International Journal of Parallel Programming (2022) 50:115–151
https://doi.org/10.1007/s10766-021-00721-2

A Profile-Based AI-Assisted Dynamic Scheduling Approach for Heterogeneous Architectures

Tongsheng Geng[1] · Marcos Amaris[2] · Stéphane Zuckerman[3] · Alfredo Goldman[4] · Guang R. Gao[5] · Jean-Luc Gaudiot[1]

Received: 16 December 2020 / Accepted: 13 July 2021 / Published online: 23 August 2021

Abstract

While heterogeneous architectures are increasing popular with High Performance Computing systems, their effectiveness depends on how efficient the scheduler is at allocating workloads onto appropriate computing devices and how communication and computation can be overlapped. With different types of resources integrated into one system, 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. Thus, we introduce a Profile-based AI-assisted Dynamic Scheduling approach to dynamically and adaptively adjust workloads and efficiently utilize heterogeneous resources. It combines online scheduling, application profile information, hardware mathematical modeling and offline machine learning estimation modeling to implement automatic application-device-specific scheduling for heterogeneous architectures. A hardware mathematical model provides coarse-grain computing resource selection while the profile information and offline machine learning model estimates the performance of a fine-grain workload, and an online scheduling approach dynamically and adaptively distributes the workload. Our scheduling approach is tested on control-regular applications, 2D and 3D Stencil kernels (based on a Jacobi Algorithm), and a data-irregular application, Sparse Matrix-Vector Multiplication, in an event-driven runtime system. Experimental results show that PDAWL is either on-par or far outperforms whichever yields the best results (CPU or GPU).

Keywords Heterogeneous many-core computing · Workload balance · Adaptive modeling · Machine learning assisted scheduling · Parallel computing

Extended author information available on the last page of the article

1 Introduction and Motivation

Nowadays, most High-Performance Computing (HPC) platforms feature heterogeneous hardware resources such as CPUs, GPUs, FPGAs, *etc.* [25]. In the future, the nodes of such platforms are expected to be even more heterogeneous. They will feature side-by-side, fast and slow computing units mixed with accelerators, I/O nodes, quantum technology [13], among others. Heterogeneous platforms must offer the promise of both better energy efficiency and performance. However, this comes at a high cost in terms of code development and resource management.

Parallel computing models and architectures have all increased in usage and importance since their emergence. The heterogeneity of actual platforms complicates the task of optimizing parallel computing programs if done by hand. This is a strong motivation for the development of automated tools and techniques for program optimization.

Indeed, even with successive generations of large-scale scientific HPC systems, data generation has grown faster than compute capabilities, which means that dealing with data-intensive applications has become a crucial challenge in scientific domains [6]. The integration of data analytics, *e.g.,* Machine Learning, and exascale computing have been hailed as the fourth paradigm of science [33].

Meanwhile, whole sectors of scientific computing continue to rely on iterative algorithms. In particular, Stencil-based computations are at the core of many essential scientific applications: Stencils are used in image processing algorithms, *e.g.,* convolutions; partial differential equation solvers, Laplacian transforms, or computational fluid dynamics [21], digital signal processing [15], linear algebra [1], *etc.* More specifically, the Jacobi iterative method has been proposed to solve sparse triangular systems arising from incomplete Cholesky preconditioning [39]. A diverse set of realistic symmetric positive definite test problems have proved that Jacobi iterations are useful for an extensive range of problems [9].

Other kernels are also used in iterative algorithms, such as sparse matrix-vector multiplications (SpMV). Unlike Stencil (regular computing per row/column), the individual work-items of SpMV exhibits a different computational load profile since the numbers of non-zero elements per row may vary significantly.

However, both Stencil and SpMV can be classified as *co-running* algorithms and can be executed on heterogeneous systems. *Co-running* has been defined by Zhang *et al.* [46] as follows: applications can be decomposed into multiple tasks and the system allows these tasks to run on CPUs or general-purpose accelerators simultaneously, *e.g. GPU*, to process different parts of the same input data. The challenge lies in how multiple Stencil or SpMV tasks can be assigned to CPUs and GPUs concurrently to increase performance.

