Co-Run Scheduling with Power Cap on Integrated CPU-GPU Systems

Qi Zhu[‡] Bo Wu[§] Xipeng Shen[*] Li Shen[‡] Zhiying Wang[‡]
[‡] National University of Defense Technology, China
[§] Colorado School of Mines, USA
[*] North Carolina State University, USA

***Abstract*—This paper presents the first systematic study on co-scheduling independent jobs on integrated CPU-GPU systems with power caps considered. It reveals the performance degradations caused by the co-run contentions at the levels of both memory and power. It then examines the problem of using job co-scheduling to alleviate the degradations in this less understood scenario. It offers several algorithms and a lightweight co-run performance and power predictive model for computing the performance bounds of the optimal co-schedules and finding appropriate schedules. Results show that the method can efficiently find co-schedules that significantly improve the system throughput (9-46% on average over the default schedules).**

I. Introduction

Recent processor development exhibits a trend towards integrated CPU-GPU architectures, exemplified by AMD Fusion [5], Intel Ivy Bridge [13], NVIDIA Denver [4], and so on. Such an integration is a double-edged sword. As CPU and GPU are integrated into a single chip, the communication latency between them is shortened. But at the same time, the tight sharing of hardware resource makes co-run interference between GPU and CPU programs become more intensified and complicated.

In this work, we aim at finding solutions to minimize the influence of such co-run performance interference through job co-scheduling. Job co-scheduling [10, 16, 23, 26] refers to a technique which schedules a batch of jobs to processors in a time sequence such that a certain objective (minimizing co-run interference, maximizing throughput, etc.) can be achieved. As a software approach, job co-scheduling offers a cheap (virtually free) way to significantly improve system throughput for shared servers, workstation clusters, and data centers [22, 25, 29].

Job co-scheduling has drawn many research interests. However, most prior efforts are focused on homogeneous multicore processors [10, 15, 16]. There are some co-scheduling studies for heterogeneous systems, but what they concern are the fair or efficient usage of the discrete GPU shared by co-running applications [8, 11, 18, 28], or mapping tasks in a single application to CPU or GPU [3, 17], rather than dealing with the complex effects of interferences between CPU and GPU programs co-running on such integrated systems.

Compared to those studies, the problems tackled in this work have the following extra complexities.

- **Job placement:** The same program may show dramatic performance differences between its CPU and GPU executions depending on the program's characteristics. In an integrated system, the GPU execution may not always outperform the CPU execution even for data parallel programs, because the difference between peak throughput of the CPU and GPU is substantially smaller than that in discrete systems.
- **Power cap:** For energy efficiency and reliability, modern processors feature power caps—the allowed upper limit of power consumption by the processor. While this feature exists on CPUs, the integrated architecture needs to allocate power to two different types of processors, which have disparate power profiles. They appear to be more sensitive to power cap in terms of both performance and reliability. Prior job co-scheduling works have not systematically considered the influence of power cap in co-scheduling.
- **Interplay:** The memory contention due to tight integration may substantially slow down the co-running applications. The degree of degradation depends on not only the applications' memory access patterns but also the power allocation. Job lengths further complicates the problem, as a long job may need to co-run with a sequence of short jobs and the lengths of a job vary along with the power allocation and memory contention.

As prior co-scheduling methods have not considered these special complexities, they are not applicable to CPU-GPU co-runs on our target processors. We are not aware of any prior co-scheduling work that has systematically considered all these complexities on integrated heterogeneous processors. To solve the co-scheduling problem, one must answer two questions. The first is how to design a scheduling algorithm that can efficiently compute a good co-schedule assuming all co-run performance info of all jobs is available. The second is how to effectively estimate the co-run performance of jobs with all the complexities considered. Neither question has been systematically explored before in the integrated heterogeneous co-scheduling environment.

This paper answers both questions. After Sections II and III give a formal definition and an example of the optimal co-scheduling problem on CPU-GPU integrated environment, Section IV presents a set of developments in co-scheduling algorithm designs. Prompted by the NP-hard complexity of the co-scheduling problem, it presents an efficient heuristic co-scheduling algorithm that builds on a *Co-Run Theorem*. It further presents a simple way to compute the performance bounds of the optimal co-schedules, which offer some references for assessing the gap between a heuristic co-schedule and the full potential of job co-scheduling.

Section V presents our solution to the second problem: enabling efficient prediction of the performance degradation and power consumption of the co-runs of an arbitrary pair of programs in a given set of workload on a CPU-GPU integrated processor with a certain frequency setting. A naive approach is to find out the performance of all possible co-run settings of the workload by profiling each group of the programs under each frequency setting. Given N programs and K frequency levels for each processor, for co-runs of two jobs, it would need many $(O(N^2 \times K^2))$ profiling runs. Our work addresses the issue through a *staged interpolation* method, which builds a co-run performance predictive model. The method uses controllable micro-kernels to first characterize the co-run performance space, through which, it predicts the co-run performance of two jobs by interpolations with their standalone performance.

We integrate the techniques into a prototype co-scheduling runtime. The performance degradation model shows high accuracy for 64 co-run pairs with predicted co-run performance only 15% on average from the ground truth. In a 8-program co-run case study, the runtime determines a co-schedule that produces 41% performance improvement over random scheduling and about 9% improvement over the system's default schedule. In a 16-program co-run case study, the improvement is 37% and 46% over the random and default schedules respectively.

Contributions: This work makes the following major contributions:

- To our best knowledge, this is the first systematic study on maximizing the throughput of CPU-GPU integrated systems through job co-scheduling with power cap considered.
- It proposes the first set of algorithms to efficiently compute the performance bounds of optimal co-schedules and search for appropriate co-schedules on such integrated systems.
- It devises a micro-benchmark to characterize the co-run performance degradation space and uses staged interpolation to accurately predict co-run performance for co-scheduling.
- It evaluates the integrated runtime with the performance models and heuristic scheduling on a set of experiments, which demonstrate significant performance benefits.

