Potential Benefit of Workload-Aware Power Allocation

To demonstrate the potential benefit of workload-aware, asymmetric power allocation among programs, we create 15 workloads, each of which consists of two programs from one of the two benchmark groups: compute-bound (C), and memory-bound (M). A benchmark is classified as either compute- or memory-bound based on how well its throughput scales as V/F goes up. If a benchmark scales relatively well in comparison to other benchmarks, it is compute-bound; otherwise, it is memory-bound. For those multi-programmed workloads one program uses a half of the available computing resources for execution, two CPU cores and one GPU SM group (six SMs). Although simplistic, this assumption enables us to gain an insight into an optimal power budget and workload partitioning problem without exploding exploration space. Moreover, supporting additional V/F domains for the GPU would incur significant hardware overhead, which is less practical.

We evaluate the performance using two metrics: IPS for throughput and IPS per Watt (IPS/Watt) for power efficiency. Both numbers are normalized to the corresponding numbers with the baseline in Section 5.1 Methodology. We use IPS, instead of instructions per cycle (IPC), since we need to calculate combined throughput for a program across both CPU and GPU to compare relative overall throughput improvement of the two concurrent programs.

To find the optimal power allocation between two programs, we perform an exhaustive search for all of the 15 workloads. The results are summarized in Table 6. All of the six benchmark programs we use for evaluation have ample data-level parallelism to favor the GPU over the CPU. However, a single GPU group cannot use more than 39.3 W (at 1.05 V/0.80 GHz) due to its power constraint. Thus, it is beneficial to exploit the unused power budget of CPU cores for executing part of the work. Furthermore, the optimal power allocation between the CPU and the GPU, as well as between two programs, is affected not only by workload characteristics but also by the evaluation metric.

Fig. 6 shows the IPS and IPS/Watt improvements for the 15 workloads with the optimal power allocation. All numbers are normalized to the baseline case where the power budget is almost equally distributed among the four CPU/GPU groups with independent V/F domains. On average, the optimal configuration improves IPS and IPS/Watt by 9 and 24 percent, respectively. Thus, workload-aware, asymmetric power allocation among processing elements can improve IPS and IPS/Watt significantly.

TABLE 6
Optimal Power Allocation for 15 Workloads

P1	P2	Metric 1: IPS				Metci 2: IPS/Watt			
		P1 (Watt)		P2 (Watt)		P1 (Watt)		P2 (Watt)	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
BFS (C)	KMN (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
BFS (C)	HOT (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
BFS (C)	NW (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
HWT (C)	KMN (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
HWT (C)	HOT (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
HWT (C)	NW (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
SRAD (C)	HOT (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
SRAD (C)	KMN (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	12.60
SRAD (C)	NW (M)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
BFS (C)	HWT (C)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
BFS (C)	SRAD (C)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
HWT (C)	SRAD (C)	17.40	39.30	17.40	23.40	17.40	16.80	17.40	16.80
KMN (M)	HOT (M)	17.40	39.30	17.40	23.40	17.40	12.60	17.40	12.60
KMN (M)	NW (M)	17.40	23.40	17.40	39.30	17.40	12.60	17.40	16.80
NW (M)	HOT (M)	17.40	39.30	17.40	23.40	17.40	12.60	17.40	16.80

5.4 Case Studies: Insight into Optimal Power Allocation for Multi-Programmed Workloads

Fig. 7 shows the results of exhaustive search for three representative cases out of the 15 workloads (shaded in Table 6). The *X*- and *Y*-axes show the power allocated to the two co-scheduled programs, respectively. Note that we use five levels of power allocation for both CPU and GPU (as shaded in Table 2). For the GPU there is one more V/F state available (1.05 V/0.99 GHz) as discussed in Section 5.1. Hence, there are 30 levels of power configuration for each program.

In our experiments, we scale only GPU V/F level and fix the CPU V/F level to the lowest one since all of the benchmarks we use for evaluation have ample data parallelism to favor the GPU. In other words, the GPU plays a dominating role for overall throughput. Thus, we derive a heuristic based on our observations, which sets the CPU V/F level to the lowest so that more power can be allocated to the GPU. This reduces the possible V/F combinations for each program to be six, and total V/F combinations to be 36 for two programs. However, three combinations exceed the