Our research is based on the following observations: most work dealing with accelerators—GPUs—has followed one of two paths: (1) most of the compute-intensive parts of applications are fully offloaded to a GPU, or (2) the workload is statically partitioned between CPUs and GPUs, with each partition running independently. Some exceptions are listed in Sect. 5. This paper presents a novel approach to the dynamic scheduling of tasks on heterogeneous systems. It is based

on a profile-based Artificial Intelligence approach and explores parallelism on GPU-based heterogeneous platforms.

The key contribution of our work is in providing a complete solution which combines profile information, a hardware resource mathematical model, online scheduling and offline machine learning to dynamically and adaptively distribute tasks onto CPUs and/or GPU, and ultimately, increase performance and lower energy consumption. Furthermore, we demonstrate how this solution can be utilized for multiple applications running on different hardware platforms. Our Profile-based Iterative Dynamic Adaptive WorkLoad Balance (PDAWL) approach for heterogeneous architectures has the following characteristics:

1. By leveraging an online scheduler, it can dynamically and adaptively adjust the workload based on the (dynamic) run-time situation, (static) information about the hardware platform, and a performance-workload estimation model (communication *vs.* computation) provided by an offline machine learning approach. Combining online and offline information improves flexibility and accuracy.
2. It follows an event-driven approach and employs multiple levels of granularity for the synchronization to explore tasks parallelism and flexibility of scheduling.
3. It employs a pure CPU and pure GPU machine learning estimation model to predict the performance of the heterogeneous model.
4. It trains small workload tasks to predict the performance of middle or large workload tasks.
5. It can be utilized to dynamically and adaptively schedule co-running applications, such as Stencil (Jacobi algorithm) and SpMV discussed in this paper, on heterogeneous platforms. Stencil has been selected for being a representative of regular data processing, while SpMV corresponds to irregular data processing.
6. It, and more specifically the Profile-based Machine learning (ML) estimation model, provides optimization suggestions for specific applications on heterogeneous systems.

The rest of the paper is organized as follows: Sect. 2 reviews the main concepts of this work; Sect. 3 describes our methodology; Sect. 4 focuses on our main experimental results; in Sect. 5, we review the literature pertinent to our work and related papers. Finally, Sect. 6 concludes this work and presents the planned future work.

2 Background

To implement our profile-based dynamic and adaptive workload scheduling system (PDAWL), we must leverage an efficient runtime system, presented in Sect. 2.1, take advantage of the computing potential offered by heterogeneous hardware, and in particular, GPUs, as described in Sect. 2.2, and explore the parallelism of different types of applications based on hardware features, see Sect. 2.3.

2.1 Codelet Model and Runtime System

The Codelet Model [48] is an event-driven execution model where a codelet is a non-interruptible sequence of instructions that runs until completion. It is *enabled* when all of its data dependencies are satisfied and *ready* when its resource dependencies are also satisfied.

The Codelet Abstract Machine (CAM) describes the mechanism on which codelets are allocated, stored, and scheduled. The CAM models an extensible, scalable and hierarchical parallel many-core architecture with two types of units: synchronization units (SUs), which perform resource management and scheduling, and computation units (CUs), which carry out the computation. CUs and SUs are grouped into several clusters where they can benefit from data locality. DARTS [3, 37] is a runtime system implementing the CAM. It maps "abstract cores," CUs and SUs, to physical processing elements (PEs).[1]

We extended DARTS from the basic homogeneous system to a more general heterogeneous many-core system. Heterogeneous DARTS specifies two types of codelets: `CPU_Codelets`, and `GPU_Codelets`, which can run simultaneously. `CPU_Codelets` are "regular" user-level data-driven tasks, destined to run on general-purpose CPUs. `GPU_Codelets`, however, are meant to run on a GPU, and as a result, must explicitly deal with not only computation but also data movement.

2.2 Heterogeneous Computing

In this work, we are considering CPU–GPU heterogeneous systems where GPU devices are connected to a host machine via a PCI Express (PCIe) bus. Host and devices have different memory address spaces. Data must be explicitly transferred between the memory pools. The execution flow of a heterogeneous application can be divided into three key stages. First, the host transfers data to the memory of the GPU; second, the main program executed on the CPU (the host) is responsible for starting threads in the GPU (the device) and launching a function (the kernel). Finally, the device sends results back to the host. Since communication is typically expensive in such systems, the main goal is to minimize the effect of the CPU–GPU communication overhead by fostering an overlap between communications and computations.