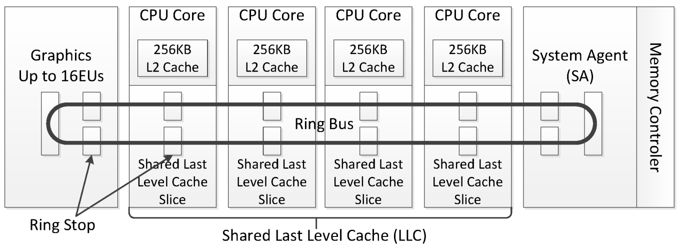

Figure 1. The structure of Ivy Bridge Processor.

II. Problem Definition and Scope

In this paper, we focus on integrated architectures that have CPU and GPU put on the same die. It represents a trend to seamlessly take advantage of both types of processors. Figure 1 depicts an example of such architectures, Intel's Ivy Bridge. The CPU and GPU on an Ivy Bridge processor share most of the memory system, including the last-level cache, on-chip network, and main memory.

We next give a formal definition of the *optimal job co-scheduling* problem tackled in this work:

Definition 2.1: In a heterogeneous processor consisting of two types of units A and B, there are a batch of independent jobs J (of possibly different lengths) that need to run. C is a given power cap. The core workload in a job may run either on A or B. The goal of the optimal job co-scheduling is to put the jobs in J into two mutually exclusive sequences, with each corresponding to the execution order of jobs on one of the two types of processors, and associate each job with a frequency level, such that the following conditions are met:

1. At any moment, the power consumption on the heterogeneous processor is no greater than the given power cap C.

2. The total time spanning from the start of the first job of J to the finish of last job of J is minimized. The time is commonly called *makespan*.

Several aspects of the definition are worth explaining. First, the defined scope assumes that each job can run on either type of processors. This is the most general scenario, corresponding to the cases when all the jobs are written in some portable language. Our study uses OpenCL [2] programs. OpenCL is a programming model supported on multiple types of platforms, for both CPU and GPU. An OpenCL program usually contains one or more OpenCL kernels that compose the core computations of the program. An OpenCL kernel is portable, runnable on both CPU and GPU.

The other situations, in which some jobs are runnable only on one type of processor, can be regarded as special cases of this general situation. In those situations, the placement of the jobs on the processors becomes straightforward, while the general case must deal with the extra complexity from the choices of job placement when searching for the best schedules.

Second, the defined scope assumes that the core workload of a job (e.g., OpenCL kernels) runs on one of the processors. It is possible that a job contains multiple kernels—including the cases in which a single kernel is split into multiple ones for them to run concurrently on both types of processors—and the scheduler could schedule some to CPU, others to GPU. In this work, we limit the scope of our schedule to an entire job (i.e., the collection of all its kernels) for two reasons. First, in practical workloads, it is not yet a common practice to partition the OpenCL kernel of a program to make part run on CPU and part on GPU simultaneously. In fact, previous work [31] shows that due to the complexity in data partitioning and communications, such partitioning often yields even worse performance than using a single processor. Second, the fine-grained kernel-level scheduling requires the source code of the program to be written in a certain form. Most programs in practice are not in such a form. We hence limit our scope of scheduling to the entire job, but acknowledge that the fine-grained direction is worth future explorations. We note that when a job is scheduled to the GPU, it means that the OpenCL kernels in that job will run on the GPU; its host thread still runs on the CPU.

Finally, it is possible that multiple jobs co-run together simultaneously on the CPU. Our problem definition excludes such schedules from considerations, for two reasons. First, on existing integrated processors, such co-runs have shown to usually result in large performance degradations to both CPU and GPU workload due to the much intensified resource contention. Second, such co-runs are helpful usually when there are no enough parallelism in a single job. Given that OpenCL is intended for massively parallel computations, such co-runs of OpenCL programs are typically unbeneficial.

As per the problem definition, the objective of our schedule is to minimize the makespan. Given a set of jobs, because the set of computations are fixed, that objective is equivalent to maximizing the amount of computations per time unit—or throughput of the system.

III. A Simple Example

We next use a simple example to convey the intuition of the job co-scheduling problem, give a glimpse at its potential benefits, and discuss the two key questions to answer for co-scheduling.

We run four OpenCL programs (streamcluster, cfd, dwt2d, and hotspot) on a system that hosts an Intel Ivy Bridge integrated processor. The processor has 4 CPU cores and 1 integrated GPU, with both types of cores being capable of executing OpenCL programs. Figure 2 shows the performance difference when running each application alone on the CPU or GPU. Among the four programs, streamcluster,

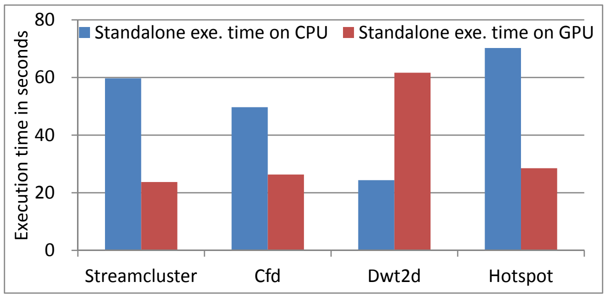

Figure 2. The standalone performance of programs on CPU and on GPU.

cfd and hotspot prefer to run on the GPU and demonstrate 2.5X, 1.8X and 2.4X performance improvement over their runs on the CPU, respectively. While dwt2d prefers to run on the CPU, with 2.5X performance improvement over its run on the GPU. We co-run dwt2d on the CPU and streamcluster on the GPU, and observe 81% and 5% slowdown compared to the standalone runs for dwt2d and streamcluster, respectively. But if we co-run dwt2d on the CPU and hotspot on the GPU, the slowdown is only around 17% for dwt2d and 5% for hotspot. The results demonstrate that the significance of pairing the applications for co-running. Finally, we set a power cap as 15 watts, and enumerate all frequency settings for both processors that satisfy the power cap, we observe that the optimal setting yields performance 2.3X better than the worst case co-schedule of the four programs.

