Profile-based Dynamic Adaptive Workload Balance on Heterogeneous Architectures*

Author names omitted for blind review

ABSTRACT

While High Performance Computing (HPC) systems are increasingly based on heterogeneous cores, the effectiveness of these heterogeneous systems depends on how well the scheduler can allocate workload onto appropriate computing devices and how communications or computations can be overlapped. With different types of resources integrated into one system, GPUs and CPUs, the complexity of the scheduler correspondingly increases. Moreover, for applications with varying problem sizes on different heterogeneous resources, the optimal scheduling approach may vary accordingly.

This paper presents a profile-based dynamic adaptive workload balance scheduling approach to efficiently utilize heterogeneous systems. Our approach follows two phases: first, we build up a dynamic adaptive workload balance algorithm. It can adaptively adjust a workload based on available heterogeneous resources and real time situation. Second, we establish a profile-based device-specific estimation model to optimize the scheduling algorithm. Our scheduling approach is tested on a standard 2D 5-point stencil computation in our Extended EDRT platform[1]. Experimental results show that when the problem size allows the working set to remain within the available GPU memory, our scheduler can obtain speedups up to 8.75× compared to the sequential version, 7× compared to the pure multi-CPU version, and stays on-par with pure GPU version. When the problem size causes the working to no longer fit in the available GPU memory, our scheduler can obtain up to 6× compared to the sequential version, 1.6× compared to the pure multi-CPU version, and 4.8× compared to the pure GPU version.

CCS CONCEPTS

• **General and reference → Performance**; • **Theory of computation → Streaming models**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Synchronization**; **Concurrent programming structures**;

KEYWORDS

Heterogeneous Computing, CUDA, Adaptive modeling, Load balancing

*Produces the permission block, and copyright information

[1]We replaced the real name of the runtime for double-blind review.

ICS 2018, June 2018, Beijing, China

ACM ISBN 978-1-4503-nnnn-n.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Author names omitted for blind review. 2018. Profile-based Dynamic Adaptive Workload Balance on Heterogeneous Architectures. In *Proceedings of ACM International Conference on Supercomputing (ICS 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nowadays, most High-Performance Computing (HPC) platforms feature heterogeneous hardware resources (CPUs, GPUs, FPGAs, storage, etc.). For instance, the number of platforms of the Top500 equipped with accelerators has significantly increased during the last years [12]. In the future it is expected that the nodes of such platforms' heterogeneity will increase even more: they will be composed of fast computing nodes, hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytics, etc. The interconnect of a huge number of such nodes will also lead to more heterogeneity. Resorting to heterogeneous platforms can lead to better performance and power consumption through the use of more appropriate resources according to the computations to perform. However, it has a cost in terms of code development and more complex resource management.

GPU boards are integrated with multi-core chips on a single compute node to boost the overall computational processing power. When scientific applications rely on large amounts of data, this poses some restrictions on how to offload some of the computation to the accelerator device, *e.g.*, a GPU, as their memory capacity is limited, and data transfers incur very limited latencies and bandwidth [11]. CUDA is a high-level platform for developing GPU applications, comprising a compiler, a library, and a runtime system. CUDA applications are organized in kernels, which are functions executed on GPUs.

The main task of a load-balancing mechanism in a heterogeneous system is to devise the best data division among the general-purpose CPUs and the accelerator—in our case, the GPU. Simple heuristic load division, such as evenly dispatch work to all the CPU cores and GPU, may result in worse performance and consume more power.

This work stems from the following observations: with a few exceptions (see Section 5), most work dealing with so-called accelerators, and in particular Graphics Processing Units (GPUs), tend to follow one of two options (1) Fully offload the most compute-intensive parts of a given application on a GPU, or (2) partition the compute-intensive steps between "GPU-friendly" portions that will run solely on the GPU, and "CPU-friendly" ones. In this paper, we attempt to answer the following question: Do we *really* need to use all the computation resources simultaneously? When parts of an application is *co-running*, *i.e.*, when it runs on both CPU and GPU cores, is the resulting performance always higher than running, say, fully on either the GPU or CPUs? As will be shown in the remainder of the paper, there is no clear-cut answer, and it all depends on a wealth of parameters, both hardware and software. Indeed, the question will yield a totally different answer when hardware

architectures, schedulers, applications or even just the problem size of a specific application change.

We consider a compute node is made of *heterogeneous hardware* when it is comprised of general-purpose CPUs, as well as some kind of *accelerator* hardware, such as GPUs or reconfigurable fabric (say, one or more FPGAs). For the purpose of this work, we will focus on GPU-based heterogeneous platforms. We consider two main types of heterogeneous architectures: integrated CPU/GPU architectures and multi-CPUs with discrete GPUs.

Co-running friendly applications, *i.e.*, which can run on both GPU and CPU concurrently, tend to have low memory/communication bandwidth requirements, compared to applications which run the workload on only one part of the system. Hence, the computation-memory ratio and computation patterns can help identify the suitability of the workload to a resource. For example, if the application is characterized such that the communication time (data transfers) between CPUs and GPUs is far higher than the computation time of a given workload, and if there is no overlap between computation and communication, this application will belong to the "co-running unfriendly class." However, the categorization may change when the application is developed in different hardware architectures or even in same hardware architecture with a changing dataset.

Furthermore, we must evaluate if the performance of a friendly co-running application using all available computing resources, *i.e.*, CPUs and GPUs, results in better performance than, say, running on a lower number of compute cores (CPUs and/or GPUs). Many researchers such as Van Craeynest *et al.* [22], J. Power *et al.* [16], V. García *et al.* [7], F. Zhang *et al.* [24], Q. Chen *et al.* [5], or C. Yang *et al.* [23] proved that heterogeneous system architectures are significantly impacted by various parameters, such as number and type of cores, the organization or topology of cores, the memory hierarchy, bandwidth and memory access pattern, the communication congestion and synchronization mechanism, as well as other hardware or software factors.

We propose a profile-based estimation model augmented with a dynamic adaptive workload balance algorithm to help effectively utilize CPU and GPU resources in different heterogeneous platforms. Our approach follows two phases: first, adaptively decide what is the best workload balance based on the application's information, obtained in real time. Second, establish a profile-based machine learning estimation model to obtain a suitable initial workload allocation to resources. Then, our dynamic adaptive workload balance algorithm will run with these information (initial workload on GPU and CPUs, number of resources) and adjust workload based on the real-time system information.

The structure of this document is as follows. Section 2 reviews the main concepts of this work; Section 3 describes our methodology; Section 4 describes our main experimental results; in Section 5 we review the literature about the area and present various related papers. Finally, Section 6 concludes this work and presents the planned future work.