2.2.1 Heterogeneous Hardware Communication

Lee *et al.* [23] analyze a set of important high throughput computing kernels on both CPUs and GPUs. They show the differences of optimization features contributing to performance improvement on these architectures. The paper concluded that CPUs can have comparable performance to GPUs if the application's code is properly optimized (*e.g.*, loops are tiled, skewed, *etc.*). Further, GPUs and CPUs are bridged by a PCIe bus, allowing high-throughput communications between the host's global memory and the accelerator's local memory. Hence the

[1] PEs can be either physical or logical cores, *e.g.*, hardware threads in an SMT architecture.

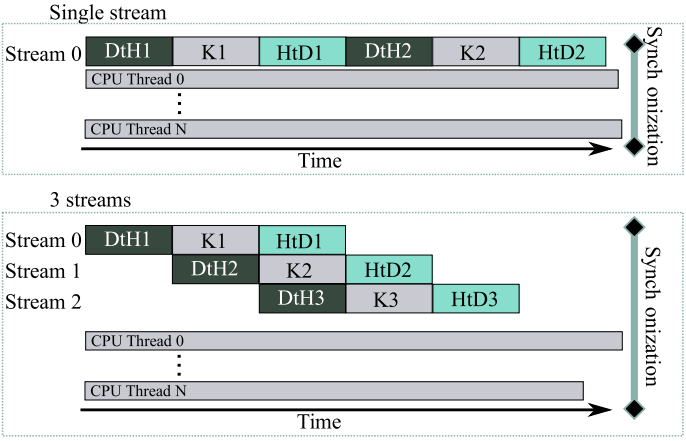


Fig. 1 Concurrent streams overlap data transfer (Color figure online)

PCIe bandwidth is always a crucial performance bottleneck ripe for improvement. Nvidia provides ways to use page-locked host memory to lower data transfer latency [31]. However, performance may be degraded if the allocated pinned memory is too large. Moreover, PCIe congestion behavior varies significantly depending on the conflicts created by communication. Martinasso *et al.* have explored the impact of the PCIe topology, a significant parameter affecting the available bandwidth [29].

This work focuses only on a single GPU per machine, leaving any PCIe topology aspects to future work.

2.2.2 Concurrent Streams on GPUs

CUDA provides stream-based constructs since version 7. This functionality allows the programmer to schedule multiple computing kernels concurrently. It lets the accelerator efficiently overlap computation and communication with the host.

Figure 1 illustrates the CUDA streaming model. We compare the sequential computation of two different kernels with their respective data transfers: one single stream *vs.* three different kernels with their respective data transfers using three streams.[2]

2.3 Data-Regular and Data-Irregular Computations in Heterogeneous Platforms

In co-running applications, the workloads can be decomposed into multiple tasks and run on different Processing Elements (PEs for both CPUs and GPU). Typically, GPUs run regular computations very efficiently, but perform poorly with irregular

[2] The second method is only possible in GPUs with at least two copy engines, one for host-to-device transfers and another for device-to-host transfers. If four copy engines are involved, stream0 and stream1 can be run parallel.

computations [10]. CPUs perform reasonably well for both, provided SIMD instructions and thread parallelism are correctly exploited. For applications containing both data-regular and data irregular computations, it will be preferable to split regular and irregular computing and run them on suitable PEs: allocate the regular parts to the GPU and the irregular/regular part to the CPU. More discussion related to regular/irregular computing can be found in the discussion of SpMV (Sects. 4.1.2 and 4.3).

3 Methodology

In this section, we start by providing the context of the problem, then describe our scheduling approach for heterogeneous architectures. The latter is described in two parts: DAWL, an adaptive workload scheduling approach; and PDAWL, which builds on top of DAWL by combining a profile-based machine learning estimation model with DAWL. DAWL is based on an online scheduling approach using a hardware resource model, and follows a rather coarse-grain approach; PDAWL builds on top of DAWL to allow for automatic fine-grain resource scheduling with hardware-specific considerations to deliver higher performance.