The results show three factors that play important roles in the co-scheduling: program-to-processor mapping, program co-run mapping, and frequency selections. To find appropriate settings to these factors, two key questions must be answered. The first is to quickly estimate how well two programs co-run together at an arbitrary processor frequency setting. The second is how, based on the estimated co-run performance, to find a way to quickly go through the potential candidate schedules, assess their quality, and then choose the best. The challenge is that the search space can be enormous. Even for just four programs, the search space contains $C_4^2 * C_2^1 * 10 * 16 = 1920$ (10 frequency levels for GPU, and 16 frequency levels for CPU) possible co-schedules. To make the co-scheduling technique broadly applicable, it is ideal to take effect online. The stringent runtime overhead tolerance makes the problem even more difficult.

IV. Algorithm Design

The next section will answer the questions on co-run performance prediction. This section provides our findings in the algorithmic dimension of the problem. So the discussions in this section assume the availability of accurate co-run performance and power models at each frequency level.

On homogeneous multicore CPU, some work [26] has proved that finding the optimal co-schedule is NP-complete in general. As homogeneous systems can be regarded as a special case of the optimal co-scheduling problem in Definition 2.1 (with a fixed processor frequency and identical processing units), our optimal co-scheduling problem is NP-hard, which implies that finding optimal co-schedules is in general infeasible to be done in polynomial time (unless NP=P). We hence design a heuristic algorithm to find good (not necessarily optimal) co-schedules. To help assess the distance from the optimal, we further present a simple way to compute the lower bound of the makespan of optimal schedules. This section presents these algorithmic contributions.

A. Heuristic Algorithm

Before describing our algorithm, we first list some notations we use as follows:

J:	the set of jobs to schedule
l_i:	the standalone run time of job i when the system is homogeneous and allows no frequency adjustment
$l_{i,p,f}$:	the standalone run time of job i on processor p with power frequency f
d_i^j:	the co-run degradation percentage of job i when job j runs on the other processor in a homogeneous system allowing no frequency adjustment
$d_{i,p,f}^{j,g}$:	the co-run degradation percentage of job i on processor p with frequency at f when job j runs on the other processor with frequency at g
T_{low}:	the lower bound of the time makespan of the executions of all the jobs

In this section, J, l_i, $l_{i,p,f}$, d_i^j, and $d_{i,p,f}^{j,g}$ are all assumed known variables (next section discusses the attainment of their values). For the convenience of discussion, sometimes we call the CPU and GPU each as a processor in the following descriptions.

The design of our algorithm leverages a *Co-Run Theorem* that we propose as follows. The theorem helps determine whether a job should be arranged to run alone (i.e., leaving the other type of processor idle while it is running) as such arrangements are also permitted in the co-schedule if they help reduce the makespan. Although the theorem is for homogeneous systems, its extension applies to heterogeneous systems well.

Co-Run Theorem: For two jobs W_1 and W_2, on a homogeneous system, suppose their standalone lengths are l_1 and l_2, co-run lengths are $l_1 + l_1 \times d_1^2$ and $l_2 + l_2 \times d_2^1$, and $l_1 + l_1 \times d_1^2 \geq l_2 + l_2 \times d_2^1$. Then if and only if $l_1 \times d_1^2 < l_2$, the co-run produces a higher throughput than the sequential executions of the two jobs.

The theorem can be easily proved as follows: The makespan of the co-run is $T_c = l_1 + l_1 \times d_1^2$. The makespan of the sequential executions is $T_s = l_1 + l_2$. Apparently, $(l_1 \times d_1^2 < l_2) \equiv (T_c < T_s)$.

We next describe the heuristic co-scheduling algorithm. To ease the understanding, we start with a basic design without considering power cap. It shows how the Co-Run Theorem helps with co-scheduling. We then explain the changes to the basic design to take power cap into considerations. Finally, we describe several ways to do post refinement of the produced co-schedule.

A.1 For Heterogeneous Systems without Power Cap

This part considers the integrated systems but with no power cap; both processors may hence run at their highest frequency levels. The algorithm is illustrated in Figure 3. It consists of the following three steps.

Step 1: Job partition based on the Co-Run Theorem. The goal of this step is to partition J into two disjoint sets S_{co} and S_{seq}. The jobs in S_{co} can potentially benefit from co-runs, while the jobs in S_{seq} should run in a standalone manner for the best performance. To determine whether a job should join S_{co} or S_{seq}, we pair each other job with it and use the co-run theorem to dictate whether the two jobs should co-run or not. If the evaluation from co-run theorem always gives a negative answer, this job should join S_{seq}. Otherwise, the algorithm puts it into S_{co}.

Step 2: Job categorization. Taking processor heterogeneity into consideration, the algorithm divides the jobs in S into three sets: *CPU-preferred*, *GPU-preferred*, and *non-preferred*. For each job, the algorithm includes it in *non-preferred* if the difference between its execution times on the CPU and GPU is smaller than or equal to a threshold D (empirically selected as 20%). When the difference is larger than D, the job should join the set preferred by the processor, on which it has better performance.

Step 3: Greedy scheduling. The algorithm first schedules jobs in S by following a rule to respect processor preference. To schedule jobs to a processor, the algorithm always picks jobs from its preferred set if that set is not empty, followed by jobs in the non-preferred set and at last the jobs from the set that prefer to run on the other processor. At the beginning of the scheduling, the algorithm picks the longest job in the GPU-preferred set and schedules it to the GPU. It then picks the job from CPU-preferred set with the least co-run interference (determined by the sum of the co-run degradation percentages) with the job on the GPU and run the picked job on the CPU. When a job finishes, the algorithm follows the scheduling rule and picks a job with the smallest co-run degradation to the running job until all three sets are empty. The algorithm then sequentially executes each job in S_{seq} on the processor that delivers the best performance to avoid co-runs.

