Orchestrated Co-scheduling, Resource Partitioning, and Power Capping on CPU-GPU Heterogeneous Systems via Machine Learning

Issa Saba, Eishi Arima(✉), Dai Liu, and Martin Schulz

Technical University of Munich, Garching, Germany
{issa.saba,eishi.arima,dai.liu,martin.w.j.schulz}@tum.de

Abstract. CPU-GPU heterogeneous architectures are now commonly used in a wide variety of computing systems from mobile devices to supercomputers. Maximizing the throughput for multi-programmed workloads on such systems is indispensable as one single program typically cannot fully exploit all avaiable resources. At the same time, power consumption is a key issue and often requires optimizing power allocations to the CPU and GPU while enforcing a total power constraint, in particular when the power/thermal requirements are strict. The result is a system-wide optimization problem with several knobs. In particular we focus on (1) co-scheduling decisions, i.e., selecting programs to co-locate in a space sharing manner; (2) resource partitioning on both CPUs and GPUs; and (3) power capping on both CPUs and GPUs. We solve this problem using predictive performance modeling using machine learning in order to coordinately optimize the above knob setups. Our experiential results using a real system show that our approach achieves up to 67% of speedup compared to a time-sharing-based scheduling with a naive power capping that evenly distributes power budgets across components.

Keywords: Co-scheduling · Resource partitioning · Power capping · CPU-GPU heterogeneous systems · Machine learning

1 Introduction

Heterogeneous CPU-GPU architecture are now broadly used in a wide variety of computing systems, including mobile devices, PCs, datacenter servers, and HPC systems. For instance, over 160 out of the 500 top-class supercomputers are now GPU-accelerated systems (as of Jun 2022) [1]. This trend is driven by the end of Dennard scaling [14], i.e., the exponential growth of single-thread performance in microprocessors had ceased, and the industry rather shifted toward thread-level parallelism and heterogeneous computing using domain specific accelerators [15]. GPUs are one of the most commonly used accelerators due to their wide range of application areas, including image processing, scientific computing, artificial intelligence and data mining.

As computing systems are becoming more powerful and more heterogeneous using a wide variety of resources, it also becoming more difficult to fully utilize the entirety of compute resources by one single application. One reason behind the trend is that it is not always easy to identify a large enough fraction of a code that can be ported to GPUs (or any other accelerator) while balancing loads across all the processing units (CPU, GPU, or any accelerators). Further, the scalability of applications inside of a chip can be limited by various factors such as intensive memory accesses and shared resource contentions, which can induce a significant waste of compute resources.

Therefore, **co-scheduling**, i.e., co-locating multiple processes in a space sharing manner, is a key feature to mitigate resource wastes and to maximize throughput on such systems, if the processes are complimentary in their resource usage. To achieve the latter, a sophisticated mechanism to **partition resources** at each component and allocate them accordingly to co-scheduled processes is indispensable. Recent commercial CPUs and GPUs support such resource partitioning features: (1) one can designate physical core allocations to co-scheduled processes on CPUs; and (2) GPUs have begun to support hardware-level resource partitioning features for co-locating multiple processes—one example is NVIDIA's Multi-Instance GPU (or MIG) feature that is supported in the *most recent* high-end GPUs to enable co-locating multiple programs at the same chip while partitioning it at the granularity of GPC [21].

Meanwhile, as power (or energy) consumption is now a first-order concern in almost all the computing systems from embedded devices to HPC systems [10, 22, 23], performance optimizations for modern computing systems, including co-scheduling, must consider power optimization and in most cases also hard power limits or constraints. Once a power constraint is set on a system, the power budgets must be distributed to components accordingly so as to maximize the performance while keeping the constraint. To realize such a mechanism, modern CPUs and GPUs now support **power capping** features that set a power limit at the granularity of chip (or even at a finer granularity for some hardware).

Driven by the above trends, this paper explicitly targets the combination of co-scheduling, resource partitioning, and power capping on CPU-GPU heterogeneous systems, and provides a systematic solution to co-optimize them using a machine-learning-based performance model as well as a graph-based scheduling algorithm. Our model takes both application profiles and hardware knob states into account as its inputs and returns the estimated performance of the co-located applications as the output. More specifically, the profiles are based on hardware performance counters, and the hardware knob states include resource partitioning and power capping on both the CPU and GPU. We use this performance model to estimate the best performance of different hardware setups for a given application pair, which is used to determine the best co-scheduling pairs in a graph-based algorithm, i.e., Edmonds' algorithm [13].

The followings are the major contributions of this paper:

1. We comprehensively and systematically optimize (1) co-scheduling pair selections, (2) resource partitioning at both CPU and GPU, and (3) power budget-

ing on both CPU and GPU, using a real CPU-GPU heterogeneous hardware platform.
2. We define an optimization problem and provide a systematic solution to select the best job pair and the best hardware setups including resource partitioning and power capping on CPU/GPU.
3. We develop a machine-learning-based performance model that takes both the characteristics of the co-located applications and the hardware states (including partitioning and power capping on CPU/GPU) into account.
4. We solve the optimization problem by using the above performance model building on the graph-based Edmonds' algorithm.
5. We quantify the benefits of our approach by using a real hardware, and show that we improve the system throughput by 67% compared to a time-sharing-based scheduling with a naive power capping that evenly distributes the power budgets across the CPU and GPU.

2 Related Work

Ever since multi-core processors appeared on the market, co-scheduling, resource partitioning, and power capping have been studied. However, ours is the first work that covers all of the following aspects simultaneously: (1) targeting CPU-GPU heterogeneous systems; (2) comprehensively co-optimizing co-scheduling pair selections, resource partitioning, and power capping, using machine-learning-based performance modeling and a graph-based algorithm; and (3) quantifying the benefits using a real hardware that is capable of both resource partitioning and power capping at both the CPU and the discrete GPU.