3.1 Problem Analysis

In heterogeneous systems, accelerator devices[3] (*e.g.*, GPUs) and hosts (*i.e.*, CPUs) play different roles: the accelerator is often seen as the "main computational power" of a compute node; and the host as either the "control unit" handling I/O communications and tasks scheduling, or as the "processing unit" responding to parallel computing requests. Currently, most systems work either by fully offloading the workload to accelerators or by statically partitioning the workload between host and accelerators, and running these partitions independently (more details can be found in Sect. 5). However, these two approaches used to statically partition workloads may cause multiple issues, *e.g.,* synchronization and waiting times, low resource utilization *etc.*, at run time, which will incur a dramatic drop in performance, and increase the total power consumption [14].

Several strategies have been proposed to overcome these issues: the first is to build mathematical models that can estimate the execution time of tasks on different computing resources and then statically allocate the corresponding workload onto hosts and accelerators. However, multiple issues are in the way: (1) building an accurate estimation model needs to consider both hardware devices and application features, while the growing variety of hardware devices and their combinations tremendously increases the difficulty. (2) It will be highly difficult to build such a model for current high-performance hardware components, such as memory hierarchy, prefetch mechanisms, Direct Memory Access (DMA), PCIe [2], [41], *etc.*; (3) *any* change in the hardware configuration may cause great performance variations, hence requiring the model to be rebuilt. (4) The complexity and

[3] In this paper, we use interchangeably the terms "device" or "accelerator."

fallibility of building a mathematical data transfer model of CUDA concurrent streams [41] widely increases with the introduction of emerging hardware, the synchronization method between host and device, *etc.* (5) A static model cannot capture runtime situations, which is another important factor that affects the accuracy of the performance estimation model. (6) Such a model may work well for coarse-grain workload partitioning, but may not be useful for fine-grain workload partitioning, which plays a pivotal role in attaining high performance on a heterogeneous system.

The second approach employs dynamic workload partitioning, which theoretically can dynamically allocate workloads onto both accelerators and host at runtime. However, synchronization and waiting time issues still occur if: (1) the synchronization and partitioning mechanisms are not matched; (2) the partitioning is not suitable for available computing resources; (3) the communication costs between accelerator and host or among accelerators are too high; (4) we must run different types of application (memory-intensive *vs.* compute-intensive) onto the different hardware configuration, as described in the paragraph describing hardware resource modeling, with unsuitable partitioning; (5) the granularity (coarse *vs.* fine grain) of the workload partitioning may be unsuitable; *etc.*

To solve the issues we just listed when modeling resources and scheduling work dynamically, the application behavior on both the host and accelerator must be carefully analyzed [14, 17, 31, 47]. We propose our approach, DAWL, as well as an optimized version, PDAWL, which will accommodate the features of both the application and the hardware resources to ensure that an application can run efficiently on heterogeneous systems. A dynamic adaptive workload (DAWL) scheduler follows an adaptive and dynamic workload partitioning approach, based on a coarse-grain model (see Sect. 3.2), while PDAWL follows a profile-based, event-driven, dynamic workload partition approach to explore fine-grain task parallelism and to maximize the throughput between resources. We evaluate our approach on different heterogeneous platforms using two *co-running* applications, Stencil and SpMV. In general, Stencil represents data regular computations, while SpMV stands as a good exemplar for data-irregular computations.

3.2 Hardware Resource Baseline, Limitations and Usage

As we discussed in Sect. 3.1, mathematical models can yield useful information for coarse-grain task scheduling. Our DAWL approach employs them to select suitable computing resources: pure CPU (*i.e.*, where only the host (CPUs) is contributing to the computation), pure GPU (*i.e.*, where only the accelerator is contributing to the overall computation: the host only handles data movement and in general I/O communications) or CPU–GPU co-running (*i.e.*, both the host and the accelerator(s) are contributing to the overall computation), to run different workload sizes. Coarse-grain workload partition means the workload is split into big chunks, such as big rows/columns chunks. It is totally different with the fine-grain workload partition (see Sects. 4.1.1 and 4.1.2).