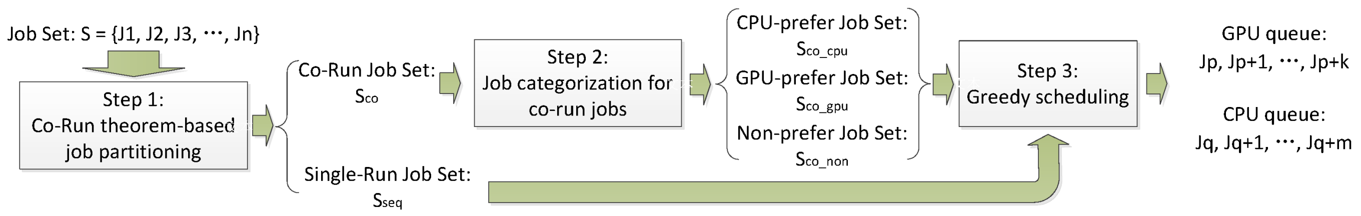
Job Set: S = {J1, J2, J3, ⋯, Jn}
Step 1: Co-Run theorem-based job partitioning
Co-Run Job Set: Sco
Single-Run Job Set: Sseq
Step 2: Job categorization for co-run jobs
CPU-prefer Job Set: Sco_cpu
GPU-prefer Job Set: Sco_gpu
Non-prefer Job Set: Sco_non
Step 3: Greedy scheduling
GPU queue: Jp, Jp+1, ⋯, Jp+k
CPU queue: Jq, Jq+1, ⋯, Jq+m

Figure 3. The flow diagram of heuristic algorithm.

A.2 With Power Cap Considered

In this case, the algorithm should select frequencies for both processors such that the aggregate power consumption does not exceed the power cap. We make the following changes to the basic algorithm described in the previous section:

Changes in step 1: When the algorithm leverages the co-run theorem to dictate whether a job can benefit from the co-run with another job, it traverses all possible frequency settings that satisfy the power cap requirement.

Changes in step 2: To compare the performance difference between CPU and GPU runs of a job, the algorithm uses the execution time of the job at the highest frequency allowed by the power cap.

Changes in step 3: Like in step 1, to determine the co-run interference for two jobs, the algorithm traverses all frequency settings allowed by the power cap to compute the minimal degradation.

A.3 Post Local Refinement

We further propose a 3-step local refinements to further enhance the co-schedules produced by the heuristic algorithm with low cost. First, it tries to swap every two adjacent jobs in a processor to see whether the swapping helps reduce the makespan. If so, keep the new order and continue to try on the next two adjacent jobs until reaching the last job on that processor. Then try the same refinement on the job list of the other processor. Second, it tries to swap two randomly picked jobs from the job list assigned to a processor. Finally, it tries to swap two jobs on two different processors. The complexities of the steps are all linear, either to the number of jobs, or to the number of random samples. They offer some quick refinements to the results from the heuristic algorithm.

B. Lower Bound

In this part, we present a simple way to compute the lower bound of the makespan of the optimal schedules. The lower bounds, although they are not sophisticatedly computed to be the tightest, offer some reference points for estimating the room that a heuristic schedule has left from the optimal. It is shown in the following formula:

$$T_{low} = \frac{1}{2}\sum_i l'_i$$

where, for each processor p,

$$l'_{i,p} = \begin{cases} \min_{j,f,g}(l_{i,p,f} + d^{j,g}_{i,p,f}); \quad if \\ \quad \min_{j,f,g}(l_{i,p,f} + d^{j,g}_{i,p,f}) < 2\min_{f'} l_{i,p,f'}; \\ 2\min_{f'} l_{i,p,f'}; \quad otherwise. \end{cases}$$

$$l'_i = \min_p l'_{i,p}$$

In the formula, $\min_{j,f,g}(l_{i,p,f} + d^{j,g}_{i,p,f})$ is the minimal co-run time of job i on processor p under a power cap with a co-runner that introduces the least interference. And $min_{f'}l_{i,p,f'}$ is the minimal standalone run time of job i, where frequency f' needs to ensure that the system does not exceed the power cap when job i is running standalone at processor p. The soundness of the formula comes directly from the Co-Run Theorem.

A side note: When computing the co-run lengths of two jobs, some care needs to be taken in treating the differences in their lengths. If $l_1 * d_1^2 < l_2 * d_2^1$, the co-run length of the first job is just $l_1 * d_1^2$, but the co-run length of the second job is not $l_2 * d_2^1$ because only part of it ($l_1 * d_1^2$ in length) actually co-runs with job 1. The remaining part is not subject to the interference from job 1. Therefore the co-run length of job 2 should be $l_1 * d_1^2 + l_2 - l_1 * d_1^2/d_2^1$.

V. Co-run Performance and Power Modeling

The previous section has assumed that co-run performance and power usage are known. This section describes how these info is predicted through some lightweight models and a *staged interpolation* method.

For a group of N programs with each processor having K frequency levels, exhaustive profiling would require at least $O(N^2 \times K^2)$ profiling co-runs. To avoid the time-consuming process, our solution uses controllable micro-kernels to first characterize the co-run performance space, through which, it predicts the co-run performance of two jobs by interpolations with their standalone performance that can be attained through some existing online sampling methods or cross-run predictive methods. Although micro-kernels have been used for characterizing performance on on CPU [33] before, the integrated heterogeneous processor features some special complexities, especially in identifying the crucial hardware resource that affects CPU-GPU co-run performance on such a system, and the interplay between

```
/* tid: thread id */
Kernel(in_data_1[], in_data_2[], out_data[],
        i_max, j_max){
   data_tmp = 0;
   for(i=0; i<i_max; i++){
      //Step 1: read data (memory related)
      data_1 = in_data_1[tid];
      data_2 = in_data_2[tid];

      //Step 2: do calculation (memory unrelated)
      for(j=0; j<j_max; j++){
         data_tmp += j;
         data_tmp = data_tmp % 10000;
      }
      //Step 3:  write back (memory related)
      data_out[tid] = data_1 + data_2*data_tmp;
   } //end of for loop
} //end of Kernel
```

Figure 4. Source code of micro-benchmark kernel.