2 BACKGROUND

2.1 Runtime System

We chose to extend EDRT [1, 2, 19] as our test runtime system. It is an implementation of the Codelet Model and the Codelet Abstract Machine (CAM) on which it relies. The CAM models a many-core architecture with two types of cores: synchronization units (SUs), which perform resource management and scheduling, and computation units (CUs), which carry out the computation. A CAM is an extensible, scalable and hierarchical parallel machine model. One SU, one or more CUs, and some local memory make up a *cluster*. Clusters, alongside some memory, can be grouped together to form a chip. One or more chips, coupled with some memory, consist of a node. Finally, a full CAM is composed of at least one node. The communication between and within components of each level of the hierarchy is done by the interconnection network. EDRT is based on the Codelet Model, proposed by paper [26]. It is a fine-grain event-driven program execution model. The Codelet Model is a hybrid between the dataflow [6] and von Neumann models of computation. A codelet *fires* (*i.e.*, is scheduled for execution) when all its dependencies (data and resources) are met.

2.2 Heterogeneous Computing

Heterogeneous computing is about efficiently using all processors in the system, including CPUs and GPUs. In our heterogeneous systems, GPUs have been connected to a host machine by a high bandwidth interconnect such as PCI Express (PCIe). Host and device have different memory address spaces, and data must be copied back and forth between the two memory pools explicitly by the data transfer functions. There are two main aspects to consider when attempting to run an application on both CPUs and GPUs concurrently, namely the communication costs incurred by CPU-GPU communications, and how to efficiently overlap computation and data transfers to reduce said cost.

2.2.1 Heterogeneous Hardware Communication Costs. On integrated CPU/GPU architectures, F. Zhang *et al.* [24] suggested that the architecture differences between CPUs and GPUs, as well as limited shared memory bandwidth, are two main factors affecting co-running performance. On SMP systems connected to GPU architectures, beside architectural differences, communications between CPUs and GPUs are another important factor. GPUs and CPUs are bridged by a PCIe bus. Data are copied from the CPU's host memory to PCIe memory first, and are then transferred to the GPU's global memory. The PCIe bandwidth is always a crucial performance bottleneck to be improved. However, the bandwidth between the CPU host memory and the GPU memory is far less than PCIe bandwidth. Nvidia [14] provides ways to pin memory, also called paged-locked memory. A pinned page can be remapped to a PCIe buffer to eliminate data transfer between the host memory and the PCIe buffer. However, consuming too much page-locked memory will reduces overall system performance.

Servers with multi-cores and multi-accelerator devices are primarily connected by PCI-Express (PCIe) bus. Congestion control mechanisms have a significant impact on communications. Moreover, the PCIe congestion behavior varies significantly depending on the conflicts created by communication. Martinasso *et al.* [13] have explored the impact of PCIe topology, which is one major parameter influencing the available bandwidth. They developed a congestion-aware performance model for PCIe communication. They found that bandwidth distribution behavior is sensitive to the

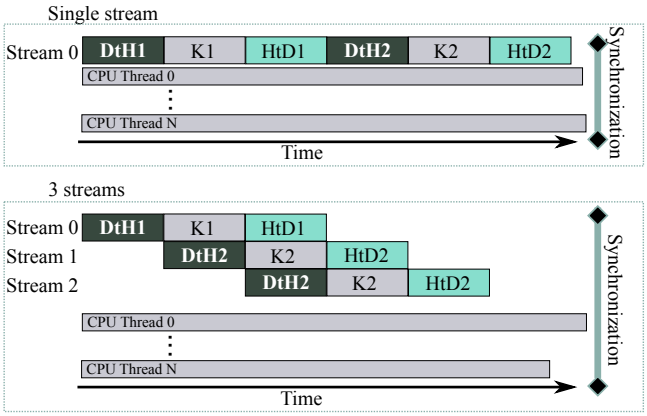

Figure 1: Concurrent Streams overlapping data transfers

transfer message size. PCIe congestion can be hidden if the overlapping communications transfer very small message sizes. However, when the message size reaches some limit, congestion will significantly reduce the theoretical transfer bandwidth efficiency.

2.2.2 Concurrent Streams on GPU. CUDA's programming model provides constructs based on streaming which are capable to schedule multiple kernels concurrently. One CUDA stream can encapsulate multiple kernels, and they have to be scheduled so they strictly follow a particular order. However, kernels from multiple streams can be scheduled to run concurrently. Operations in different streams can be interleaved and overlapped, which can be used to hide data transfers between host and device.

A main optimization of the developed application was to overlap data transfers across the PCIe bus [13]. This is only possible using CUDA streams and pinned memory in the host. Using pinned host memory enables asynchronous memory copies, lowers latency, and increases bandwidth. This way, streams can run concurrently. However, this goal is constrained by the number of kernel engines and copy engines exposed by GPUs, and synchronization must be explicit in the stream kernels.

There are GPUs with only a single copy engine and a single kernel engine. In this case, data transfer overlapping is not possible. Different streams may execute their commands concurrently or out of order with respect to each other. When an asynchronous CUDA stream is executed without specifying a stream, the CUDA runtime uses the default stream 0; but when a set of independent streams are executed with different ID numbers, these streams avoid serialization, achieving concurrency between kernels and data copies.

Figure 1 explains the streaming model which we used to improve the performance of our target GPU application. In this figure we compare the sequential computation of two different kernels with their respective data transfers: one single stream vs. 3 different kernels with their respective data transfers using 3 streams. The second method is only possible in GPUs with two copy engines, one for host-to-device transfers and another for device-to-host transfers.

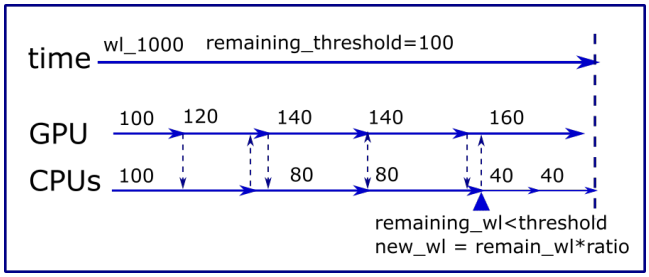

Figure 2: DAWL example

3 METHODOLOGY

In this section, we will describe our approach consisting of two parts: we propose a dynamic adaptive workload algorithm (DAWL) as a first step, and a profile-based machine-learning estimation model as an optimization over it.

3.1 Algorithm: Dynamic Adaptive Workload between Heterogeneous resources

While GPGPU (General-propose computing on GPUs) have evidenced outstanding results in many scientific areas, tools to further capitalize this parallel devices and more refined implementations are still to emerge [14]. Researchers in this area can create applications with a concept of parallelism that are able to run in both CPU and GPU architectures. However, the implementation of an algorithm for parallel execution on both CPUs and GPU does not guarantee it will execute with the same efficiency on both architectures. In particular, data-parallel algorithms dealing with large blocks of data, *i.e.*, featuring intense array arithmetic and regular (array-based) data structures, can greatly benefit when implemented exclusively to run on a GPU [9, 25].