3.2.1 Hardware Baseline Modeling

In this section, we present a baseline communication model, which is kept simple—no communication-computation overlap—on purpose. Equation 1 models the GPU execution time consisting of two types of costs: communication and computation.memcpy$_{H\rightarrow D}$ (resp. memcpy$_{DtoH}$) denotes communications from the host (resp. the accelerator) to the accelerator (resp. the host), to load the initial data (resp. to store the results back into the host), Compute$_D$ is the time required to process a given workload on the accelerator, and NumThreads$_D$ is the number of available processing elements.

$$GPU_{model} = \text{memcpy}_{H\rightarrow D} + \frac{\text{Compute}_D}{\text{NumThreads}_D} + \text{memcpy}_{D\rightarrow H} \tag{1}$$

Equation 2 models the CPUs execution time. Compute$_H$ and NumThreads$_H$ are the overall computation time on a general-purpose processing element on the host and the number of available processing elements on the host, respectively.

$$CPU_{model} = \frac{\text{Compute}_H}{\text{NumThreads}_H} \tag{2}$$

Equation 3 computes r, the ratio between GPU_{naive} and CPU_{naive} (these last two parameters are computed in Eqs. 1 and 2 respectively). r is a "hardware resource fitness" indicator of which part of the system should be favored. If $r \gg 1$, then the workload will execute much faster if it is on an accelerator. Hence, most if not all of the computation will be carried on the GPU. On the contrary, if $r \ll 1$, then the amount of data transfers is saturating the PCIe bus when running it on a GPU or the computing is not suitable for GPU processing, and in general, the overall computation is much faster using general-purpose processing elements. When $r \approx 1$, task scheduling must enable co-running, so that both the host and the accelerator are allocated their fair share of the work in order to complete the computation as fast as possible.

$$r = \frac{\text{CPU}_{model}}{\text{GPU}_{model}} \tag{3}$$

3.2.2 Limitations and Usage

Section 2.2 shows that, due to the various DMA engines available on modern GPUs, as well as the `Stream` technique in CUDA, it is possible to overlap communications and computations. Furthermore, it is quite hard to accurately estimate GPU computation times since the GPU utilization rate depends on factors associated with the GPU hardware and software architecture, such as the multi-level computing (thread) hierarchy, the GPU inner scheduler for tasks allocation on Streaming Multiprocessors (SMs), *etc*. Considering all the above factors, Eq. 1 is a worst-case view of a single GPU's performance. Conversely, it guarantees performance will be maximal if GPU_{naive} is "small enough" (see below).

Equation 2 is also rather naïve: while data transfers with the DRAM are not negligible, they take orders of magnitude less time than data transfers on a PCIe bus, which cannot be neglected. Moreover, HPC processors embed very efficient and aggressive data prefetching mechanisms, which tend to fully hide DRAM transfer latencies—especially in the case of consecutive reads or writes. However, the risk for cache conflicts in multicore systems (*e.g.*, false sharing) may cause significant drops in performance. Equation 2 is utilized to estimate the average performance of multi-threaded computing.

In DAWL, Eq. 3 is employed to estimate the initial workload on computing resources, CPUs and/or GPUs. At runtime, to allocate suitable workloads on the (different) computing resources, all three equations and real-time execution recording history, including the size of the workload and the corresponding execution time, *etc.*, should act in concert. The real-time recording history as an optimization factor can help increase the accuracy of the naïve mathematical model to some extent (more details can be found in Sect. 3.3.)