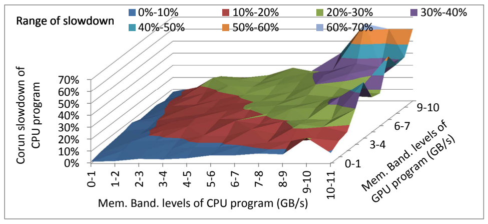

Figure 5. Spectrum of CPU program degradation due to memory contention.

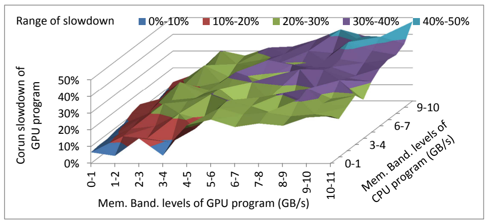

Figure 6. Spectrum of GPU program degradation due to memory contention.

performance and power cap. We are not aware of prior work on modeling CPU-GPU co-run performance and power usage on such settings.

In the rest of this section, We first describe the synthesized micro-benchmark, and its use in characterizing the co-run performance degradation space. We then describe how to use it with interpolation to predict co-run performance degradation and power consumption.

A. Devising the Micro-benchmark

To construct a co-run performance degradation space for estimate the performance degradations of co-running programs, two complexities have to be addressed. The first is to identify the critical hardware resource(s), the contention on which influences the co-run performance the most. The second is to devise a micro-benchmark, which can be easily controlled to generate different levels of pressure on that resource.

We focus on an architecture with the CPU and GPU sharing the last-level cache and the whole main memory system. Previous work on multicore CPU [35] have reported that the main memory access contention, rather than LLC contention, is the dominant reason of the co-run slowdown in the real system. We have observed similar phenomena on the heterogeneous integrated systems (both Intel and AMD). So, in the modeling, we primarily consider the impact of memory access contention. Section VI will show that it is sufficient to help yield appropriate schedules.

We devise a micro-benchmark as a software stressor that apply controllable pressure to the memory system. It is written in OpenCL such that it can run on both CPU and GPU. The source code of kernel is shown in Figure 4. The kernel contains three arrays, in_data_1, in_data_2, out_data. These arrays are large enough so that no one single array can stay in LLC during execution. In step 1, each thread the kernel launches reads data from in_data_1 and in_data_2. Then, step 2 controls the amount of arithmetic calculations, which does not access memory because all operands are hosted in registers. In step 3, the kernel writes back data to out_data. By setting the array sizes and the number of loop iterations in step 2, the kernel generates various degrees of memory demands.

B. Characterizing Co-run Degradation Space

To collect profiled data to construct the co-run performance degradation space, we run the micro-benchmark alone on the CPU and GPU, respectively, with 11 parameter settings to evenly cover main memory bandwidth throughput levels from 0 GB/s to 11 GB/s. We then co-run two micro-benchmarks with each pair of the throughput settings and collect the performance degradation data for both.

Figure 5 shows the co-run performance degradation for the micro-benchmark runs on the CPU with a co-running micro-benchmark instance on the GPU. The two axes at the horizontal plane show the throughput settings of the micro-benchmarks running on the CPU and GPU, respectively. The vertical axis shows the performance degradation for the micro-benchmark running on the CPU. Figure 6 shows the graph of GPU degradations (the two axes on the horizontal plane switched positions from Figure 5).

The graphs show that higher-throughput micro-benchmark executions tend to suffer larger slowdowns and lead to more serious degradation for the co-runner. CPU and GPU show different co-run degradation patterns. GPU appears to suffer more from co-runs, with most degradations in the 20%-40% range, while the CPU suffers 20% or less degradations in around half of the evaluated cases. Interestingly, the CPU shows much more serious slowdown than the GPU when both co-runners have a high memory demand (over 8.5 GB/s). For instance, the largest degradation for the CPU is about 65%, but only 45% for the GPU.

C. Staged Interpolation in Co-Run Degradation Space

Using the co-run performance model for predicting co-run performance of real programs requires the memory bandwidth statistics of the standalone executions of the real programs. With that info, their co-run degradations can be then predicted through two-dimensional linear interpolations upon the performance space already characterized by the micro-benchmark.

There are already many solutions on efficiently attaining or estimating standalone performance or power consumption of a program through sampling [9], statistical method [20] or cross-run prediction [27]. In this paper, to assess the full capability of the proposed co-scheduling algorithm and co-run performance models, we use offline profiling to record the standalone performance and power usage at each frequency level for experimental purpose. In practical settings, those existing lightweight methods can be used to estimate those metrics on the fly with minimum overhead.

VI. Evaluation

This section evaluates the efficacy of the co-scheduling method.

Platform. Our work employs an Intel Ivy Bridge processor, i7 3520M, which has an integrated GPU HD Graphics 4000. The CPU and GPU share a 4MB last-level cache. The CPU frequency is adjustable and ranges from 1.2 GHz to 3.6 GHz, while the GPU frequency can be changed from 350 MHz to 1.25 GHz. The system runs Ubuntu 14.04LTS with Linux kernel version 4.2. We use Intel OpenCL driver (version 1.2.0.43) for the CPU and an open-source GPU driver, Beignet (version 0.9.1) [1], to support OpenCL programs on the GPU as Intel's driver for GPU does not work on Linux. All programs are compiled by g++ (version 4.8) with O3 option.