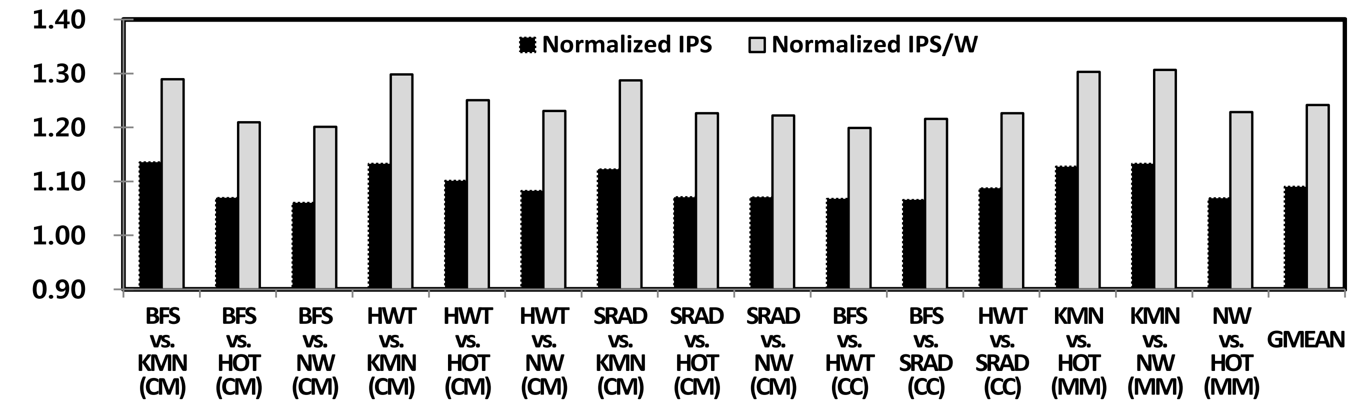

Fig. 6. Overall IPS and IPS/Watt for 15 multi-programmed workloads.

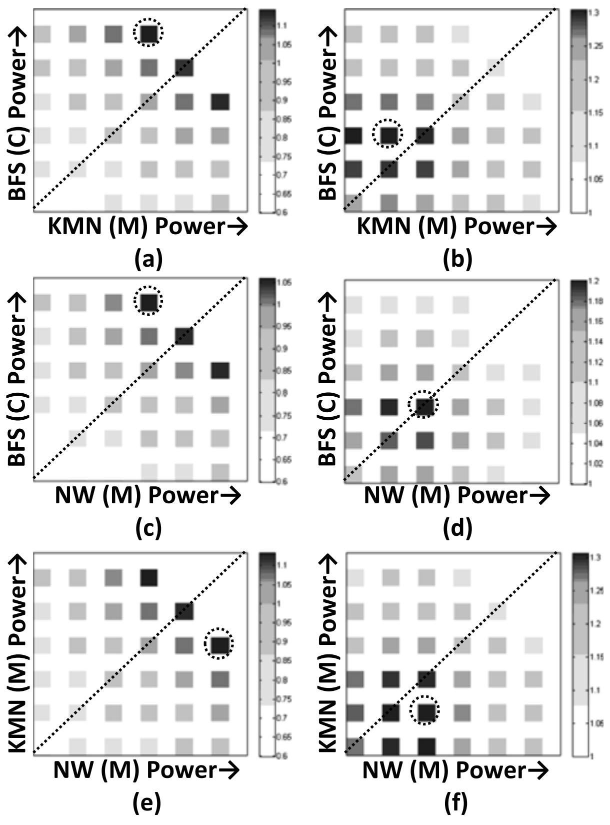

Fig. 7. Results of exhaustive search: (a) IPS and (b) IPS/Watt for BFS versus KMN; (c) IPS and (d) IPS/Watt for HWT versus SRAD; (e) IPS and (f) IPS/Watt for KMN versus NW.

chip-level power budget; hence only 33 combinations (instead of 296 with varying CPU V/F settings) are possible as illustrated in Fig. 7.

We use both IPS and IPS/Watt for performance metrics, and all numbers are normalized to the baseline case (i.e., GPU at 0.93 V/0.64 GHz and CPU at 0.81 V/2.31 GHz). The baseline case almost uses up the entire chip power budget (96.4 W) and fairly allocate it between the CPU and the GPU. In Fig. 7 the higher the number is, the darker the corresponding box is. A circle marks the optimal power configuration; the dotted line in each graph indicates a line of uniform power allocation between the two programs.

Case 1. BFS (C) versus KMN (M): Figs. 7a and 7b show IPS and IPS/Watt, respectively, when running BFS (compute-bound) and KMN (memory-bound) concurrently. As expected, allocating more power to BFS improves both IPS and IPS/Watt since BFS has a higher marginal throughput gain to additional power allocation than KMN. The optimal configuration improves IPS by 13.5 percent and IPS/Watt by 28.9 percent over the baseline.

Case 2. HWT (C) and SRAD (C): Figs. 7c and 7d show the IPS and IPS/Watt curves when running compute-bound programs concurrently: HWT and SRAD. Although they are all compute-bound programs, HWT is relatively more compute-intensive than SRAD. Therefore, to optimize IPS, more power budget should be allocated to HWT, which has a higher marginal gain. For IPS/Watt, there are multiple optimal configurations whose IPS/Watt numbers are almost the same. We observe 8.7 and 22.6 percent improvements over the baseline case for IPS and IPS/Watt, respectively.

Case 3. KMN (M) versus NW (M): Figs. 7e and 7f show the IPS and IPS/Watt curves for a workload composed of memory-bound programs together: KMN and NW. Although they have similar performance characteristic (memory-bound), we find that the optimal configuration for IPS falls on the NW side. As expected, it is because marginal throughput gain of NW is bigger than KMN. Accordingly, allocating more power on NW is beneficial for IPS as well as IPS/Watt. Note that, in general, IPS/Watt favors non-maximum configurations because of diminishing returns of power efficiency as the allocated power budget increases. In this case, we observe 13 and 30.7 percent gains of IPS and IPS/Watt, respectively, over the baseline configuration.