3.3 The Dynamic Adaptive WorkLoad (DAWL) Scheduler

The Dynamic Adaptive Work-Load (DAWL) Scheduler is an online scheduler where the workload distribution is based on a computation-communication model (Eqs. 1, 2, and 3) and runtime situation (the real-time execution recording history). It was created to decide what tasks should be scheduled and where to schedule the workload (*i.e.*, host or device) to minimize the load imbalance between

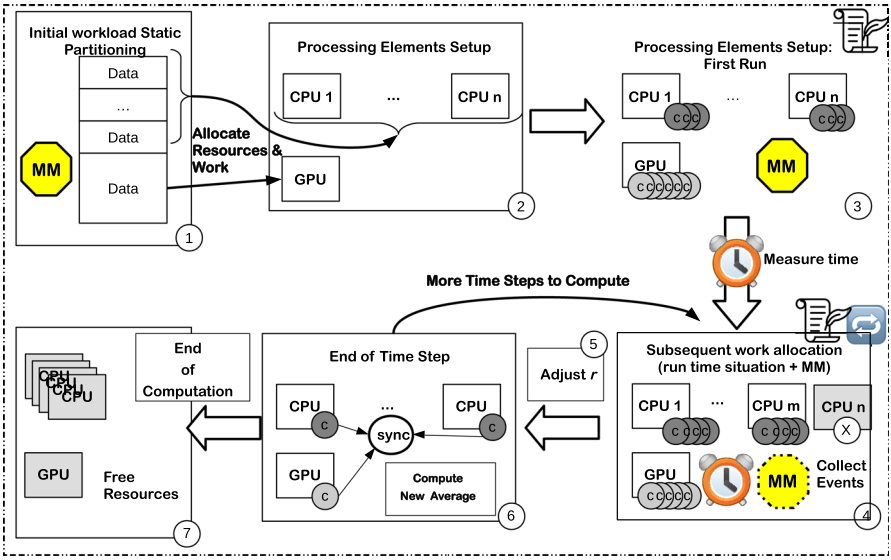


Fig. 2 The dynamic adaptive work-load scheduling algorithm (DAWL). Mathematical model (MM) occurs in DAWL's steps 1, 3, and 4. The dashed frame MM in step 4 stands for the optimized MM (Color figure online)

heterogeneous processing elements. It consists of seven main steps, illustrated in Fig. 2. We detail each step below.

1. Set up the initial workload on the Processing Elements (PEs), namely CPUs and/or GPU. PEs can be given different amounts of work based on their modeled "throughput" (see Eqs. 1–3).
2. Configure PEs based on step 1. This configuration includes the number of CPUs which will be put to work, whether the GPU will also be used, what portion of the available space in the shared memory (for the host) and global memory (for the accelerator) must be allocated, the number of streams on the GPU, *among other things*.
3. Simultaneously run tasks on both CPUs and GPUs, and time each run for their specific workloads. The current execution information will be recorded as follows:
   - *For CPUs* the number of running threads, the workload and corresponding execution time on the single thread, and total workloads and total execution time on all the threads.
   - *For GPUs* the amount of data transfer between host and device, the corresponding data transfer time, number of concurrent streams (if applicable), the number of thread blocks used and the corresponding computing time.[4]
4. Iteratively and adaptively adjust the workload based on the current run time situation and mathematical model; this entails several sub-steps:
   - *Update recording history* once the current allocated workload on the PE is finished, the current execution information will be updated into the recording history. If there are duplicates, the average execution time will be recorded.
   - Check the status of other PEs (running or waiting) as well as the corresponding recording history to estimate the completion time of other PE(s).
   - *Optimize model* The optimized mathematical model combines the original model (MM) with the history timing measurements. The new estimated execution time formula will be $\text{Time}_{new} = \alpha \cdot \text{Time}_{MM} + \beta \cdot \text{Time}_{his}$, where $\alpha + \beta = 1$ and the value of α will decrease with the number of iterations, while β is just the opposite as it will increase with the number of iterations. The other PEs status is also considered to allocate suitable sizes of workload on current PEs.
   - *Allocate workload on currently available PEs based on collected information and our optimized model* in addition to the size of the allocated workload, the number of threads (for CPU) and the number of concurrent streams (for the GPU) may also be adjusted.

[4] The information of concurrent stream and thread blocks are only for reference, as it cannot be efficiently utilized by the coarse-grain baseline model.

– *Repeat the whole procedure until the remaining workload is within* 10% *of the total workload* this 10% of the workload (remaining workload) is for the last step load balance optimization which is a fine-grain task scheduling approach and to guarantee there is no busy waiting at least for the last 10% of the total workload.