Benchmarks. We select eight OpenCL programs from the Rodinia benchmark suite [6] (streamcluster, cfd, dwt2d, hotspot, srad, lud, leukocyte, and heartwall), which is widely used in many existing works. We discard the other programs in Rodinia because those programs show unstable performance across runs due to the immature support of the third-party driver, or some OpenCL features (such as APIs) of those programs are not supported by Beignet. The benchmarks cover multiple domains, such as machine learning, fluid dynamics, and particle simulation. They cover both compute-intensive and memory-intensive workloads. We use large enough inputs so that all instances of the benchmarks in the experiments run for at least 20 seconds.

A. Points of Comparisons

Existing co-scheduling methods are either about homogeneous processors [10, 15, 22, 23] or about efficiently or fairly sharing discrete GPU across applications [7, 8, 18, 28, 30], rather than dealing with co-run interferences between integrated CPU and GPU that are subject to power caps. These methods are hence not applicable to our problem. For example, Tian and others employs A*-search for finding good schedules for co-running jobs on multicore [26]; using the method for our problem would not answer the questions on how to decide the frequencies each processor shall use when running a job to observe the power cap, how to handle the impact of the frequency control on the performance of each possible co-schedule, and how to decide which type of processor a job should be placed.

Since there is no comparable prior work, we design two alternative approaches to compare with our heuristic algorithm, and employ the lower-bounds computed by our technique to roughly estimate the gap from the optimal.

Random co-scheduling (Random): This approach randomly picks jobs to run on CPU or GPU. Whenever a processor becomes idle, it randomly picks a new job to occupy that processor, or it just leaves the idle processor idle as some jobs prefer to be executed alone.

System's default co-scheduling (Default): In this approach, we want to leverage the default Linux scheduler to schedule the jobs. However, we need to partition the jobs to determine which jobs should run on which processor. To do that, we use the offline-profiled results for standalone runs to determine each program's preference. We rank the programs by the ratio between the standalone CPU run time and GPU runtime at the highest frequency. We partition the ranked programs and run the first partition on the GPU and the rest on the CPU. We ensure that the partitioning minimizes the sum of execution times of the longer partition.

Note that the two co-scheduling approaches above only produce co-schedules without a way to control the power consumption. When one of these approaches is used and the total power consumption is above the power cap, there are two methods to adjust frequency of the CPU and GPU described below.

GPU-biased: This approach tries to leverage the GPU's high-throughput. When the aggregate power consumption exceeds the power cap, it always lowers the CPU's frequency until the CPU reaches the lowest frequency. GPU's frequency is lowered only when the power cap requirement is still not satisfied. When there is energy room to raise frequencies, it always raises the GPU's frequency if it's not the highest yet.

CPU-biased: This approach is the opposite of the GPU-biased approach. It prefers to first lower the GPU's frequency and raise the CPU's frequency.

We evaluate the following two co-scheduling approaches proposed in this paper.

HCS: This approach follows the heuristic co-scheduling algorithm without post refinement.

HCS+: This approach is HCS plus the post refinement described in Section IV-A.

B. Performance and Power Model Accuracy

To evaluate the co-run performance model, we co-run every pair of the eight programs, and randomly select one to run on the GPU and the other on the CPU. There are 64 pairs in total (including two instances of the same program). For each pair, we test two frequency settings without power cap. In the first setting, we set both CPU and GPU at the

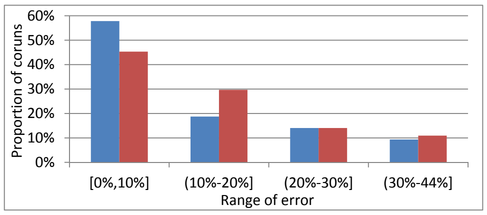

Figure 7. The error rate distribution of performance model. The x-axis represents the range of error rate, and y-axis represents the proportion of the 64 pairs that falls into the range of error rate.

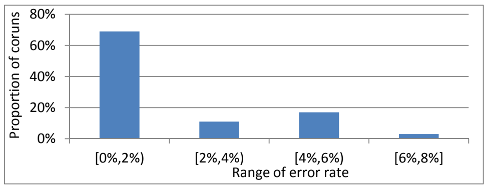

Figure 8. The error rate distribution of power model. The x-axis represents the range of error rate, and y-axis represents the proportion of the 64 pairs that falls into the range of error rate.

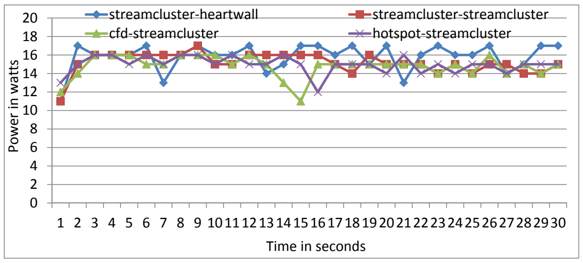

Figure 9. The power sample of four random co-runs. Co-run pair A-B means A on CPU and B on GPU.

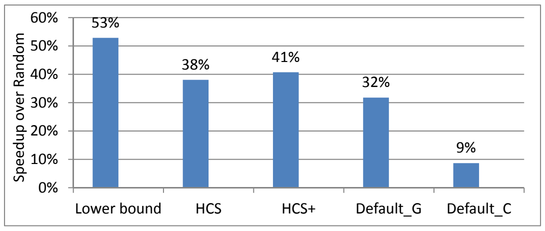

Figure 10. Speedup over Random (8 program instances; TDP=15 watts.)

highest frequency. In the second setting, we use the medium frequency (2.2GHz for CPU and 0.85GHz GPU).