6 Runtime DVFCS Algorithms

6.1 Single Programmed Workloads

According to the analysis performed in Section 4, the basic idea of the runtime DVFCS algorithm for single programmed workloads is to adaptively adjust the power efficiency between the CPU and the GPU through DVFCS such that the overall throughput is maximized for a given workload. To develop a runtime DVFCS algorithm, we need to answer the following two questions:

- What percentage of work should we assign to the CPU and GPU, respectively?
- Which knob, DVFC or DCS, should we use first to adjust power efficiency?

In this section, we, answer these two questions, describe the proposed runtime DVFCS algorithm, and evaluate the effectiveness of the proposed runtime DVFCS algorithm using the detailed cycle-level simulator.

6.1.1 Optimization Solution Space Analysis

First, as shown in Fig. 3, the execution time of both the CPU and the GPU scales linearly with the percentage of work assigned for most benchmarks. This is because the input data size is large enough; even 6.25 percent (1/16) of the workload can fully occupy the computing resources and thus its execution time is approximately equal to 6.25 percent of the execution time.

Due to this linear relationship, the optimal workload partitioning point for given V/F levels for the CPU and the GPU, the number of CPU cores, and the number of GPU SMs can be determined after each invocation of a kernel in the main execution loop of the benchmark. Suppose the current invocation assigns X_i% of the workload to the GPU and (100-X_i)% to the CPU, and the CPU and GPU execution times are T_{CPU} and T_{GPU}, respectively. Then, the optimal workload partitioning point for the next invocation X_{i+1}% of work on the GPU can be calculated by simply solving Eq. (2):

$$T_{GPU} \cdot \frac{X_{i+1}}{X_i} = T_{CPU} \cdot \frac{100 - X_{i+1}}{100 - X_i}. \tag{2}$$

To obtain X_{i+1} as a function of X_i, T_{CPU}, and T_{GPU}, Eq. (2) can be rearranged as follows:

$$X_{i+1} = \frac{T_{CPU} \cdot X_i}{(T_{CPU} - T_{GPU}) \cdot X_i + T_{GPU}}. \tag{3}$$

```
Initialize: W_GPU = 0.5, W_CPU = 0.5;
Conf = [N_CPU=N_CPU_TOT, V/F_CPU=nom, N_GPU=N_GPU_TOT, V/F_GPU=nom];    // init. config (V/F, # of CPU cores, and # of GPU SMs)
Is_Final_VF = Is_Final_N_GPU = Is_Final_N_CPU = FALSE;               // init. flags to indicate whether or not the searches of V/F,
                                                                     // # of CPU cores, and # of GPU SMs values are completed.
[T_GPU, T_CPU] = get_exe_time(W_GPU, W_GPU, Conf);                   // get exe. time of CPU and GPU after the first kernel launch
while(!Is_Final_N_GPU || !Is_Final_N_CPU || !Is_Final_VF) {          // repeat until the search is completed
    while(!Is_Final_VF) {                                            // search V/F level first
        if(Worse || V/F_MIN == V/F_MAX)                              // worse throughput or no more V/F value to adjust:
            Is_Final_VF = TRUE; break;                               // stop V/F binary search and set Is_Final_VF flag
        W_GPU_op = get_ W_GPU_op(T_GPU, T_CPU, W_GPU, W_CPU);        // find the optimal partitioning point using Eq.(3)
        W_CPU_op = 1 - W_GPU_op;
        P_EFFGPU = W_GPU_OP / P_GPU(N_GPU, V/F_GPU);                 // calculate power efficiency of CPU and GPU at given
        P_EFFCPU = W_CPU_OP / P_CPU(N_CPU, V/F_CPU);                 // workload partitioning point
        if(P_EFFGPU < P_EFFCPU)                                      // higher CPU power efficiency:
            V/F_MAX = V/F_CPU;                                       // allocate more power to CPU (binary search w/ higher V/F)
        else                                                         // lower CPU power efficiency:
            V/F_MIN = V/F_CPU;                                       // allocate less power to CPU (binary search w/ lower V/F)
        V/F_CPU = (V/F_MIN + V/F_MAX) / 2;
        V/F_GPU = get_GPU_VF(P_TOT, P_CPU(V/F_CPU, N_GPU), N_GPU);   // calculate V/F of GPU satisfying total power budget
        [T_GPU, T_CPU] = get_exe_time(W_GPU_OP, W_GPU_OP, Conf);     // get exe. time of CPU and GPU for new workload partitioning
    }                                                                // and config
    if(Is_Final_VF && !Is_Final_N_CPU){                              // search # of CPU cores after completing V/F search
        N_CPU--; V/F_CPU++;                                          // decrease # of CPU cores/ increase V/F for the same power
        [T_GPU, T_CPU] = get_exe_time(W_GPU_OP, W_GPU_OP, Conf);     // get exe. time of CPU and GPU for new config
        if(Worse)                                                    // worse throughput or no more N_CPU to adjust:
            Is_Final_N_CPU = 1; continue;                            // stop N_CPU search
        else                                                         // better throughput
            Is_Final_VF = 0;                                         // repeat the tuning for V/F
    }
    if(Is_Final_VF && Is_Final_N_CPU && !Is_Final_N_GPU){            // repeat searching # of GPU cores similar to searching # of
        N_GPU--; V/F_GPU++;                                          // CPU cores after completing a search for CPU
        [T_GPU, T_CPU] = get_exe_time(W_GPU_OP, W_GPU_OP, Conf);
        if(Worse)
            Is_Final_N_GPU = 1; continue;
        else
            Is_Final_VF = 0;
    }
}
Opt_conf = Config;
```

Algorithm 1. Runtime DVFCS algorithm for single programmed workloads.