M. Bhadauria et al. explored co-scheduling multi-threaded programs in a space sharing manner using a multi-core processor [9]. S. Zhuravlev et al. focused on the shared resource contention across co-located programs on multi-core processors and proposed a scheduler-based solution to mitigate the interference [26]. R. Cochran et al. proposed Pack & Cap that optimizes the scale of multi-threaded applications via the thread packing technique while applying power capping [12]. Then, H. Sasaki et al. provided a sophisticated power-performance optimization method that coordinates the thread packing technique and DVFS for multi-programmed and multi-threaded environments [24]. These seminal studies provided insightful ideas, *however they did not target CPU-GPU heterogeneous systems.*

Few studies looked at the combination of co-scheduling and power capping on *CPU-GPU heterogeneous systems*. Q. Zhu et al. worked on the combination of job scheduling and power capping for integrated CPU-GPU systems [25], but they did not cover the following aspects: resource partitioning inside of CPU/GPU; and co-scheduling multiple processes on the GPU in a space sharing manner. R. Azimi et al. developed a framework called PowerCoord that allocates power budgets to components on CPU-GPU heterogeneous systems for co-scheduled workloads [5], but their work did not target adjusting the resource partitions as well. *Recent hardware advances (e.g., NVIDIA MIG feature* [4,21]*) made it possible to apply both the process co-location and resource partitioning on both CPUs and GPUs, which opened up new optimization opportunities.*

There have been several studies that utilize machine learning (including linear regression) for performance/power modeling in the literature. B. Lee et al. utilized the linear regression to predict performance for CPUs [18]. E. İpek et al. conducted microarchitectural design space explorations using a neural network [17]. B. Barnes et al. proposed a statistical approach to predict performance of parallel applications [7]. H. Nagasaka et al. constructed a power consumption model for GPUs that is based on the linear regression and hardware performance counters [19]. Beyond these pioneering studies, machine-learning-based approaches have been utilized also for more complicated system design and optimization purposes such as: clock frequency setups at both CPU and GPU at the same time on a CPU-GPU integrated chip [6]; power capping setups on CPU, DRAM, and NVRAM [3]; coordination of thread/page mapping and prefetcher configurations [8]; and CPU-GPU heterogeneous chip designs in the industry [16]. We follow the literature and utilize a neural network that is tailored to solve our new problem.

3 Motivation, Problem, and Solution Overview

3.1 Motivation: Technology Trends

Setting a power cap on a processor is a crucial feature and is now supported on a variety of commercial CPUs and GPUs. One prominent use case for this feature is to protect a chip from overheating and, instead of having to be conservative, to adjust the needed settings to the machine environment such as the cooling facility. Another prominent use case is enabling a hierarchical and cooperative power budgeting across components or compute nodes while keeping a total power constraint, which has been widely studied from standalone computers to large-scale systems, including datacenters and supercomputers [10,22,23]. In our work, we target CPU-GPU heterogeneous computing systems (or nodes) and optimize the power cap setups on both CPU and GPU in order to maximize performance under a total node power constraint.

As compute nodes are becoming fatter and systems more heterogeneous, it is also becoming more difficult to fully utilize an entire node's resources by one single process. For instance, compute resources are typically under utilized for memory-bound applications, while memory bandwidth is often wasted for compute-bound applications. Further, some applications are suitable for running on GPUs, but others are not. To improve resource utilization, mixing different kinds of processes and co-scheduling (or co-locating) them on the node at the same time while setting resource allocations accordingly at both the CPU and GPU is a desired feature.

3.2 Problem Definition

Figure 1 illustrates the overall problem we target in this paper. Here, we assume that we have one single job (or process) queue on the system (Q). We convert

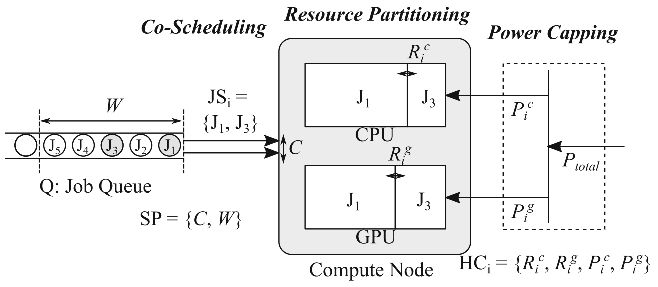

Fig. 1. Problem overview

the job queue into a list of job sets (or pairs) to co-schedule $(\mathrm{JS}_1, \mathrm{JS}_2, \cdots)$. Note these jobs are selected from inside of the window (W) on the queue. The concurrency, i.e., the maximum number of jobs launched at a time, is limited by a given parameter (C), and we particularly target $C = 2$ in this paper, meaning that no more than 2 CPU-GPU jobs will be co-scheduled at any given time. This value was chosen as for higher values no polynomial-time algorithms for job-set selection is known. We represent a set of these scheduling parameters as $\mathrm{SP} = \{C, W\}$. When launching/co-locating the ith job set $(\mathrm{JS_i})$, we optimize the hardware knob configurations $(\mathrm{HC_i})$, i.e., resource partitioning on CPU/GPU (R_i^c/R_i^g) as well as the power cap setups on CPU/GPU (P_i^c/P_i^g). Note, the sum of the power caps must be less than or equal than the given total power constraint P_{total}.