Figure 7 shows the accuracy of the predicted slowdown for both cases. The accuracy is defined as the relative error of the predicted performance degradation compared to the real degradation. The x-axis shows multiple error ranges, and the y-axis represents the fraction of the 64 pairs in a range. For both frequency settings, around half of the co-runs have errors below 10% and more than 70% below 20%, which shows the reasonable accuracy of the co-run performance degradation model. On average, at the high frequency setting, the model shows 15% deviation from the ground truth, while the average error for the medium frequency setting is 11%. A plausible reason is that at high frequency, the programs have higher memory demands, leading to more serious and complex memory contention and hence complicating the prediction.

We use the same 64 pairs to evaluate the accuracy for power consumption prediction. For each pair, we select the frequencies for the CPU and GPU that meet the 16-watt power cap and yield the best performance. Figure 8 reports the histogram of the errors of using the power of standalone runs at the same frequency to predict the power usage of the co-runs. No error is larger than 8%. For 69% of the co-run pairs, the error for the power prediction is less than 2%. Only 3% of the co-runs demonstrate the worst accuracy (6–8%). The average error is 1.92%, demonstrating the effectiveness of the simple prediction approach.

Figure 9 shows the aggregate power consumption of the CPU and GPU for four randomly selected co-run pairs; one sample per second. It shows that the power consumption is lower than the power cap in most time, confirming the effectiveness of the power prediction and control. When the power consumption exceeds the power cap, the exceeded amount of power is typically less than 2W.

C. Performance Comparisons

Table I shows offline profiling results. Six out of the eight programs prefer running on the GPU. The program dwt2d runs much faster on the CPU and is the only program in the CPU-preferred set. The program lud has similar performance on the two processors, and hence belongs to the non-preferred set. All other programs are in the GPU-preferred set. The min. co-run time (CPU) and min. co-run time (GPU) rows show the execution time when the program runs with a co-runner that introduces the least contention.

The comparison uses Random as the baseline. We use Random to schedule the eight program 20 times with different seeds and take the average of the makespans. GPU-biased policy is used to adjust frequency to meet the 16 watts power constraint.

Figure 10 shows the performance improvement from the scheduling approaches, Default (Default_G and Default_C for the GPU-biased and CPU-biased policies respectively),

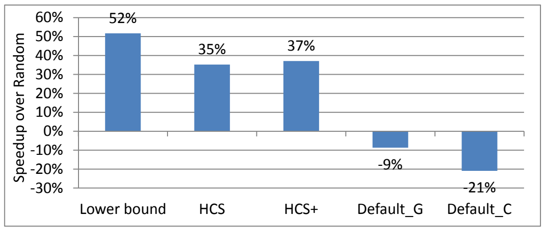

Figure 11. Speedup over Random (16 program instances; TDP=15watts.)

Table I
STANDALONE EXECUTION TIME PROFILED OFFLINE AND CO-RUN EXECUTION TIME WITH THE CO-RUNNER THAT INTRODUCES THE SMALLEST PERFORMANCE DEGRADATION PREDICTED BY THE PERFORMANCE MODEL.

Job Name	streamcluster	cfd	dwt2d	hotspot	srad	lud	leukocyte	heartwall
Min. co-run time (CPU)	62.70	57.15	29.97	84.99	61.66	32.48	61.06	66.71
Min. co-run time (GPU)	27.28	34.22	62.28	38.21	31.77	36.49	26.77	30.35
Standalone time (CPU)	59.71	49.69	24.37	70.24	51.39	27.76	50.88	54.68
Standalone time (GPU)	23.72	26.32	61.66	28.52	23.71	24.83	23.08	22.99
Preferred	GPU	GPU	CPU	GPU	GPU	Non	GPU	GPU

HCS, HCS+, and the best possible speedup from the lower bound makespan calculation. Recall that Default leverages HCS's models to partition the programs, which takes into account the program's processor preference. The preference awareness helps Default_G and Default_C improve performance by 32% and 9%, respectively. Default_G provides better performance, because it prefers high frequency on the GPU, which produces higher throughput for the programs. Compared to Default, HCS considers co-run contention and adjusts both CPU and GPU's frequencies for optimized performance. It outperforms Default_G by 6%. The local refinements help HCS+ get 3% extra performance gains.

D. Scalability Analysis

We also conduct a 16-program instance study to show the scalability of HCS. In this more complicated case, the benefits of HCS+ over the default schedules are much more significant. We launch two instances for each of the eight programs with different inputs. As shown in Figure 11, in this case, HCS produces 35% performance improvement over Random, and HCS+ delivers 37% improvement, 15% away from the lower bound. Default_G still outperforms Default_C, but both perform even worse than Random, showing 9% and 21% degradation, respectively. HCS+ gives over 46% speedups over the default schedules. Note that Random determines a fixed order for the co-runs, and at any time only launches up to one program to the CPU. Default, however, launches multiple programs to the CPU at the very beginning, which share the CPU through context switching. The context switching introduces overhead and worsens locality at the cache level and the main memory level (i.e., more page faults).

Scheduling Overhead. The scheduling algorithm takes almost no time to run (less than 0.1% of the makespan) for its linear computational complexity.

VII. RELATED WORK

Job scheduling: Co-scheduling a batch of jobs to a system for the best throughput is not a new problem. Researchers designed a number of approaches to map jobs to homogeneous [10, 15, 16, 22, 23] or heterogeneous machines [7, 8, 18, 28, 30]. The major concerns include locality [14–16, 23], job length [26], program characteristics [7, 10, 18, 28, 30], energy efficiency [18, 22], or the effective usage of the GPU that is shared by multiple applications [8].

As mentioned in the introduction and evaluation sections, none of those co-scheduling solutions is applicable to our problem as they do not consider the special complexities in power cap, placement of jobs, or their subtle interplay with the CPU-GPU interferences on the integrated environment.