If an application requires the use of a barrier between the host and the accelerator device in a heterogeneous system, a significant effort must be put into load-balancing to achieve high-performance. As a result, the workload behavior for both the host and the device(s) must be carefully analyzed. Among the various parameters that must be gathered, we can cite the number of general purpose and device processing elements, the size and topology of the host's cache hierarchy, the devices's main memory capacity, the PCIe bandwidth, the maximum number of concurrent transfers that can occur between the host and the devices, as well as the problem size to be distributed over both the host and its device(s).

$$GPU_{naive} = \frac{Transfer_{(H \to D)} + Compute_{(D)} + Transfer_{(D \to H)}}{NumThreads_{(D)}} \quad (1)$$

$$CPU_{naive} = \frac{Compute_{(H)}}{NumThreads_{(H)}} \quad (2)$$

$$r = \frac{CPU\_naive}{GPU\_naive} \quad (3)$$

Based on the hardware configuration information of the PCIe, GPU, *etc.*, we can roughly obtain an initial idea by using Equations 1, 2, and 3.

Algorithm 1: Dynamic Adaptive workload balance between Heterogeneous Resources

```
1  Function main(Hardware_Info,WL(problem_size),GPU_WL,CPU_WL,Limit_WL,
2                GPU_Change_ratio,CPU_Change_ratio):
3      step1: decision = hardware_choose(Hardware_Info,WL)
4      step2: if decision = Co_running then
5          Co_running_WL_balance(Hardware_Info,total_WL, GPU_WL,CPU_-
             WL,Limit_WL,GPU_Change_ratio,CPU_Change_ratio)
6      else if decision = CPU then
7          run_CPUs(Hardware_Info,WL)
8      else
9          run_GPU(Hardware_Info,WL)
10     end
11 Function hardware_choose(Hardware_Info,WL):
12     if  WL < GPU_memory_available_size then
13         if r»1 then
14             decision = GPU
15         else
16             decision = CPU
17         end
18     else
19         decision = Co_running
20     end
21 Function
     Run_Func(Hardware_Info,type,WL,Remaining_WL,Limit_WL,Change_ratio):
22     if type = CPU then
23         CPU_Func(Hardware_Info,WL)
24     else
25         GPU_Func(Hardware_Info,WL)
26     end
27     if Remaining_WL<Limit_WL then
28         WL = Remaining_WL;
29     else
30         (faster,Ratio) = check(CPU_status,GPU_status)
31         if faster = CPU then
32             TWL = WL * (1-change_ratio)
33         else
34             TWL = WL * (1+change_ratio)
35         end
36         if Remaining_WL<TWL then
37             WL = Remaining_WL * Ratio
38         else
39             WL = TWL
40         end
41     end
42     Remaining_WL -= WL
43     sync_GPU_CPU(Remaining_WL,MEM)
44     renew(WL_min,WL_max)
45     Run_Func(Hardware_Info,type,WL,Remaining_WL,Limit_WL)
46 Function SYNC_Rebalance_Func(Hardware_Info,
     CPU_WL_Info,GPU_WL_Info):
47     CPU_WL = Rebalance(CPU_WL_Info)
48     GPU_WL = Rebalance(GPU_WL_Info)
49     IsChange = check_Hardware(Hardware_Info)
50     if IsChange=true then
51         CPU_WL = CPU_Update(CPU_WL,Hardware_Info)
52         GPU_WL = GPU_Update(GPU_WL,Hardware_Info)
53     end
```

For the remainder of this paper, we will use a 5-point 2D stencil kernel as a case study. We will leverage it to explain our Dynamic Adaptive Work-Load (DAWL) balance scheduling algorithm below, and outlined in algorithms 1 and 2.

When the stencil problem size is less than the available device memory, $r \gg 1$. Hence, all computation will be carried on the device. When it is larger than the available device memory, Algorithm 1's *hardware_choose* function will choose how the workload will be

Algorithm 2: Asynchronous Parallel function and load balance

```
1  Function Co_running_WL_balance(Hardware_Info,total_WL,
     GPU_WL,CPU_WL,Limit_WL,GPU_Change_ratio,CPU_Change_ratio):
2      GPU_initialize(Hardware_Info,GPU_WL)
3      CPU_initialize(Hardware_Info,CPU_WL)
4      Remaining_WL = total_WL - GPU_WL-CPU_WL
5      do
6          PARALLEL EXECUTION: GPU and CPU
7          GPU: Run_Func(Hardware_Info,GPU,GPU_WL,Remaining_WL,
8                        Limit_WL,GPU_Change_ratio)
9          CPU: Run_Func(Hardware_Info,CPU,CPU_WL,Remaining_WL,
10                       Limit_WL,CPU_Change_ratio)
11         SYNC_Rebalance_Func(Hardware_Info,CPU_WL_Info,
             GPU_WL_Info)
12     while Iteration !=0
```

distributed in a co-running manner. Then, CPU and GPU processing elements can execute the workload concurrently. In addition, in the specific case of our stencil kernel, each time step relies on the results obtained in the previous one. Thus, such a computation requires both the host and the device to synchronize regularly during the application's lifetime. There are two cases which will determine how the workload will be partitioned between the host and the device. First, the static initial workload on GPU, which should be smaller than the available GPU memory; second, we can obtain the initial workload from our estimation model, outlined in Section 3.2. It is based on the compute node's hardware configuration, such as the number of CPU cores, last-level of cache (LLC), the GPU type *etc.* The DAWL scheduling algorithm is meant to minimize workload imbalance between heterogeneous processing elements. To overcome the problem of load imbalance for heterogeneous systems, DAWL dynamically adjusts the workload distribution on different computing resources based on real time information. Algorithm 2 is composed of six steps (steps 3 and 4 are described in Figure 2):

(1) Initialize CPU and GPU configurations, *e.g.*, determine the number of processing elements, their initial workload, *etc.*.
(2) Run the tasks on CPUs and GPU simultaneously.
(3) Monitor the computation. For example, when CPUs finish computing, CPUs will check the GPU status. If computing on CPUs is faster than on GPU, it means the CPUs must be given a larger workload. Hence, the workload will be adjusted for CPUs in the next tasks group. The GPU will be targeted similarly, with a few limitations, *e.g.*, the GPU workload must fit in the device's memory size.
(4) Adapt to borderline cases. When the remaining workload reaches a given threshold, *e.g.*, 10% of the total workload, CPUs or GPU will only take half of remaining workload as next task no matter which one finishes first. The first half will be allocated to whichever set of processing elements finishes first. Thus, the first and second halves may run on different or the same type of cores.
(5) Synchronize and re-balance workload when all the compute tasks in one time step finish. The load-balancing function will check all the workload information, and compute the mean workload for each processing element type. The hardware status will also be re-checked: if the hardware changed, *e.g.*, if several CPUs are not available at this time, the system must adjust the workload on both CPUs and GPU.