The following is the mathematical formulation as an optimization problem:

$$\begin{aligned}
&\textit{inputs} \ \ \mathrm{Q} = \{\mathrm{J_1}, \mathrm{J_2}, \cdots, \mathrm{J_W}\}, P_{total}, \mathrm{SP} \\
&\textit{outputs} \ \mathrm{L_{JS}} = \{\mathrm{JS_1}, \mathrm{JS_2}, \cdots\}, \mathrm{L_{HC}} = \{\mathrm{HC_1}, \mathrm{HC_2}, \cdots\} \\
&\min \quad \sum_{i=1}^{|\mathrm{L_{JS}}|} CoRunTime(\mathrm{JS_i}, \mathrm{HC_i}) \\
&s.t. \quad CoRunTime(\mathrm{JS_i}, \mathrm{HC_i}) \leq SoloRunTime(\mathrm{JS_i}, P_{total}) \\
&\qquad P_i^c + P_i^g \leq P_{total}, \ \ 1 \leq |\mathrm{JS_i}| \leq C \\
&\qquad (\forall i : 1 \leq i \leq |\mathrm{L_{JS}}| (= |\mathrm{L_{HC}}|)) \\
&\qquad \mathrm{JS_1} \cup \cdots \cup \mathrm{JS}_{|\mathrm{L_{JS}}|} = \mathrm{Q}, \ \ |\mathrm{JS_1}| + \cdots + |\mathrm{JS}_{|\mathrm{L_{JS}}|}| = W
\end{aligned}$$

The inputs are the job queue, the total power cap setup, and the set of the scheduling parameters. The outputs are the lists of job sets $(\mathrm{L_{JS}})$ and the associated hardware configurations $(\mathrm{L_{HC}})$. The objective is to minimize the sum of the co-run execution time (*CoRunTime*), each of which is a function of the co-located jobs as well as the hardware configurations.

We take several constraints for this optimization problem into count: the first is the requirement that the space-sharing co-run execution should take shorter time than the time-sharing execution with exclusive solo-runs under the power

Table 1. Definitions of parameters/functions

Parameter or Function	Remarks
Q	A list or queue of jobs: $Q = \{J_1, J_2, \cdots, J_W\}$
J_i	ith job in the job list (or queue)
P_{total}	The total power cap for the target computing node
SP	A set of scheduling parameters: $SP = \{C, W\}$
C	The maximum number of concurrently executed jobs
W	The number of scheduling targets on the job queue
L_{JS}	A list of job sets to be co-scheduled: $L_{JS} = \{JS_1, JS_2, \cdots\}$
JS_i	ith set of jobs in L_{JS} to be co-scheduled
L_{HC}	A list of hardware configurations associated with the job sets: $L_{HC} = \{HC_1, HC_2, \cdots\}$
HC_i	The hardware configurations for ith job set: $HC_i = \{R_i^c, R_i^g, P_i^c, P_i^g\}$
$R_i^*(* = c/g)$	The resource partitioning setup on CPU/GPU for ith job set
$P_i^*(* = c/g)$	The power cap set up on CPU/GPU for ith job set
$CoRunTime(JS_i, HC_i)$	The total execution time when co-locating JS_i with HC_i
$SoloRunTime(JS_i, P_{total})$	The total time when executing JS_i in a time-sharing manner under the total power cap (P_{total}); The power caps to CPU/GPU are optimized for each job execution

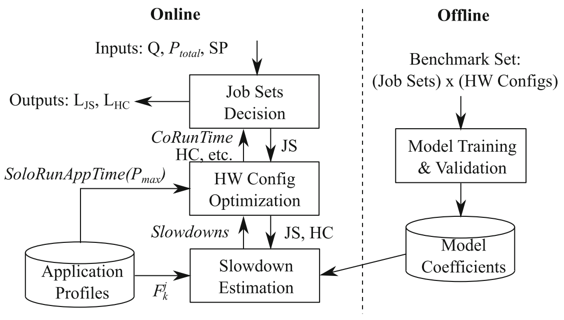

Fig. 2. Workflow of our solution

cap ($SoloRunTime$)—otherwise we should not co-schedule them. The second one is the power constraint, i.e., the sum of the CPU/GPU power caps must be less than or equal to the total node power cap. The next constraint regulates the concurrency on the system, i.e., the number of jobs in a set to be co-scheduled (JS_i) must be less than or equal to C. The last two constraints specify that the list of job sets (L_{JS}) must be created from the job queue (Q) in a mutually exclusive and collectively exhaustive manner. Note Table 1 summarizes the parameters/functions used.

3.3 Solution Overview

Figure 2 depicts the overall workflow of our approach. As shown in the figure, it consists of an offline (right) and an online part (left).

During the offline part, we train the coefficients of our performance model, which we describe later in the paper, by using a predetermined benchmark set. More specifically, by executing various job sets while changing the hardware configurations, we generate a large enough number of data sets, which are used as inputs for the model training.

During the online part, we solve the optimization problem described in Sect. 3.2. This solution process consists of three parts (from top to bottom), and they work in a cooperative manner. We first determine the list of co-scheduling job sets ($\mathrm{L_{JS}}$) and return it with the associated list of optimal hardware configurations ($\mathrm{L_{HC}}$) (top part in the left figure). This component then communicates with the next stage (middle part in the left figure), i.e., continuously sends a temporal job set (JS) and receives the estimated co-run execution time (*CoRunTime*) along with the optimal hardware configurations (HC) and the solo-run time (*SoloRunTime*), which are used for the scheduling decisions. The component in the middle optimizes the hardware configurations (HC) for the job set (JS) given by the previous component. More specifically, it continuously sends the job set (JS) and a temporal hardware configuration (HC) to the third part (bottom part in teh left figure) and receives the estimated slowdowns until finding the optimal hardware configuration. The latter component estimates the slowdowns for the given jobs (JS) with the given hardware configuration (HC) by using the associated job profiles as well as the model coefficients obtained in the offline model training. Here, we assume that the profile of a job is collected beforehand during its first run without co-scheduling nor power capping[1]. The details are described in the next section that provides also the definitions of $SoloRunAppTime(P_{max})$, F_k^j, and $Slowdown$ shown in the figure.