Second, both the CPU and GPU are generally more power efficient when operating at lower V/F. We plot HotSpot's execution time versus power efficiency ratio for different numbers of CPU cores and GPU SMs in Fig. 8. Each point represents one valid configuration (i.e., V/F of the CPU and GPU, the number of CPU cores, and the number of GPU SMs) that satisfies the chip, CPU, and GPU power constraints and delivers higher throughput than the CPU-oriented configuration.

A similar trend is also observed in all other benchmarks. Therefore, we use the following strategy for our proposed runtime DVFCS algorithm. When we need to allocate less power to the CPU (or GPU), we lower its V/F level first. If the V/F level is already at the lowest possible level but the power efficiency ratio still indicates less power should be assigned to the CPU (or GPU), we begin to turn off CPU cores (or GPU SMs). Similarly, when we need to allocate more power to the CPU (or GPU), we activate the disabled CPU cores (or GPU SMs) first. Only after there are no more available CPU cores (or GPU SMs), we begin to increase its V/F if we still need to allocate more power to either the CPU (or GPU). To set the V/F level and the number of CPU cores (or GPU SMs), we apply a binary search method for the given range of V/F levels and the number of available CPU cores (or GPU SMs). To avoid a local optimum, we run an additional iteration with fewer cores after the V/F level is found to see whether or not we can obtain a higher throughput with fewer CPU cores (or GPU SMs). Because the solution points are not very close to the optimal point, it is unlikely that the runtime DVFCS algorithm converges to a local minimum. Summarizing the above discussion, Algorithm 1 shows our runtime DVFCS algorithm.

Algorithm 1, which can be implemented as a part of a runtime system (e.g., the OpenCL runtime layer), can be invoked every time a kernel is launched. Once the throughput improvement converges, the algorithm no longer changes the V/F levels, the number of the CPU cores, or the number of GPU SMs. However, if the algorithm observes a notable throughput change, it will begin to re-adjust the V/F levels, the number of the CPU cores, and the GPU SMs.

6.1.2 Evaluation of Runtime DVFCS Algorithm

Fig. 9 plots the throughput from our cycle-level SCHP simulator for all the benchmarks based on four different configurations: (i) the GPU-oriented configuration, (ii) the configuration selected by an exhaustive search method (denoted by "Optimal DVFCS"), (iii) the configuration selected by running the proposed runtime DVFCS algorithm until it converges (denoted by "Runtime DVFCS"), and (iv) the configuration selected by running DVFCS for only three iterations (denoted by "Runtime DVFCS-3I"). The optimized configurations selected by DVFCS leads to 12 percent higher throughput on average than the GPU-oriented configuration, while the optimal configurations after the exhaustive search is on average 13 percent higher; only the results for BPR and SRAD using the proposed runtime DVFCS algorithm are slightly worse than using the exhaustive search.

Our experiment shows that our runtime DVFCS algorithm takes five to eight iterations (6.8 iterations on average)

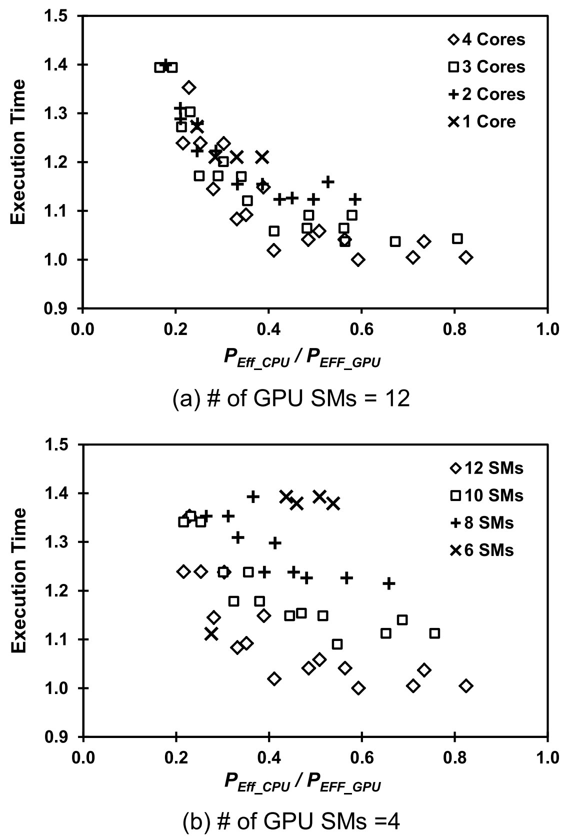

Fig. 8. HotSpot's execution time versus power efficiency ratio for different numbers of CPU cores and GPU SMs.

to achieve the optimal DVFCS configuration for eight of the 12 workloads. Moreover, Fig. 9 shows that significant throughput improvements are achieved after just three iterations, with an average throughput improvement of 9.1 percent. This indicates that the overhead of the runtime algorithm is negligible considering that many GPGPU applications execute the same kernels many times for the same size input set; for example, HotSpot and Heartwall repeat the execution of the same kernel(s) for a specified time window and a specified number of frames, respectively.