(6) Reset CPUs and GPU and free allocated memory.

3.2 Optimization: profile-based machine-learning estimation model

While DAWL can dynamically adjust the workload according to real-time information, an unsuitable initial workload may drag down the performance when the problem size is relatively small. As shown in Figure 3, when problem sizes are close to a specific drop point, an unsuitable initial workload (such as an initial workload on GPU close to the problem size) lowers the performance. It is not a guaranteed behavior; Figure 3 shows the performance of a small initial GPU workload (EDRT-DAWL-2GB-1) can be better than a big initial workload (EDRT-DAWL-2GB-1). However, assuming a small workload will yield better results by default cannot be our golden rule, especially for stencil types of application. Indeed, such applications feature heavy data dependencies, need to be synchronized when partitioning the workload into different tasks. A smaller workload means more communication, as the number of tasks and workloads are in inverse proportion, and the number of tasks and shared data are proportional. A suitable initial workload can help us fully utilize the computing resources with reasonable amounts data transfers.

To solve this problem, we propose a profile-based machine-learning estimation model to optimize DAWL. A black-box machine learning method, *i.e.*, an automatic model without any user intervention, is used. Our estimate model follows three phrases:

(1) Collect hardware architecture information (see Table 1) as parameters to our black-box estimation model.
(2) Collect application runtime profile information as training data (samples).
(3) Utilize the information gathered from the first two steps to build a profile-based estimation model for a given heterogeneous platform, and furthermore obtain customized initial workload on different heterogeneous architectures and how many devices are actually necessary.

In our machine learning black-box estimation model (step 3), different algorithms were tested, including linear regression, logistic regression and random forests. The best match and less computationally complex algorithm was chosen to run our statistical model. To evaluate how well the model fits the data, the statistical measure, $R_{squared}$, is widely used. It is also known as the coefficient of determination. $R_{squared}$ is defined as the percentage of the response variation that is explained by a linear model. $R_{squared} = Explained\ variation/Total\ variation$. *R_squared* is always between 0 and 100%. 0% indicates that the model explains none of the variability of the response data around its mean and 100% indicates that the model explains all the variability of the response data around its mean.

Once the best matched statistical model is obtained, an estimation formula can be built up to predict the application performance on this specific heterogeneous platform. For a given problem size, a minimization multi-variable function can be used to obtain an estimation of the initial workload.

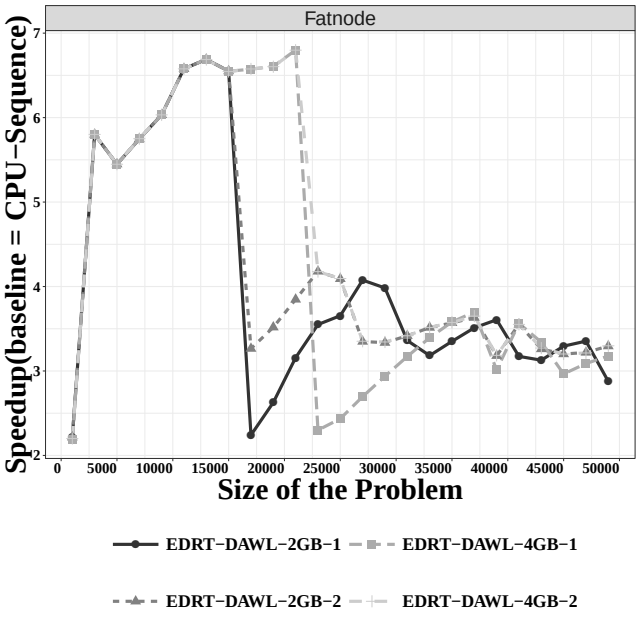

Figure 3: Stencil application: available GPU memory (av_-GPU): 2GB vs 4GB, initial workload (GPU=CPU): $0.5 \times av\_GPU$ **(1) vs** 2000×2000 **(2)**

4 EXPERIMENTAL RESULTS

4.1 Experimental Testbed

4.1.1 Experiment Hardware Overview. Several heterogeneous SMP nodes with multicores were selected for our experiments. They are representative of the current many core-based hardware trends. The main characteristics of those systems are shown in Table 1. Each system is equipped with Intel processors and Nvidia GPUs. `fatnode`'s general purpose CPUs are made of Intel Xeon® E5-2670, with hyper-threading activated, and a Tesla K20® (Kepler architecture with Compute Capability 3.5) board. `Super Micro` sports two Intel Xeon®E7 v2, and embeds 4 Tesla K20 boards®. `CCSL Valinhos` contains one processor Intel i7-4770® and one Tesla K40®. `Debian` has one processor i7-4930K®, with one Nvidia Titan®board and one GeForce GT 630®. `Hive` embeds one Intel i7-3930k® and two GeForce GTX 680®.

4.1.2 Runtime System: Extending EDRT. The original version of EDRT mainly focuses on homogeneous many-core systems. However, this work targets workload balancing between heterogeneous resources. To validate and verify DAWL, we extended EDRT to support heterogeneous architectures. We will only describe the logical model without deep explanations about implementation details in the implementation because of space limitation.[2] To handle GPU tasks, we developed a new type of Codelet, named *GPU-Codelet*, consisting of two parts: CUDA host code and CUDA kernel code. The host and kernel codes can run concurrently or sequentially. To guarantee each code runs on its intended device (GPU or CPU), a

[2]We intend to publish the source code of all our experiments and modifications to EDRT upon acceptance of this paper.

Table 1: Experiment HardWare Environment

Param. / Machines	CPU Parameters					GPU Parameters					PCIe
	CPUThreads	Clock	# Socket	L3 Cache	CPU Mem	# SM	Clock	L2 Cache	GPU Mem	# CE	
1. Fatnode (K20)	32	2.6 GHz	2	20 MB	64 GB (2x32)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
2. Super Micro (K20)	40	3 GHz	2	25 MB	256 GB (2x128)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
3. CCSL (Valinhos) (k40)	8	3.4 GHz	1	8 MB	16 GB	15	0.75 GHz	1.5 MB	12 GB	2	10.3 GB/s
4. Debian (Titan)	12	3.4 GHz	1	12 MB	31 GB	14	0.88 GHz	1.5 MB	6 GB	2	11.5 GB/s
5. Hive (GTX 680)	12	3.2 GHz	1	12 MB	30 GB	8	1.06 GHz	0.5 MB	2 GB	1	5.6 GB/s

GPU-CPU codelet scheduling method was designed. A mechanism was added to monitor CPU-codelets and GPU-codelets, so that the two types of tasks can synchronize with each other.