4 Modeling and Optimization

4.1 Slowdown Estimation for a Given Job Set and Hardware Setup

Metric Formulations: We first provide simple formulations for the metrics appeared in Sect. 3.2 as follows:

$$\begin{aligned}
CoRunTime(\mathrm{JS},\mathrm{HC}) &= \max_{\mathrm{J_j}\in\mathrm{JS}} CoRunAppTime_j(\mathrm{JS},\mathrm{HC})\\
SoloRunTime(\mathrm{JS},P_{total}) &= \sum_{\mathrm{J_j}\in\mathrm{JS}} SoloRunAppTime_j(P_{total})\\
CoRunAppTime_j(\mathrm{JS},\mathrm{HC}) &= Slowdown_j(\mathrm{JS},\mathrm{HC})\cdot SoloRunAppTime_j(P_{max})\\
SoloRunAppTime_j(P_{total}) &= Slowdown_j(\{\mathrm{J_j}\},\{R^c_{max},R^g_{max},OptP^c_j,OptP^g_j\})\\
&\quad\cdot SoloRunAppTime_j(P_{max})
\end{aligned}$$

The parameters and functions used to formulate them are summarized in Table 2. The first equation denotes that the total execution time when co-scheduling a job set (JS) is determined by the longest execution time in the set. The second

[1] In case no profile is available for a job, which we do not cover in the paper, we can exclude it from the co-scheduling candidates at the first stage in the diagram and execute it exclusively without power capping while obtaining the profile for the future references.

Table 2. Definitions of parameters or functions to formulate $CoRunTime()$/ $SoloRunTime()$

Parameter or Function	Remarks
$CoRunAppTime_j(\text{JS, HC})$	The execution time of jth job in a given job set (JS) when co-scheduling JS under a given hardware setup (HC)
$SoloRunAppTime_j(P_{total})$	The execution time of jth job in a given job set (JS) when its exclusive solo run under a power cap (P_{total})
$Slowdown_j(\text{JS}, \text{HC})$	The slowdown ratio of jth job in a given job set (JS) caused by co-scheduling JS under a given hardware setup (HC)
J_j	jth job in a given job set (JS)—JS = $\{\text{J}_1, \text{J}_2, \cdots\}$
$R^*_{max}(* = c/g)$	The maximum resource allocation on CPU/GPU to a given job
$OptP^*_j(* = c/g)$	The optimal power cap set up on CPU/GPU for jth job in a set when exclusive solo run ($OptP^c_j + OptP^g_j = P_{total}$)
P_{max}	The maximum total power cap or TDP ($P_{total} \leq P_{max}$)
F^j_k	kth parameter to characterize the features of J_j, given by hardware performance counters on both CPU and GPU

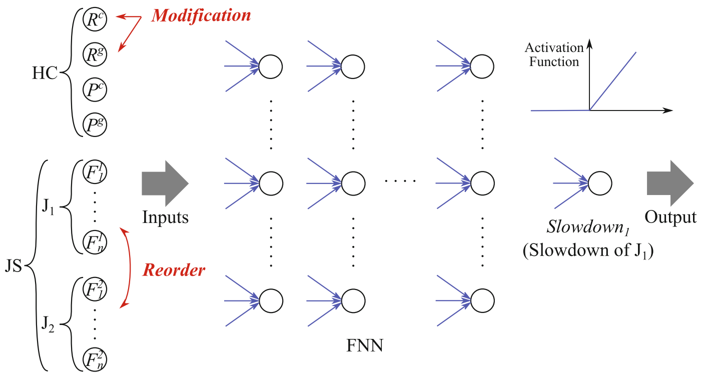

Fig. 3. General structure of our performance modeling ($C = 2$)

equation represents that the total execution time of time-shared scheduling is simply the sum of the solo-run execution time of the jobs in the set. The third equation shows that the execution time of a co-scheduled job ($CoRunAppTime_j$) is equal to the slowdown ($Slowdown_j$) multiplied by the solo-run execution time without power capping ($SoloRunAppTime_j(P_{max})$). In the fourth one, the performance degradation caused by power capping for a solo run can be described by using the same slowdown function used in the third equation. In this paper, the solo-run execution time without power capping is given by the associated profile, and we predict the slowdown part in those last two equations.

Performance Modeling: Figure 3 illustrates the general structure to model the slowdown function provided above. Here, we utilize a simple feedforward neural network (FNN) to estimate the slowdown for the first job (J_1) in the job set (JS) when co-scheduling. We regard the slowdown as a function of the job features (F_k^j) of all the co-located jobs as well as the hardware configuration (HC) to assess various factors such as scalability, interference, and resource allocations. The job features here are the hardware performance counters collected from both the CPU and GPU during the profile run described in Sect. 3.3. The exact definitions for the job features used in our evaluation are listed in Sect. 5.1. As for the slowdowns of the other co-located job(s), we simply reorder or replace the input locations (i.e., exchange the location between J_1 and J_j) and modify the resource allocation parameters (R_*) accordingly so that the allocations are associated with the new job order. Further, we also utilize the model to estimate the impact of power capping on solo-run performance. To do so, we simply designate HC = $\{R^c_{max}, R^g_{max}, OptP^c_j, OptP^g_j\}$ as previously mentioned and set all the job features other than the first job to zero in the model inputs. The detailed network configurations such as the exact inputs, the layer setups, the activation function, or the loss function are described in Sect. 5.1.