6.2 Multi-Programmed Workloads

The results in Section 5 support our argument that workload-aware, asymmetric power allocation can significantly improve the throughput and power efficiency for multi-programmed workloads. Thus, we describe runtime algorithms

TABLE 7
Search Results with Runtime Algorithm 2

P1	P2	Metric 1: IPS			Metric 2: IPS/Watt		
		Predicted	Actual	Miss Penalty (%)	Destination	Actual	# of Iter.
BFS (C)	KMN (M)	(L6, L4)	(L6, L4)	-	(L3, L2)	(L3, L2)	4
BFS (C)	HOT (M)	(L5, L5)	(L6, L4)	1.17	(L3, L2)	(L3, L2)	4
BFS (C)	NW (M)	(L6, L4)	(L6, L4)	-	(L3, L3)	(L3, L3)	4
HWT (C)	KMN (M)	(L6, L4)	(L6, L4)	-	(L3, L2)	(L3, L2)	4
HWT (C)	HOT (M)	(L6, L4)	(L6, L4)	-	(L3, L2)	(L3, L2)	4
HWT (C)	NW (M)	(L6, L4)	(L6, L4)	-	(L3, L3)	(L3, L3)	5
SRAD (C)	KMN (M)	(L6, L4)	(L6, L4)	-	(L3, L2)	(L3, L2)	4
SRAD (C)	HOT (M)	(L6, L4)	(L6, L4)	-	(L3, L2)	(L3, L2)	4
SRAD (C)	NW (M)	(L5, L5)	(L6, L4)	0.63	(L3, L3)	(L3, L3)	5
BFS (C)	HWT (C)	(L6, L4)	(L6, L4)	-	(L3, L3)	(L3, L3)	5
BFS (C)	SRAD (C)	(L6, L4)	(L6, L4)	-	(L3, L3)	(L3, L3)	5
HWT (C)	SRAD (C)	(L5, L5)	(L6, L4)	1.00	(L3, L3)	(L3, L3)	5
KMN (M)	HOT (M)	(L5, L5)	(L6, L4)	1.10	(L2, L2)	(L2, L2)	3
KMN (M)	NW (M)	(L4, L6)	(L4, L6)	-	(L2, L3)	(L2, L3)	4
NW (M)	HOT (M)	(L6, L4)	(L6, L4)	-	(L2, L3)	(L2, L3)	4
Average-	-	-	-	0.98	-	-	4.27

based on heuristics that find an optimal power configuration efficiently to maximize IPS and IPS/Watt, respectively. Experimental results with the 15 multi-programmed workloads are summarized in Table 7. When searching for an optimal IPS/Watt point, we assume the threshold value of 0.001 IPS/Watt and an initial configuration of the lowest V/F point.

Algorithm to optimize IPS. According to our experiments, an optimal configuration is always found on the perimeter of the configuration space. More specifically, there are only three distinct optimal points for all 15 workloads represented by the following pairs of GPU power levels for the two programs P1 and P2: (GPU_{P1}, GPU_{P2}) = (L6, L4), (L5, L5), and (L4, L6), where L6 indicates the highest V/F level. Depending on (1) which program is more compute-bound and (2) how much different the two programs' throughput sensitivities to the allocated power budget, the system may allocate more power to one program or the other; if their difference in throughput sensitivities is smaller than a threshold, both programs receive the same power budget (of Level 5 since both programs cannot be on Level 6 within the chip power budget). Note that the CPU power is always set to Level 1 (i.e., the lowest V/F level) since our workloads favor the GPU over the CPU.