4.2 Experimental Results: DAWL

DAWL was evaluated on the five systems we described in Section 4.1 (see Table 1 for details). In the following experiments, all systems were configured so that only 2 GB were seen as available by the runtime system. This was intended to evaluated DAWL more fairly, by reducing the "parameters surface" to explore. All our experiments were run using a 5-point 2D stencil computation kernel, with a fixed number of time steps (the convergence test was left out to make the experiments more deterministic). All matrices hold double precision values.

In this section, we adopt a static initial workload methodology to validate and verify the DAWL algorithm. The initial GPU and CPU workload both were 2000×2000 (*rows* $\times$ *columns*).

4.2.1 Performance Analysis: Full Resource Usage. To comprehensively characterize DAWL, we performed a series of workload performance analyses. We compared the `EDRT-DAWL` performance with `GPU-Only`, `CPU-Seq`, `EDRT-CPU`, `EDRT-GPU` executions (see Table 2 for details). Here, `EDRT-DAWL` is the implementation of DAWL on EDRT. Depending on the application, problem size, and hardware information, `EDRT-DAWL` may run on multiple CPUs or GPUs, or be co-running on both CPUs and GPUs.

Figure 4 shows the speedup of different stencil variants, using the `CPU-Seq` version as a baseline. Everything was run on all systems. Figure 5 zooms in Figure 4 for matrix sizes 17000×7000 and onward. `EDRT-GPU` uses the concurrent streaming mechanism described in Section 3.2, for all problem sizes. From Figure 4, we can see the performance of `EDRT-GPU` is stable. We also wrote another variant of `EDRT-GPU`, which uses solely the traditional one-stream method (not shown here due to space limitations). Its performance is very bad when the problem size is larger than the GPU's memory capacity. The `GPU-Only` variant was slightly optimized: When the problem size was smaller than the GPU memory capacity, we used the one-stream method to avoid superfluous synchronizations between the host and device. We used concurrent streaming when problem sizes were larger than the GPU's memory capacity, in order to overlap communication and computation.

Figure 4 shows that, even though there are a lot of differences, overall, `EDRT-DAWL`'s performance drops dramatically at 17000×17000. Based on the *hardware_choose* function (detailed in Algorithm 1), when the problem size is smaller than 17000×17000, the application belongs to the GPU-friendly category and all the workload will be on GPU. Figure 4 validates our solution. On all systems, `EDRT-DAWL`'s performance is very close to `GPU-Only`. If the GPU's

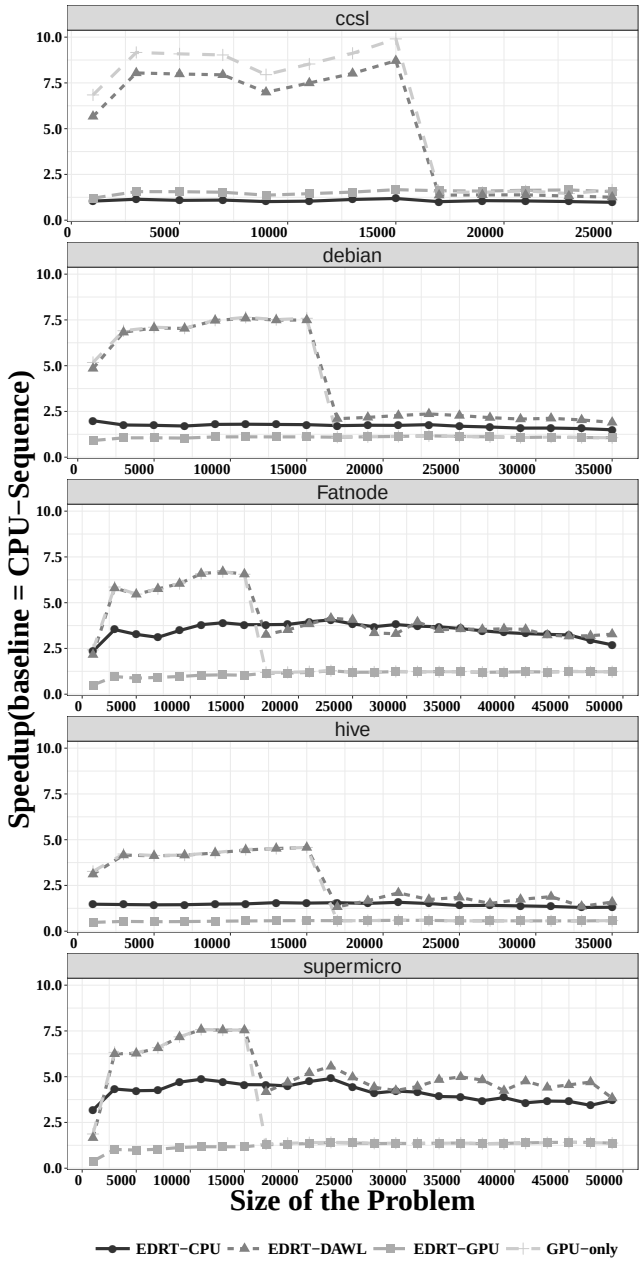

Figure 4: Speedup of the different Stencil versions

Table 2: Stencil kernel implementation

Implementation	Illustration
CPU-Seq	sequential c++ code
GPU-Only	cuda code
EDRT-CPU	multi-threads c++ code
EDRT-GPU	CUDA code on EDRT (concurrent streams)
EDRT-DAWL	DAWL hybrid code on EDRT

memory capacity changes, the inflection point also changes. As shown in Figure 3, the inflection point shifts from 17000 × 17000 to 23000× when the GPU's memory capacity changes from 2GB to 4GB. When problem sizes are larger than 17000 × 17000, the application changes to the co-running friendly category, and computation will run on both CPUs and GPU. Figure 5 shows that the speedup ratios are quite different on different systems with different problem sizes. On the fatnode and Super Micro, EDRT-DAWL and EDRT-CPU are alternately faster; on the CCSL Valinhos, EDRT-DAWL and GPU-Only are similar; on Hive and debian, EDRT-DAWL is faster than EDRT-CPU.

From Figure 3, we can observe the initial workload significantly affects performance when problem sizes are relatively small; their impact is reduced when the problem size keeps increasing. This is because the bigger the problem size, the more time EDRT-DAWL has to balance the workload. Except for the initial workload, our profile-based analytical model can also help us obtain the competitiveness change ratio. We will describe it in Section 4.3.