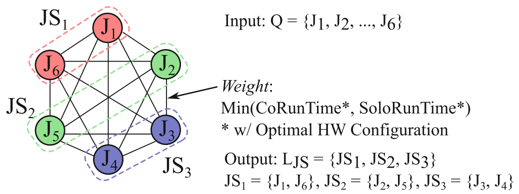
JS1
JS2
JS3
J1
J2
J3
J4
J5
J6
Input: Q = {J1, J2, ..., J6}
Weight:
Min(CoRunTime*, SoloRunTime*)
* w/ Optimal HW Configuration
Output: LJS = {JS1, JS2, JS3}
JS1 = {J1, J6}, JS2 = {J2, J5}, JS3 = {J3, J4}

Fig. 4. Overview of graph-based job sets creation ($W = 6$, $C = 2$)

4.2 Hardware Setup Optimization for a Given Job Set

By using the performance model provided above, we optimize the hardware configuration parameters (HC) for a given job set (JS) when co-scheduling. Here, we attempt to pick up the best hardware configuration (HC) from all the possible configurations so as to minimize $CoRunTime(\text{JS}, \text{HC})$. In this study, we simply utilize the exhaustive search, i.e., testing all the possible HC for the model inputs and choosing one that minimizes $CoRunTime$ for the given job set (JS). This is because the number of all the possible setups for HC on our target platform (or other systems available today) is limited as described later in Sect. 5.1. If the configuration space would explode in future systems, applying heuristic algorithms (e.g., hill climbing) would be a promising option. In addition, we select the pair of ($OptP^c_j$, $OptP^g_j$) for each job (J_j) in a given job set so as to obtain $SoloRunTime(\text{JS}, P_{total})$, for which we also explore in an exhaustive manner under the constraint of $OptP^c_j + OptP^g_j = P_{total}$.

4.3 Job Sets Selection

We then make scheduling decisions using the above hardware setup optimization functionality based on the results of our performance model. We regard the job

Algorithm 1: Job Scheduling Procedure ($C = 2$)

```
Inputs: Q = $\{J_1, \cdots, J_W\}$, $P_{total}$, SP = $\{C = 2, W\}$
Outputs: $L_{JS} = \{JS_1, JS_2, \cdots\}$, $L_{HC} = \{HC_1, HC_2, \cdots\}$

/* Initialization                                                          */
1  $L_{JS} \leftarrow \varnothing$; $L_{HC} \leftarrow \varnothing$;
2  Vortexes← Q; Edges← $\varnothing$; Weights← $\varnothing$; HWConfigs← $\varnothing$; CoRunFlags← $\varnothing$;
/* Graph creation                                                          */
3  for $i = 1 \rightarrow W$ do
4      for $j = i + 1 \rightarrow W$ do
5          Edges.push_back($\{J_i,J_j\}$); // Append this job set
6          (HCco, $CoRunTime$) ← GetOptimalCoRunHWConfig($J_i$,$J_j$);
7          (HCsolo1, HCsolo2, $SoloRunTime$) ← GetOptimalSoloRunHWConfig($J_i$,$J_j$);
           /* Append the weight and the HW config (incl. co-run or solo-runs) that
              minimizes time for this job set                                  */
8          if $CoRunTime \leq SoloRunTime$ then
9              Weights.push_back($CoRunTime$); CoRunFlags.push_back(1);
               HWConfigs.push_back({HCco});
10         else
11             Weights.push_back($SoloRunTime$); CoRunFlags.push_back(0);
               HWConfigs.push_back({HCsolo1, HCsolo2});
12         end
13     end
14 end
/* Job sets decision w/ Edmonds' Algorithm                                 */
15 $L'_{JS}$ ← EdmondsAlgorithm(Vortexes, Edges, Weights);
16 $L'_{HC}$ ← PickupSets(HWConfigs, $L'_{JS}$); // Pick the associated HW setups /w $L'_{JS}$
17 $L_{Flag}$ ← PickupSets(CoRunFlags, $L'_{JS}$); // Create a co-/solo-run flag list
/* Divide sets in $L'_{JS}$/$L'_{HC}$ if solo-run execution is better than co-scheduling   */
18 while $L_{Flag} \neq \varnothing$ do
19     $Flag$ ← $L_{Flag}$.pop_front(); JS ← $L'_{JS}$.pop_front(); HC ← $L'_{HC}$.pop_front();
20     if $Flag = 1$ then
21         $L_{JS}$.push_back(JS); $L_{HC}$.push_back(HC);
22     else
23         while JS $\neq \varnothing$ do
24             $J$ ← JS.pop_front(); $HCsolo$ ← HC.pop_front();
25             $L_{JS}$.push_back({J}); $L_{HC}$.push_back({HCsolo});
26         end
27     end
28 end
29 return ($L_{JS}$, $L_{HC}$);
```

co-scheduling problem as a minimum weight perfect matching problem and solve it using Edmonds' algorithm [13]. Figure 4 depicts the overview of the solution. In the figure, the vertices represent the jobs in the queue ($\mathrm{Q} = \{\mathrm{J}_1, \cdots, \mathrm{J}_\mathrm{W}\}$), and the weights represent the minimum execution time for the associated job sets. To obtain each weight, we estimate both of the best $CoRunTime$ and $SoloRunTime$ for each edge (or job pair) by using the model-based hardware configuration optimization described above, and choose one from them so that the execution time is minimized. Then, by using the graph, we create the list of job sets ($\mathrm{L}_\mathrm{JS} = \{\mathrm{JS}_1, \mathrm{JS}_2, \cdots\}$) that includes all jobs in the queue in a mutually exclusive and collectively exhaustive manner, while minimizing the sum of the weights of L_JS. This is a well-known minimum weight perfect matching problem and is identical to the optimization problem defined in Sect. 3.2 except that a job set can be executed in the time-sharing manner, which we can easily convert to meet the problem definition in Sect. 3.2 by simply dividing such a job set into multiple job sets, all of which include only one job. The Edmonds' algorithm provides the optimal solution with polynomial time complexity, particularly when the scheduling parameter set (SP) meets both of the following conditions: (1) W is an even number; and (2) C is equal to 2 [13]. For the former, we simply set the window size to an even number, and as for the latter, we focus on $C = 2$ to limit the complexity as described before. Note that a more precise version of the solution is described in Algorithm 1.