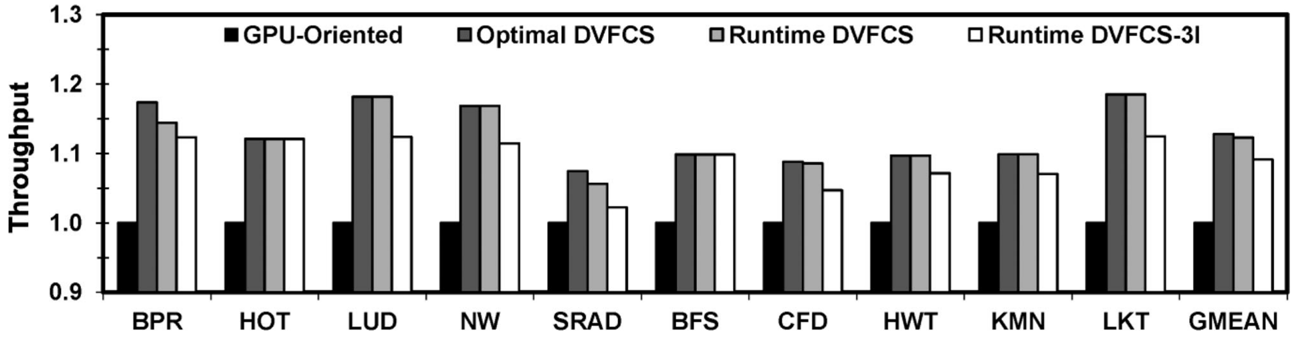

Fig. 9. Normalized throughput comparisons using the detailed cycle-level simulator.

```
while (every regular time intervals)

IPS_MIN    = get_IPS(P1_CPU_L1,P1_GPU_L1,
                     P2_CPU_L1, P2_GPU_L1)
IPS_P1_MAX = get_IPS(P1_CPU_L1,P1_GPU_L6,
                     P2_CPU_L1, P2_GPU_L1)
IPS_P2_MAX = get_IPS(P1_CPU_L1,P1_GPU_L1,
                     P2_CPU_L1, P2_GPU_L6)

SLOPE_P1 = cal_slope (IPS_MIN, IPS_P1_MAX)
SLOPE_P2 = cal_slope (IPS_MIN, IPS_P2_MAX)

if (|SLOPE_P1-SLOPE_P2|) < (THRESHOLD)
    set_IPS(P1_CPU_L1,P1_GPU_L5,P2_CPU_L1,P2_GPU_L5)
else if (SLOPE_P1 > SLOPE_P2)
    set_IPS(P1_CPU_L1,P1_GPU_L6,P2_CPU_L1,P2_GPU_L4)
else
    set_IPS(P1_CPU_L1,P1_GPU_L4,P2_CPU_L1,P2_GPU_L6)

end while
```

(a)

```
P1_CPU_FINAL = P1_CPU = P1_CPU_L1
P1_GPU_FINAL = P1_GPU = P1_GPU_L1
P2_CPU_FINAL = P2_CPU = P2_CPU_L1
P2_GPU_FINAL = P2_GPU = P2_GPU_L1
IPSW_BASE = get_IPSW(P1_CPU,P1_GPU,P2_CPU,P2_GPU)

while (1)
  P1_GPU = P1_GPU_FINAL + 1 level
  P2_GPU = P2_GPU_FINAL
  IPSW_P1 = get_IPSW (P1_CPU,P1_GPU,P1_CPU,P2_GPU)

  P1_GPU = P1_GPU_FINAL
  P2_GPU = P2_GPU_FINAL + 1 level
  IPSW_P2 = get_IPSW (P1_CPU,P1_GPU,P1_CPU,P2_GPU)

  if (IPSW_BASE >= IPSW_P1 &&IPSW_BASE >= IPSW_P2)
    exit
  else if (IPSW_P1 >= IPSW_P2)
    IPSW_BASE = IPSW_P1
    P1_GPU_FINAL = P1_GPU_FINAL + 1 level
  else
    IPSW_BASE = IPSW_P2
    P1_GPU_FINAL = P1_GPU_FINAL
end while
```

(b)

Algorithm 2. Runtime algorithms for multi-programmed workloads: (a) IPS (b) IPS/Watt.

Algorihm 2a presents an optimization algorithm based on runtime performance monitoring. As in Fig. 7, the runtime monitor measures the slopes of performance improvement when the power budget for GPU increases from L1 to L6 for both programs. These slopes are denoted by SLOPE_P1 and SLOPE_P2 for the two programs P1 and P2, respectively. If their difference is greater than a specific threshold (denoted by THRESHOLD), the algorithm allocates more power budget to the program that has a higher slope. This algorithm can be generalized to deal with multiple, Pareto optimal points along the perimeter of the configuration space by considering the amount of difference between SLOPE_P1 and SLOPE_P2. Although the algorithm finds an optimal configuration only once at program invocation, it can be also extended to deal with the phase behavior of a workload over time. We can achieve this by monitoring IPS periodically and triggering recalibration as needed, for example. Columns 3-5 of Table 7 show the search results. The 11 of 15 workloads reach the optimal points correctly. For the remaining four non-optimal cases, average IPS loss is only 0.98 percent.

Algorithm to optimize IPS/Watt. In many cases, the optimal power configuration for IPS/Watt is placed near the minimum power configuration (i.e., the case of all L1's). Therefore, Algorithm 2b implements a simple gradient search algorithm. Again, as GPU power allocation dominantly determines the program's throughput, we only search for an optimal configuration of GPU power while CPU is put to the lowest V/F level. The algorithm starts searching at the minimum power configuration and the measure power efficiency of the next power level for both P1 and P2. If one of the two yields higher power efficiency, the system moves to that configuration. The optimization loop continues iteration until all of the neighboring configuration points have lower energy efficiencies than the current configuration. Columns 6–8 of Table 7 show the results of searching optimal IPS/Watt points for 15 workloads. This algorithm takes three to five iterations with an average of 4.27 iterations to reach the true optimal power configuration for IPS/Watt with 100 percent accuracy.

7 Related Work

Luk et al. proposed a technique that adaptively partitions a given workload between the CPU and the GPU in a heterogeneous computing system to improve the overall throughput of a heterogeneous computing system [8]. The execution times of a given kernel on the CPU and GPU are estimated based on a performance model trained by a profiling method and the workload is partitioned between the CPU and the GPU to minimize the total execution time. Wang and Ren evaluated a workload partitioning algorithm to improve the overall power efficiency of a heterogeneous computing system based on an Intel CPU and an AMD GPU [9]. Unlike our study, in these two studies the target heterogeneous computing system is comprised of a CPU and a GPU that are two separate chips. Therefore, sharing the power budget between the CPU and the GPU was not considered.