The research on scheduling for integrated architectures mainly focuses on mapping workloads of the same application to the CPU and GPU. Kaleem and others [17] propose adaptive online profiling to partition the workloads for load balance, which is not feasible for solving our problem as it requires the considerations of the contention between all possible pairs of the job set at various frequency levels. The overhead is not affordable for runtime co-scheduling.

Maximizing performance under a power cap: The power cap constrain is becoming prominent, which has been studied by both hardware and software community. Venders even introduced special hardware approaches to enforce power cap, such as RAPL (Running Average Power Limit) [24] and ACPI (Advanced Configuration and Power Interface) [12]. The purpose of these techniques are to establish industry-standard interfaces for enabling OS-directed configuration for power management.

Zhang and Hoffmann [32] evaluated a set of software, hardware, and hybrid approaches to maximize performance as well as satisfying a power cap requirement. They focused on a single parallel application on the CPU and used control theory to dynamically choose optimal parameters for maximized performance. Our platform is different from theirs and the input to our problem is a set of applications. Komoda and others [18] considered job scheduling on heterogeneous systems under a power cap. They developed empirical models to guide the DVFS setting and job mapping. Although the CPU and GPU share the same power budget, they do not share the same memory system and hence do not interfere with each other in terms of performance.

Contention mitigation for integrated architectures: Mekkat et. al [21] demonstrated the different access behaviors of the CPU and GPU on the shared last-level cache. They proposed novel management policies to better coordinate the accesses for performance improvement. Lee and Kim [19] proposed a core-sampling technique to predict the integrated GPU's performance and leveraged cache partitioning to improve performance for heterogeneous workloads. With those approaches supported by commodity integrated processors, the co-running applications would have better and more predictable performance. Zhu et. al [34] provided a preliminary exploration on understanding the co-run performance of CPU and GPU programs on integrated GPUs, with no co-scheduling solutions proposed.

VIII. Conclusion

In this paper, we studied the problem of job co-scheduling on an integrated system with a power cap. We revealed the important factors, including memory contention, power contention and job lengths, that affect performance. We proposed heuristic algorithms to efficiently find good co-schedules, which produce throughput improvement by as much as 46% over the deafult schedules.

References

[1] Beignet. https://01.org/zh/beignet/.

[2] Opencl specification 2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[3] A. M. Aji, A. J. Pena, P. Balaji, and Wu Chun Feng. Automatic command queue scheduling for task-parallel workloads in opencl. In *IEEE International Conference on CLUSTER Computing*, pages 42–51, 2015.

[4] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's first 64-bit arm processor. *IEEE Micro*, 35(2):46–55, 2015.

[5] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[7] Linchuan Chen, Xin Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. pages 1–11, 2012.

[8] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *ASPLOS*, 2016.

[9] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *PACT*, 2008.

[10] Ali El-Moursy, Rajeev Garg, David H Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *IPDPS*, 2006.

[11] Chris Gregg, Michael Boyer, Kim Hazelwood, and Kevin Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Proceedings of the 2nd Workshop on Applications for Multi- and Many-Core Processors. San Jose, CA*, 2011.

[12] H. P. Intel and Phoenix Microsoft. Advanced configuration & power interface (acpi) specification. 2006.

[13] D. James. Intel ivy bridge unveiled the first commercial tri-gate, high-k, metal-gate cpu. *Proceedings of the Custom Integrated Circuits Conference*, pages 1–4, 2012.

[14] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, pages 220–229, October 2008.

[15] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. The complexity and approximation of optimal job co-scheduling on chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems.*, 22(7), 2011. (DOI: 10.1109/TPDS.2010.193).

[16] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of The International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, pages 201–215, 2010.

[17] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *PACT*, pages 151–162, 2014.

[18] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, and Shinobu Miwa. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *IEEE International Conference on Computer Design*, pages 349 – 356, 2013.

[19] Jaekyu Lee and Hyesoon Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[20] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing, 2009.

[21] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *PACT*, 2013.

[22] David K. Newsom, Olivier Serres, Sardar F. Azari, and Abdel Hameed A. Badawy. Energy efficient job co-scheduling for high-performance parallel computing clusters. In *IEEE International Conference on Smart City/socialcom/sustaincom*, pages 550–556, 2015.

[23] Thomas Phan, Kavitha Ranganathan, and Radu Sion. Evolving toward the perfect schedule: Co-scheduling job assignments and data replication in wide-area systems using a genetic algorithm. In *Job Scheduling Strategies for Parallel Processing, International Workshop, Jsspp 2005, Cambridge, Ma, Usa, June 19, 2005, Revised Selected Papers*, pages 173–193, 2005.

[24] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.

[25] Wei Tang, Narayan Desai, Venkatram Vishwanath, Daniel Buettner, and Zhiling Lan. Job coscheduling on coupled high-end computing systems. pages 317–326, 2011.

[26] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of ACM Computing Frontiers*, pages 41–50, 2009.

[27] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[28] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for opencl programs on CPU/GPU heterogeneous platforms. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10, 2014.

[29] Kenneth Yoshimoto, Patricia Kovatch, and Phil Andrews. Co-scheduling with user-settable reservations. In *International Conference on Job Scheduling Strategies for Parallel Processing*, pages 146–156, 2005.

[30] Feng Zhang, Jidong Zhai, Wenguang Chen, and Bingsheng He. To co-run, or not to co-run: A performance study on integrated architectures. In *IEEE International Symposium on Modeling*, pages 89–92, 2015.

[31] Feng Zhang, Jidong Zhai, Wenguang Chen, Bingsheng He, and Shuhao Zhang. To co-run, or not to co-run: A performance study on integrated architectures. In *23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2015, Atlanta, GA, USA, October 5-7, 2015*, pages 89–92, 2015.

[32] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

[33] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Microarchitecture (MICRO), Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[34] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang. Understanding co-run degradations on integrated heterogeneous processors. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2014.

[35] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *Acm Sigplan Notices*, 45(3):129–142, 2010.