5 Evaluation

5.1 Evaluation Setup

Environment. For our evaluation, we use the platform summarized in Table 3. Our approach is applicable when both the CPU and GPU are capable of both resource partitioning and power capping. This is usually the case for most of the commercial CPUs today, and we utilize an NVIDIA A100 GPU card that supports the MIG feature and power capping [21].

Table 4 summarizes the resource partitioning and power capping settings we explore in this evaluation. We allocate CPU cores in a compact fashion, i.e., physically adjacent cores are assigned to the same program. We partition the GPU into 3GPCs/4GPCs or 4GPCs/3GPCs, on which low level memory hierarchies including L2 caches and memory modules are shared across all the GPCs[2]. To collect performance counter values when profiling, we utilize Linux perf [2] command for the CPU and NSight Compute [20] for the GPU. By using these profiling frameworks, we collect the performance counter values listed in Table 5. The definitions of these performance counters are the same as those shown in the tools (Table 5).

Benchmarks and Dataset. We use the Rodinia benchmarks [11], which is a well-known benchmark suite widely-used for various heterogeneous computing

Table 3. Evaluation system specifications

Name	Remarks
CPU	AMD Ryzen Threadripper 2990 WX, 32 cores
Main Memory	DDR4 2933 MT/s x4ch, 64 GB (Total)
GPU	NVIDIA A100 40 GB PCIe, 8GPCs
Operating system	Ubuntu 20.04.4 LTS, Kernel Version 5.4.0-120-generic
Compiler and drivers	GCC/G++ Version: 9.4.0, CUDA Version: 11.6, Driver Version: 510.73.08

[2] One GPC must be disabled when using MIG. Other partitioning options such as 1GPC/6GPCs or 2GPCs/5GPCs are not supported. We first create one GI with 7GPCs and then create CIs consisting of 3GPCs/4GPCs inside of it [4,21].

Table 4. Power cap and partitioning setups

Variable	Selections
P_{total}/P_{max}	350, 400 [W]/500 [W]
P^c_*	100, 125, 150, 175, 200, 225, 250(max) [W]
P^g_*	150, 175, 200, 225, 250(max) [W]
R^c_*	(# of cores for J_1, # of cores for J_2): (2,30), (8,24), (16,16), (24,8), (30,2) (= co-runs), (32,0) (= solo-run, R^c_{max})
R^g_*	(# of GPCs for J_1, # of GPCs for J_2): (3,4), (4,3) (= co-runs), (8,0) (= solo-run, R^g_{max})

studies, as well as a synthetic compute-intensive dense matrix-vector multiplication program (`matvec`). In particular, from the Rodinia benchmark suite, we pick up seven programs that utilize both CPU and GPU extensively/cooperatively. Further, the `matvec` program uses both CPU and GPU in a cooperative manner, i.e., a part of the computation is offloaded to GPU and the rest is performed on CPU. We then create three different job queues (*JobMix1*, *JobMix2*, and *JobMix3*) with different window sizes (W) ranging from 4 to 8. The programs in the queues are selected mutually-exclusively (and randomly for *JobMix1*/*JobMix2*) from the eight benchmarks.

We then generate the training/validation/test datasets by using the benchmarks. More specifically, we randomly select $8 \times 2{=}16$ job pairs out of all the possible ${}_8C_2 = 28$ pairs and measure the co-scheduling slowdowns for each of them while testing 100 different hardware setups that is identical to all the co-run hardware setups that meet $P_{total} = 350$ or 400 [W] in Table 4. To validate the performance model, we divide the dataset in the following way: the first 12 pairs multiplied by 100 hardware configurations (= 2,400 data points) are used for the training and validation; and the rest of the 800 data are used for the inference testing. Note the above division process is based on random pair selections. The training and validation here are corresponding to the offline procedure shown in Fig. 2 in Sect. 3.3.

Neural Network Architecture and Training. Table 7 lists our neural network architecture and training setups based on the general structure described in Sect. 4.1. In our neural network, all the inputs are normalized between 0 and 1 (including the hardware configuration) in order to equalize the significance of them, which ultimately helps the convergence. To normalize the resource partitioning states (R^*_i), we simply pick the first element that represents the number of core or GPC allocation to the first job (J_1), and then divide it by the maximum number of the resource allocation (32 for cores and 8 for GPCs in our environment). We set up two hidden layers to well recognize the patterns in the input values, which is better than relying on one single hidden layer for this purpose. The rectified linear activation function is applied to all the layers except for the input layer, and all the neurons in both the hidden layers and the output have biases. The input layer is fully connected with the first hidden layer in order to use the model while re-ordering the job inputs (see also Sect. 4.1).

In our Python implementation, the training with the dataset described above takes only few minutes, and the slowdown estimations for all the jobs in a job set takes only 1.17 ms in total.