Jeong et al. proposed an effective partitioning scheme of DRAM bandwidth to optimize a pair of co-scheduled programs running on the CPU and the GPU, respectively [25]. The target workload contains a program with real-time constraints placed on the GPU. By dynamically adjusting the priorities to access DRAM, the SCHP meets the real-time constraints for the GPU program, and subject to that, maximizes the throughput of the CPU program. Kim et al. also co-optimized CPU and GPU programs on an SCHP platform integrated with hybrid DRAM/PRAM-based main memory [26]. Adriaens et al. advocate spatial multitasking on GPU to allow multiple programs to run simultaneously and maximize resource utilization [24]. Unlike our work their proposal is to partition SMs on a single, discrete GPU. Therefore, they do not take into account complex tradeoffs in power budget allocation between the CPU and the GPU on SCHP. There are recent proposals to support preemptive multitasking on GPUs [27], [28] to execute multiple kernels concurrently via traditional time sharing. Their primary goal is to improve the responsiveness of high-priority kernels and quality-of-service (QoS). In contrast, this work aims to maximize the system-wide throughput and power efficiency by jointly optimizing power and workload allocation for both CPU and GPU.

Li and Martinez proposed a runtime DVFCS algorithm to minimize the power consumption of a homogeneous multicore CPU under a performance constraint [14]. While producing near-optimal solutions efficiently via search space pruning, their algorithm does not take into account the heterogeneity of processing elements and complex power budget constraints imposed by modern SCHPs. Lee et al. proposed a runtime algorithm that scales (i) the number of active SMs and (ii) the V/F of both SMs and on-chip interconnects/caches to maximize the throughput of a homogeneous GPU under a power constraint [13]. Again, their work targets the homogeneous GPU, but not SCHPs.

We target an SCHP and maximize the throughput under a power constraint by exploiting different performance/power trade-offs between the CPU and GPU. Furthermore, our novel runtime algorithms improve the effectiveness of the workload partitioning technique by jointly applying a DVFCS technique to both the CPU and GPU. And, we support optimization for multi-programmed workload by taking into account per-program characteristics and evaluation metrics. This allows the SCHP to utilize the power budget more efficiently and flexibly, to achieve higher performance for the same power budget.

8 Conclusion

In this paper, we consider an SCHP, in which CPU cores and GPU SM groups have independent V/F domains and CPU cores and GPU SMs have independent PG devices, respectively. Since the CPU and GPU are on the same chip, they share the chip power budget while each must satisfy its own power constraint. Compared with the SCHP configuration operating at the nominal V/F levels, we first demonstrate that jointly optimizing workload and power partitioning between the CPU and the GPU can improve the throughput of *single-programmed* workloads by 13 percent. Second, we show that allocating power by considering (i) per-program characteristics and (ii) evaluation metrics improves the throughput (IPS) and power efficiency (IPS/Watt) of *multi-programmed* workloads by 9 and 24 percent, respectively, compared with the baseline configuration, which allocates a fair share of hardware and power resources to each program. Last, we propose effective runtime algorithms for both single and multi-programmed workloads; our algorithms can find an optimal or near-optimal configuration within 5-8 iterations for single programmed workload and within 3-5 iterations for multi-programmed workloads, to achieve performance (both throughput and throughput/Watt) within 10 percent of that of the exhaustive search. While these algorithms are demonstrated to work well for regular, long-running data-parallel kernels, further research is necessary to handle effectively short and/or irregular programs (e.g., with complex phase behaviors), which we leave for future work.

Acknowledgments

This work is supported by the Ministry of Science, ICT & Future Planning (MSIP) under the "IT R&D program of MSIP/KEIT" (KI001810041244), "Research Project on High Performance and Scalable Manycore Operating System" (#14-824-09-011), and "Basic Science Research Program" (NRF-2014R1A1A1005894), and US National Science Foundation (NSF) CNS-1217102. Jae W. Lee is the corresponding author of this paper.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. IEEE Int. Symp. Comput. Arch.*, 2011, pp. 365–376.
[2] S. Nussbaum, "AMD trinity fusion APU," in *Proc. IEEE/ACM Symp. High Perform. Chips*, 2012, pp. 283–322.
[3] Qualcomm, Snapdragon MDP MSM8660 datasheet [Online]. Available: https://developer.qualcomm.com/, 2011.
[4] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2011, pp. 264–266.
[5] M. Linderman, J. Collins, H. Wang, and T. H. Meng, "Merge: A programming model for heterogeneous multi-core systems," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 287–296.
[6] P. Wang, J. Collins, G. Chinya, H. Jiang, X. Tian, M. Girkar, N. Yang, G. Lueh, and H. Wang, "EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 156–166.
[7] H. Wang, V. Sathish, R. Singh, M. Schulte, and N. Kim, "Workload and power budge partitioning for single-chip heterogeneous processors," in *Proc. IEEE/ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 401–410.
[8] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 45–55.
[9] G. Wang and X. Ren, "Power-efficient work distribution method for CPU-GPU heterogeneous system," in *Proc. IEEE Parallel Distrib. Process. Appl.*, 2010, pp. 122–129.
[10] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, Mar-Apr. 2012.
[11] D. Foley, M. Steinman, B. Branover, G. Smaus, A. Asaro, S. Punyamurtula, and L. Bajic, "AMD'S Llano fusion APU," in *Proc. IEEE/ACM Symp. High Perform. Chips*, 2011, pp. 1–36.
[12] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. IEEE Int. Symp. Microarchit.*, 2006, pp. 347–358.
[13] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. Kim, "Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling," in *Proc. IEEE/ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 111–120.
[14] J. Li and J. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *Proc. IEEE/ACM Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 77–87.
[15] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, and M. Shoaib, "The gem5 simulator," *ACM SIGARCH Comput. Arch. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[16] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads using a Detailed GPU Simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Boston, MA, USA, 2009, pp. 163–174.
[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. IEEE Int. Symp. Comput. Archit.*, 2000, pp. 128–138.
[18] NVIDIA [Online]. Available: http://www.geforce.com/hardware/notebook-gpus/geforce-gts-260m/specifications
[19] C. Shuai, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2009, pp. 44--54.
[20] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. IEEE Int. Symp. Microarchit.*, 2009, pp. 469–480.
[21] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proc. IEEE/ACM Int. Symp. Comput. Archit.*, 2010, pp. 280–289.
[22] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm early design exploration," *IEEE T. Electron. Devices*, vol. 53, no. 11, pp. 2816–2823, Nov. 2006.
[23] Apple iPad Air 2 [Online]. Available: http://www.apple.com/ipad-air-2, 2014.