4.2.2 Performance Analysis: Varying the CPU Number. Figure 6 shows the experimental results related to our second question: do we really need all the computation resource simultaneously when we run a co-running friend application? Due to page limitations, we only show one system running with different threads. This servers stands for a classical type of hardware configuration: a "regular" GPU and a two-way SMP chip multiprocessing system. We used two different strategies to pin threads to physical processing elements: *spread* and *compact*. According to the hardware topology gathered by the runtime system, the *spread* strategy attempts to allocate software threads to processing elements (*i.e.*, a physical core or thread) as far from each other as possible. On the contrary, the *compact* strategy attempts to allocate software threads as closely as possible on the available processing elements. *Spread* can potentially help performance when fewer threads are involved in the computation, as each thread can benefit from more shared LLC or private L1-L2 cache space. However, this may incur a higher overhead if threads are sharing a fair amount of data. On the contrary, the *compact* pinning strategy ensures that locality is maximal, but may end up under-utilizing the underlying hardware if too few threads are involved in the overall computation. Figure 7 shows the topology of fatnode.

Even though the *compact* and *spread* methods affect plenty the performance, the rough trend is the same. When the threads number reaches its limit, increasing the number of threads does not improve performance—which is expected, because of memory conflicts. Section 4.3 will detail how our analytical model can help

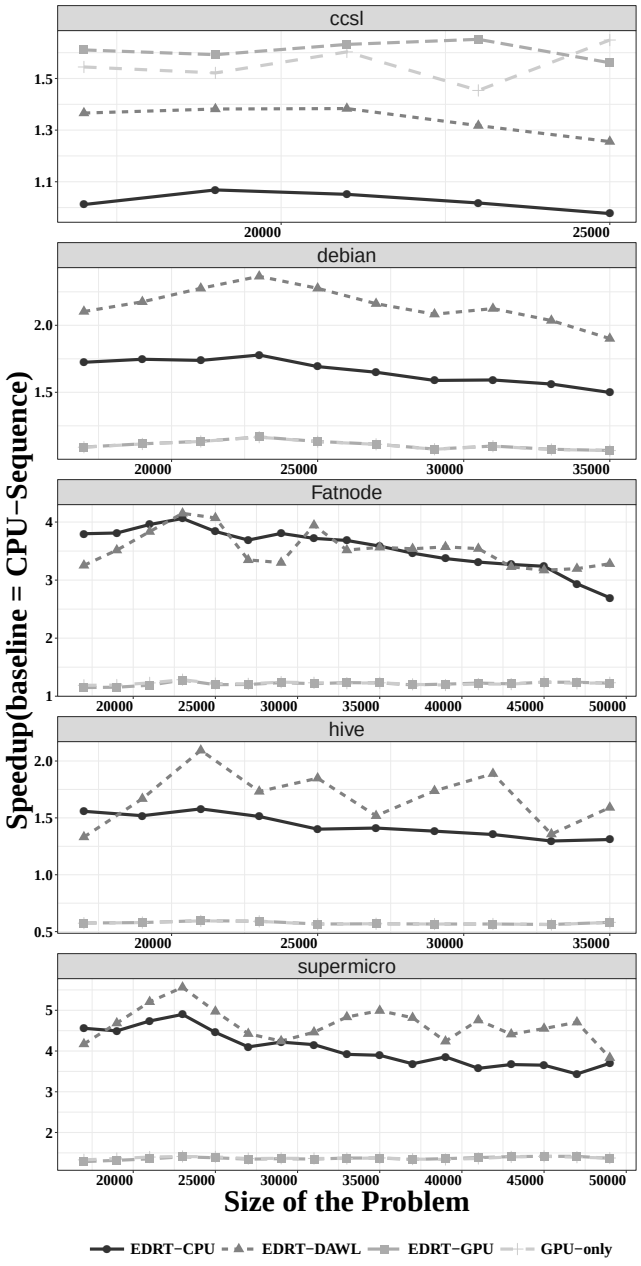

Figure 5: Speedup of the different Stencil versions of matrices higher than 17K

obtain a suitable threads number based on the application and hardware configuration.

4.3 Experimental Results: Profile-based Machine-learning Estimation Model

As we describe in Section 3.2, to build up a profile-based Machine-learning estimation model for heterogeneous architecture, collecting data is the first step. Sampling method is good way to

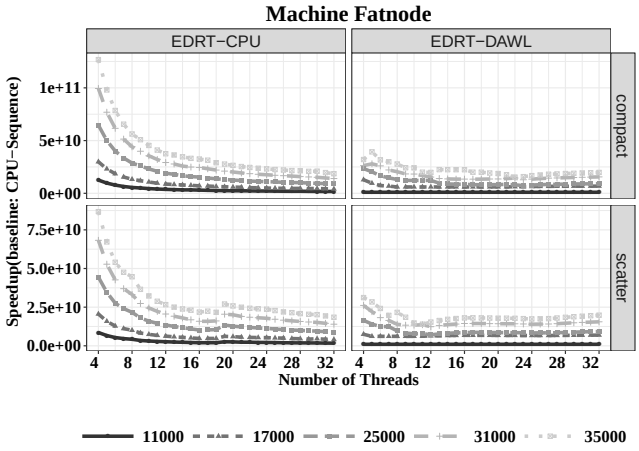

Figure 6: Performance with a varying number of general-purpose processing elements on `fatnode`

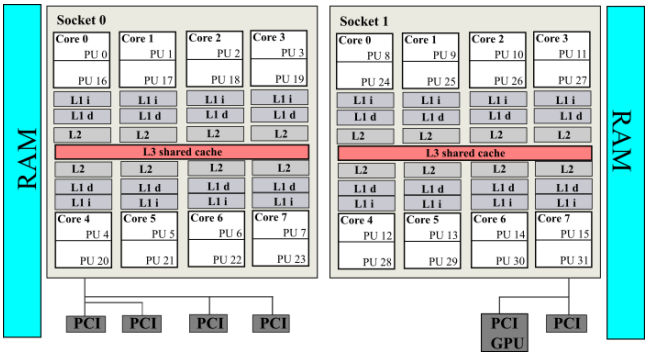

Figure 7: Topology of `fatnode`

obtain both training data and test data. Our target application is co-running friendly application corresponding to stencil code which problem size are larger than 17000*17000 with 2GB available GPU memory. Our approach is to utilize small problem size data to predict big problem size execution time and data with small time step(iteration) to predict large time step. Total sample set including test and training data set consists of several parts: initial GPU workload (1000,2000,4000,8000), initial CPU workload $GPU\_workload * (0.5, 1.0, 1.5, 2.0)$, iteration (1,2,8) problem size (starting from 17000 to 35000 with step of 2000), and CPU core number (4,8,16,..largest CPU core number on the server). The sample sets are a little bit of different on different servers, such as the largest ccsl server problem size sample will be 25000 since main memory is smaller than the others. Total sample on 5 servers is 6480.