5.2 Experimental Results

Figure 5/6 demonstrate the total execution time comparisons across multiple different scheduling and resource management polices for different total power cap setup ($P_{total} = 350, 400$[W]). The vertical axis indicates the total execution time, while the horizontal axis lists job queues (*JobMix1*-*JobMix3*) in both the figures. The details of the compared policies listed in the legends are as follows: *Time Sharing + Naive Pow Cap* schedules jobs in the time-sharing manner while setting up the power caps to the CPU and GPU equally; *Time Sharing + Opt Pow Cap* also utilizes the time-shared scheduling but the power caps are set to the optimal; and *Our Co-scheduling* schedules jobs and configures the hardware using our proposed approach. As shown in these figures, we achieve significant speedups by up to 67.4% (= (108.8/65.0 − 1)*100) by using our approach compared with *Time Sharing + Naive Pow Cap*. Note the hardware partitioning is done only at the job launches, thus the overhead is negligible here. Table 8 presents the list of job sets created by our approach for each queue under different power capping. The job set selections can change depending on the total power cap setup, which implies our approach can flexibly deal with hardware environment changes, e.g., with changes in the power supply level.

We then compare the measured and estimated execution time (excluding online scheduling time) for different power cap setups in Fig. 7/8. The X-axis indicates the accumulated execution time of all the co-scheduled job sets created from *JobMix3* by using our approach. As shown in the figure, the estimated

Table 5. Collected performance counters (**F**)

Component	Counters and Definitions
CPU	F_1^* = cpu-util, F_2^* = context-switches, F_3^* = page-faults, F_4^* = IPC, F_5^* = stalled-cycles, F_6^* = branch-misses, F_7^* = L1-dcache-load-misses, F_8^* = L1-icache-load-misses, F_9^* = dTLB-load-misses, F_{10}^* = iTLB-load-misses
GPU	F_{11}^* = Memory[%], F_{12}^* = DRAM Throughput[%], F_{13}^* = TEX cache Throughout[%], F_{14}^* = LLC Throughput[%], F_{15}^* = Compute[%], F_{16}^* = Waves per SM, F_{17}^* = Achieved Occupancy[%], and F_{18}^* = Warps per SM

Table 6. Tested job mixes

Name	Job Mix
JobMix1	Q ={gaussian, lud, pathfinder, streamcluster}, $C = 2$, $W = 4$
JobMix2	Q ={gaussian, srad, hotspot, pathfinder, lavaMD, matvec}, $C = 2$, $W = 6$
JobMix3	Q ={gaussian, srad, hotspot, lud, pathfinder, lavaMD, streamcluster, matvec}, $C = 2$, $W = 8$

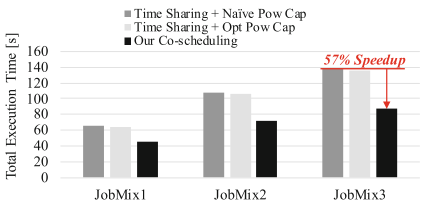

Fig. 5. Execution time comparison ($P_{total} = 350[W]$)

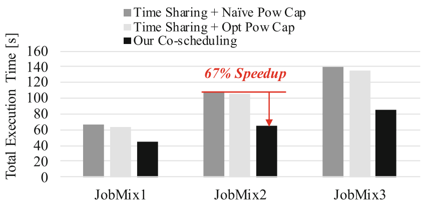

Fig. 6. Execution time comparison ($P_{total} = 400[W]$)

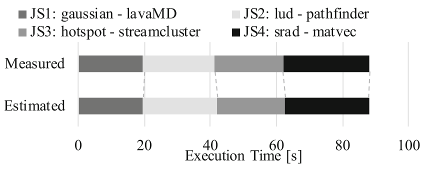

Fig. 7. Comparison of measured vs estimated time (*JobMix3*, P_{total} = 350[W])

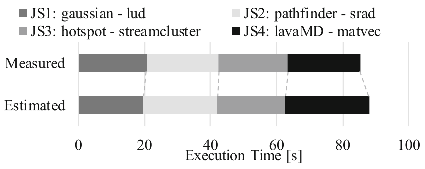

Fig. 8. Comparison of measured vs estimated time (*JobMix3*, P_{total} = 400[W])

execution times are close to the measured ones, and the total estimation error is only 0.4% or 3.1% for $P_{total} = 350$ or 400, respectively. Note that our approach achieves closer performance to the optimal as the error becomes smaller. This is because the Edmonds' algorithm returns the optimal scheduling job sets if the performance estimation is 100% accurate.

Finally, we demonstrate the hardware setup decisions made by our scheduler in Figs. 9/10/11, in particular, for *JobMix3* under the total power cap of P_{total} = 350[W]. The X-axis indicates the job sets created from *JobMix3* by our approach, while the Y-axis represents the breakdown of power caps, core allocations, or GPC allocations in Figs. 9, 10, or 11, respectively. As shown in these figures, these hardware knobs are set very differently in accordance with the characteristics of co-located jobs, including the task size on CPU/GPU,

Table 7. Model and training setups

Type	Parameter List
Model	[**Input layer**]= 4 HW config states (HC) + 18 HW counters (J_1) + 18 HW counters (J_2); [**# of hidden layers**]= 2; [**# of neurons in each hidden layers**]= 18 (= # of HW counters); [**Layer connection**]= Fully connected; [**Activation function**]= Rectified Linear
Training	[**Learning rate**]= 0.001; [**Batch size**]= 4; [**Optimizer**]= Stochastic Gradient Descent; [**# of epochs**]= 200; [**Loss function**]= Mean Square Error

Table 8. Lists of job sets created by our approach

P_{total}	Lists of Job Sets (L_{JS})
350 W	(*JobMix1*): {gaussian-lud, pathfinder-streamcluster}, (*JobMix2*): {gaussian-hotspot, pathfinder-lavaMD, matvec-srad}, (*JobMix3*): {lavaMD-gaussian, lud-pathfinder, hotspot-streamcluster, matvec-srad}
400 W	(*JobMix1*): {gaussain-lud, pathfinder-streamcluster}, (*JobMix2*): {hotspot-lavaMD, srad-pathfinder, matvec-gaussian}, (*JobMix3*): {gaussian-lud, srad-pathfinder, hotspot-streamcluster, lavaMD-matvec}