[24] J. Adriaens, K. Compton, N. Kim, and M. Schulte, "The case for GPGPU spatial multitasking," in *Proc. IEEE/ACM Int. Symp. High-Perform. Comput. Archit.*, 2012, pp. 1–12.

[25] M. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Proc. IEEE/ACM Des. Autom. Conf.*, 2012, pp. 850–855.

[26] D. Kim, S. Lee, J. Chung, D. Kim, D. Woo, S. Yoo, and S. Lee, "Hybrid DRAM/PRAM-based main memory for single-chip CPU/GPU," in *Proc. IEEE/ACM Des. Autom. Conf.*, 2012, pp. 888–896.

[27] J. J. K. Park, Y. Park, and S. A. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 593–606.

[28] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. IEEE Int. Symp. Comput. Archit.*, 2014, pp. 193–204.

[29] V. Sathish, M. Schulte, and N. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *Proc. IEEE/ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 325–334.

[30] J. Nickolls and W. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar.-Apr. 2010.

[31] Khronos Group, OpenCL Standard [Online]. Available: http://www.khronos.org/opencl/, 2008.

Jae Young Jang received the BS degree in electronics engineering from Sungkyunkwan University (SKKU), Korea, in 2012. He is currently a MS/PhD student in the Department of Electrical and Computer Engineering at the same university. His research interests include heterogeneous architectures, parallel programming, and Java Virtual Machine (JVM) optimization.

Hao Wang received the BS degree in microelectronics from Peking University. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research focuses on heterogeneous processors and memory systems.

Euijin Kwon received the BS degree in electrical engineering from Inha University, Korea, in 2004 and the MS degree in the Department of Semiconductor and Display Engineering at Sungkyunkwan University (SKKU) in 2014. He has been with Samsung Electronics since 2004, where he is a senior engineer working on SoC design and foundry business projects. His research interests include computer architecture and SoC design.

Jae W. Lee received the BS degree in electrical engineering from Seoul National University, the MS degree in electrical engineering from Stanford University, and the PhD degree in computer science from the Massachusetts Institute of Technology (MIT) in 2009. He is currently an assistant professor in the Department of Semiconductor Systems Engineering at Sungkyunkwan University (SKKU), Korea. His research areas include computer architecture, compilers, parallel programming, and computer security.

Nam Sung Kim received the bachelor's and master's degrees in electrical engineering from the Korea Advanced Institute of Science and Technology, and the PhD degree in computer science and engineering from the University of Michigan-Ann Arbor. He is an associate professor at the University ofIllinois-Urbana-Champaign. He has been conducting interdisciplinary research that cuts across device, circuit, and architecture for power-efficient computing. Prior to joining Illinois-Urbana-Champaign, he was with the University of Wisconsin-Madison from 2008 to 2015, and a senior research scientist with Intel from 2004 to 2008. He has published more than 120 refereed articles in highly-selective conferences and journals in the field of digital circuit, processor architecture, and computer-aided design. His top five most frequently cited papers have more than 2,800 combined citations and the total number of combined citations for all his papers exceeds 5,000. He has also served several prominent international conferences and a journal such as IEEE/ACM International Symposium on Computer Architecture (ISCA), IEEE/ACM International Symposium on Computer Microarchitecture (MICRO), IEEE/ACM International Symposium on High-Performance Computer Architecture (HPCA), IEEE/ACM Design Automation Conference (DAC), and *ACM Transactions on Design Automation of Electronic Systems* as a technical program committee and an associate editor. He received the IEEE Design Automation Conference (DAC) Student Design Contest Award in 2001, Intel Fellowship in 2002, IEEE International Symposium on Microarchitecture (MICRO) Best Paper Award in 2003, US National Science Foundation (NSF) CAREER Award in 2010, and IBM Faculty Award in 2011 and 2012, and the University of Wisconsin Villas Associates Award in 2015. He is a senior member of the IEEE.