Second step is to obtain the best matched model. multiple regression algorithms (linear regression, logistic regression) with multiple Meta-functions and Polynomial functions, and ensemble learning method (random forests) *etc.* are test. Finally, the best fit mode is the simple linear regressions (Meta-function) whose $R\_squared$ are

Table 3: Mean Absolute Percentage Error (MAPE)

Machines	supermicro	fatnode	debian	hive	ccsl
MAPE	7.41%	6.43%	1.68%	3.49%	3.45%

between 93 and 94%, Figure 8 explains the variance of the data in this percentage.

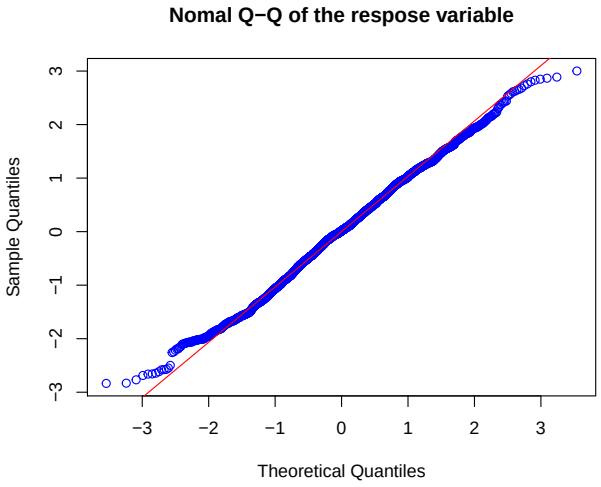

Figure 8: Quantiles of the model with all the samples

To measure the progress of the learning algorithm we have used the Mean Absolute Percentage Error (MAPE). With this error we have analyzed the reliability of our approaches. Table 3 shows Mean Average Percentage Error (MAPE) of the linear model for each machine.

As a second goal in this statistical estimation model, we wanted know which parameters had more impact in the construction of the model. For this reason, we used the absolute value of the $t-statistic$ for each model parameter and calculate the importance of each parameters in the model, we made this experiment with 17 software and hardware parameters, the result was that 6 parameters are enough to predict the performance of the developed Stencil application, these were: NumOfSocket, CPU_Clock, CPU_Cores, Total_Workload, Initial_GPU_Workload, CPU.GPU_Workload_Ratio. Except 3 workload related parameters, 3 other parameters are all related to the CPUs. In our case, GPUs in 5 servers are pretty similar, that is the reason why GPUs parameters do not have relevance in our model. The situation will change if more GPUs type are involved in the experiments. GPUs and PCIe information were hidden in parameter Workload parameters.

Finally, by using estimation model(execution estimation formula) and mathematical optimization method (minimize multi-variable function), the best initial workload on GPU and CPU, most efficient device number can be obtained.

Combining our profile-based estimation model and DAWL, we obtained the the results shown in Figure 9. Compared to `EDRT-CPU`, which always uses all the CPUs, this new implementation of `EDRT-DAWL`

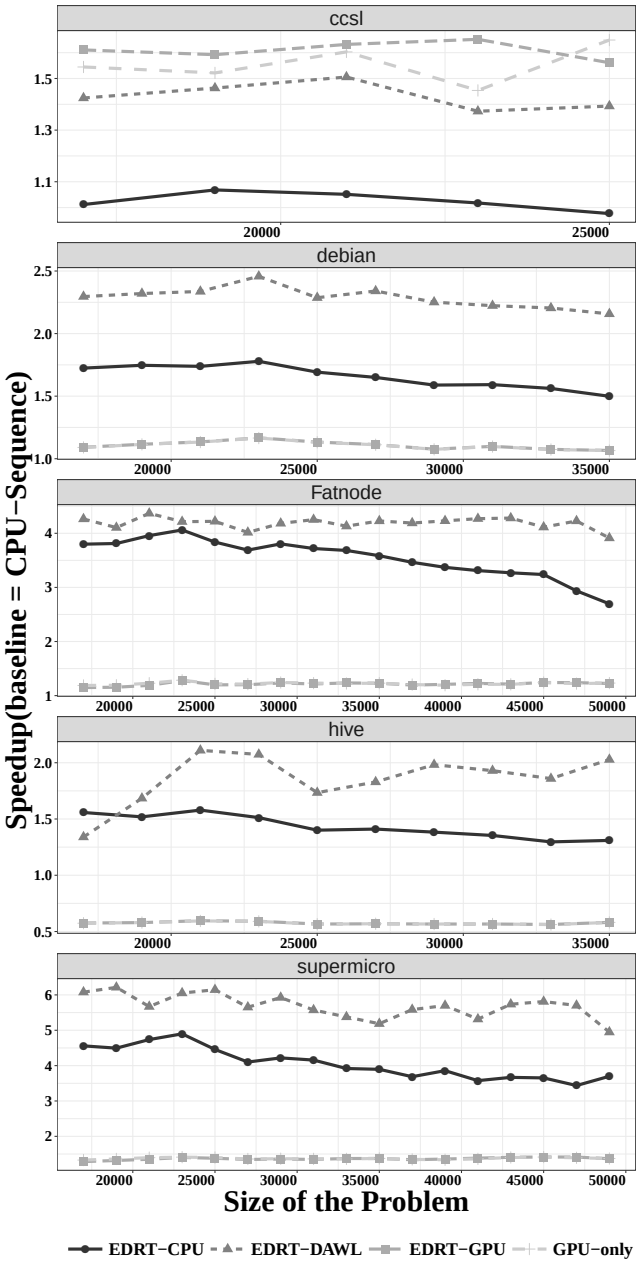

Figure 9: Speedup of the different Stencil versions of matrices higher than 17K

uses at most half of the CPUs (depending on the hardware). Our scheduler can obtain up to 6× speedup compared to the sequential version, 1.6× speedup compared to the multiple core version, and 4.8× speedup compared to the pure GPU version.

5 RELATED WORK

The efficient utilization of the computing capacity offered by CPU-GPU heterogeneous resources is a challenging problem. Moreover, with the development of chip multiprocessing technology, new types of heterogeneous platforms, equipped with a deep memory hierarchy system, Non-Uniform Memory Access (NUMA) and multiple different types computation devices has become very common in the academic and industry areas, and even in daily life. As a result, a number of compilers, runtime systems, scheduling methods and frameworks [3, 8, 10, 17, 21] have been introduced.

Teodoro *et al.* [21] have proposed and implemented a performance variation aware scheduling technique along with an estimation optimization model to collaboratively use CPUs and GPUs on a parallel system. A number of scheduling methods or library [15, 18, 20] were combined with StarPU [3], a task programming library for hybrid architectures based on task-dependency graphs, to perform scheduling and handle task placement in heterogeneous systems. In StarPU, the user provides different kernels and tasks for each target device and specifies inputs and outputs of each task. The runtime ensures that data dependencies are transfered to the devices for each task. Furthermore, the user can specify explicit dependencies between tasks.