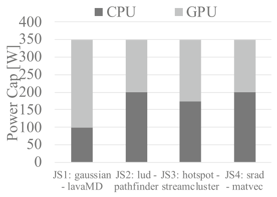

Fig. 9. Power cap setups (JobMix3, $P_{total} = 350[W]$)

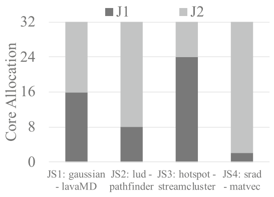

Fig. 10. Core allocation (JobMix3, $P_{total} = 350[W]$)

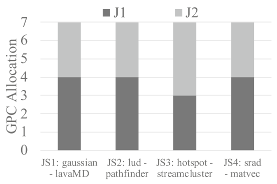

Fig. 11. GPC allocation (JobMix3, $P_{total} = 350[W]$)

the compute/memory intensity, and the interference on shared resources. As our performance modeling can recognize these features well based on the corresponding hardware performance counters and the well-structured neural network, our approach achieves the significant performance improvement by up to 67%.

6 Conclusion

In this paper, we targeted co-scheduling, resource partitioning, and power capping comprehensively for CPU-GPU heterogeneous systems and proposed an approach to optimize them, which consists of performance modeling and a graph-based scheduling algorithm. We demonstrated how a machine learning model, namely a neural network, can successfully be used to predict the performance of co-scheduled applications, while using the application characteristics and partitioning/power states as inputs. We then moved on to the application pair selections where we successfully applied Edmond's algorithm to determine the mathematically optimal pairing. The experimental result using a real system shows that our approach improves the system throughput by up to 67% compared with a time-sharing-based scheduling with a naive power capping that evenly distributes power budgets on CPU/GPU.

Acknowledgements. This work has received funding under the European Commission's EuroHPC and H2020 programmes under grant agreement no. 956560 and was supported by the NVIDIA Academic Hardware Grant Program.

References

1. Top 500 list (2022). https://www.top500.org/lists. Accessed 24 July 2022
2. perf: Linux profiling with performance counters (2022). https://perf.wiki.kernel.org/index.php/Main_Page. Accessed 24 July 2022
3. Arima, E., Hanawa, T., Trinitis, C., Schulz, M.: Footprint-aware power capping for hybrid memory based systems. In: Sadayappan, P., Chamberlain, B.L., Juckeland, G., Ltaief, H. (eds.) ISC High Performance 2020. LNCS, vol. 12151, pp. 347–369. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50743-5_18
4. Arima, E., et al.: Optimizing hardware resource partitioning and job allocations on modern gpus under power caps. In: ICPPW (2022)
5. Azimi, R., et al.: PowerCoord: a coordinated power capping controller for multi-CPU/GPU servers. In: IGSC, pp. 1–9 (2018)
6. Bailey, P.E., et al.: Adaptive configuration selection for power-constrained heterogeneous systems. In: ICPP, pp. 371–380 (2014)
7. Barnes, B.J., et al.: A regression-based approach to scalability prediction. In: ICS, pp. 368–377 (2008)
8. Barrera, S., et al.: Modeling and optimizing NUMA effects and prefetching with machine learning. In: ICS, no. 34 (2020)
9. Bhadauria, M., et al.: An approach to resource-aware co-scheduling for CMPs. In: ICS, pp. 189–199 (2010)
10. Cao, T., et al.: Demand-aware power management for power-constrained HPC systems. In: CCGrid, pp. 21–31 (2016)
11. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: IISWC, pp. 44–54 (2009)
12. Cochran, R., et al.: Pack & cap: adaptive DVFs and thread packing under power caps. In: MICRO, pp. 175–185 (2011)
13. Cook, W., et al.: Computing minimum-weight perfect matchings. In: INFORMS Journal on Computing, vol. 11 (1999)
14. Dennard, R.H., et al.: Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J. Solid-State Circ. **9**(5), 256–268 (1974)
15. Eeckhout, L.: Heterogeneity in response to the power wall. IEEE Micro **35**(04), 2–3 (2015)
16. Greathouse, J.L., et al.: Machine learning for performance and power modeling of heterogeneous systems. In: ICCAD, pp. 1–6 (2018)
17. ïpek, E., et al.: Efficiently exploring architectural design spaces via predictive modeling. In: ASPLOS, pp. 195–206 (2006)
18. Lee, B.C., et al.: Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: ASPLOS, pp. 185–194 (2006)
19. Nagasaka, H., et al.: Statistical power modeling of GPU kernels using performance counters. In: International Conference on Green Computing, pp. 115–122 (2010)
20. NVIDIA: Nsight compute (2022). https://developer.nvidia.com/nsight-compute. Accessed 24 July 2022
21. Nvidia: Nvidia multi-instance GPU (2022). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Accessed 24 July 2022
22. Patki, T., et al.: Exploring hardware overprovisioning in power-constrained, high performance computing. In: ICS, pp. 173–182 (2013)
23. Sarood, O., et al.: Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In: SC, pp. 807–818 (2014)

24. Sasaki, H., et al.: Coordinated power-performance optimization in manycores. In: PACT, pp. 51–61 (2013)
25. Zhu, Q., et al.: Co-run scheduling with power cap on integrated CPU-GPU systems. In: IPDPS, pp. 967–977 (2017)
26. Zhuravlev, S., et al.: Addressing shared resource contention in multicore processors via scheduling. In: ASPLOS, pp. 129–142 (2010)