Panneerselvam and Swift proposed Rinnegan [15], a Linux kernel extension and runtime library, and implemented and validated that decisions of where to execute a task must consider not only execution time of the task, but also current heterogeneous system conditions. Sukkari *et al.* [20] proposed an asynchronous out-of-order task-based formulation of the Polar Decomposition method to improve hardware occupancy using fine-grained computations and look-ahead techniques.

The main challenge of the load-balancing mechanism is to precisely divide workload on processing units. A simple heuristics devision approach may result in worse performance. Belviranli *et al.* proposed a dynamic load-balancing algorithm for heterogeneous GPU clusters named the Heterogeneous Dynamic Self-Scheduler (HDSS) [4]. Sant'Ana *et al.* described a novel profile-based load-balance algorithm [18] named PLB-Hec for data-parallel applications in heterogeneous CPU-GPU clusters. PLB-HeC algorithm performs an online customized estimation of performance curve models for each devices (CPU or GPU). Like a typical data-parallel application, data in PLB-HeC is divided in blocks, which can be concurrently processed by multiple threads. The granularity of the block size for each processing unit is crucial for performance: incorrect block sizes will produce idleness in some processing units and reduce performance. To get good block sizes, PLB-HeC solves the problem in three phrases: first, it dynamically computes performance profiles for different processing units at runtime; second, using a non-linear system model, they determine the best distribution of block size among different processing units; third, they rebalance the block size during execution. PLB-Hec obtained higher performance gains with more heterogeneous clusters and larger problems sizes.

All the works presented above rely on StarPU to implement their various strategies. Of all of them, Belviranli *et al.* and Sant'Ana *et al.*'s work are probably closest to our own (they rely on online profiling, or resort to some machine-learning techniques to perform load-balancing decisions). However, this work and most of the previous ones tend to focus on loosely synchronized parallel workloads, where specific tasks are often run only a specific type of processing element (*e.g.*, CPU or GPU). On the contrary, our work

focuses on workloads that are iterative in nature, feature heavy data dependences, and require regular and possibly frequent synchronization operations between the device and the host. The work itself is "homogeneous," but it can be run on either the host or the device, depending on their state of idleness, the remaining work size to perform, *etc.*

6 CONCLUSIONS AND FUTURE WORK

The performance of HPC system on heterogeneous many-core system dependents on how well a scheduling policy allocates its workload its processing elements. An incorrect decision can degrade performance and waste energy and power. This paper makes the following contributions: first, a dynamic adaptive workload balance (DAWL) scheduling algorithm is proposed. It can adaptively adjust workload based on available heterogeneous resources and real time information; second, a profile-based machine-learning estimation model is built to optimize our scheduler algorithm. The estimation model can obtain a customized initial workload on various heterogeneous architectures, as well as how many devices are necessary. With these information, DAWL can reduce its adaptation time. As a case study, we used a 5 points 2D stencil and ran it on an extended implementation of the EDRT runtime system. We reported that, when the workload fit in the GPU memory, we could get speedups as high as 8.75× compared to the sequential baseline, 7× when comparing to a pure multicore version, and stay on-par with a pure GPU version. When the workload size goes beyond the GPU memory capacity, our scheduling system can reach speedups up to 1.6× the pure multicore version, and 4.8× compared to the pure GPU version.

We plan to augment our model with power-consumption parameters to enrich a profile-based DAWL and determine good trade-offs between performance and power on heterogeneous architectures. This will result in a more complex analytical model.

REFERENCES

[1] Authors omitted for blind review purposes 2017. Title omitted for blind review purposes. *ACM Trans. Archit. Code Optim.* 14, 4, Article 47 (Dec. 2017), P pages.

[2] Authors omitted for blind review purposes 2017. Title omitted for blind review purposes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. P–P.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (Feb. 2011), 187–198. https://doi.org/10.1002/cpe.1631

[4] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. 2013. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Trans. Archit. Code Optim.* 9, 4, Article 57 (Jan. 2013), 20 pages. https://doi.org/10.1145/2400682.2400716

[5] Q. Chen and M. Guo. 2017. Contention and Locality-Aware Work-Stealing for Iterative Applications in Multi-socket Computers. *IEEE Trans. Comput.* PP, 99 (2017), 1–1. https://doi.org/10.1109/TC.2017.2783932

[6] J. B. Dennis. 1974. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, London, UK, UK, 362–376. http://dl.acm.org/citation.cfm?id=647323.721501

[7] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. 2016. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581277

[8] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. 2015. A Framework for Application-Guided Task Management on Heterogeneous Embedded Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 42 (Dec. 2015), 25 pages. https://doi.org/10.1145/2835177

[9] Benedict R. Gaster and Lee Howes. 2012. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer* 45 (2012), 42–52. https://doi.org/10.1109/MC.2012.257

[10] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 1299–1308. https://doi.org/10.1109/IPDPS.2013.66

[11] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE Computer Society, Washington, DC, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[12] TOP500 Supercomputer List. [n. d.]. http://www.top500.org (visited on Nov. 2017). ([n. d.]).

[13] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas C. Schulthess, and Torsten Hoefler. 2016. A PCIe Congestion-aware Performance Model for Densely Populated Accelerator Servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=3014904.3014989

[14] NVIDIA. 2015. *CUDA C: Programming Guide, Version 7.*

[15] Sankaralingam Panneerselvam and Michael Swift. 2016. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 373–386. https://doi.org/10.1145/2967938.2967964

[16] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 457–467.

[17] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. 2010. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/1810085.1810106

[18] L. Sant'Ana, D. Cordeiro, and R. Camargo. 2015. PLB-HeC: A Profile-Based Load-Balancing Algorithm for Heterogeneous CPU-GPU Clusters. In *2015 IEEE International Conference on Cluster Computing*. 96–105. https://doi.org/10.1109/CLUSTER.2015.24

[19] Authors omitted for blind review purposes 2013. Title omitted for blind review purposes. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, P–P.

[20] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes. 2017. Asynchronous Task-Based Polar Decomposition on Single Node Manycore Architectures. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2755655

[21] G. Teodoro, T. M. Kurc, T. Pan, L. A. D. Cooper, J. Kong, P. Widener, and J. H. Saltz. 2012. Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 1093–1104. https://doi.org/10.1109/IPDPS.2012.101

[22] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 213–224. http://dl.acm.org/citation.cfm?id=2337159.2337184

[23] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. 2010. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *2010 IEEE International Conference on Cluster Computing*. 19–28. https://doi.org/10.1109/CLUSTER.2010.12

[24] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. 2017. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (March 2017), 905–918. https://doi.org/10.1109/TPDS.2016.2586074

[25] Z. Zhong, V. Rychkov, and A. Lastovetsky. 2012. Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*.

[26] Authors omitted for blind review purposes 2011. Title omitted for blind review purposes. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11)*. ACM, New York, NY, USA.