5. *Schedule the last* 10% *of the total workload* Calculate the value of *ratio*, where $ratio = \text{CPU}_{cur}/(\text{CPU}_{cur} + \text{GPU}_{cur})$. CPU_{cur} and GPU_{cur} are the amount of all work finished on CPUs and GPU, respectively. The corresponding GPU ratio is obtained using the same method. The CPUs or the GPU only take $\lfloor ratio \times$ remaining workload$\rfloor$ amount of work. The remaining workload is dynamically allocated to whichever (set of) PE(s) is available after early completion. Note: this is an application-based optimization.
6. Evaluate the load-balance metrics collected during the time step execution, in particular, the execution time. Adjust (coarsen) the task granularity based on available PEs and the metrics.
7. *Free all resources* PEs and memory.

3.4 Profile-Based Machine Learning Estimation Model

We have developed an optimized version of DAWL (PDAWL). PDAWL is a Profile-based Dynamic Adaptive Workload balance (PDAWL) which combines Machine Learning algorithms with runtime profiler information to solve the issues raised by the coarse-grain baseline mathematical model. The PDAWL framework consists of two components, illustrated in Fig. 3. On the left side of this figure, we can see our DAWL approach and the ML process on the right side.

Here, DAWL is responsible for online scheduling, while the ML model is in charge of providing performance estimation information. DAWL compares the baseline model with the performance estimation of the ML techniques. The ML component creates a performance prediction distribution for fine-grain workload using run time profiler information. Once the ML-based model is built, it is utilized in DAWL to replace and/or cooperate with the baseline model and then follow the DAWL online scheduling strategies (see Sect. 3.3.)

A known weakness of offline ML models is that they cannot be adjusted once the training process has completed [36]. With PDAWL, it is possible to compensate this weakness and provide guidance to an online scheduler even with changes in software or hardware. A combination of offline ML-based models with online schedulers is required when dealing with real-time constraints. If there are no real-time constraints, then online ML methods, *e.g.*, a stochastic gradient algorithm, may be used instead of offline methods.

As shown in Fig. 3, the heterogeneous ML model is built in two steps: first, CPU and GPU ML models are built separately; second, they are combined to predict how much workload to schedule in each type of PE(CPUs and GPU), building a heterogeneous ML model. We used `Oprofile` [24] and `Nvprof` [31] to collect runtine profile information from CPU and GPU executions. This data was used as

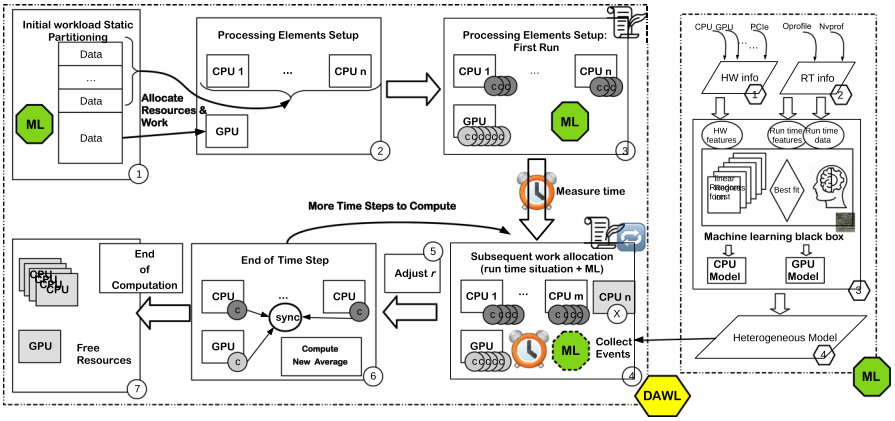


Fig. 3 PDAWL—the profile-based dynamic adaptive work-load scheduling algorithm: DAWL (see the bold hexagon DAWL, the left side) coupled with machine learning (see the bold octagon ML, the right side). Machine learning occurs in DAWL's steps 1, 3, and 4. The dashed frame ML in step 4 stands for the optimized ML (Color figure online)

input features for GPU and GPU ML models. Below, we describe in detail the steps to create the heterogeneous ML model:

1. Collect information about the hardware of the host and devices. Table 1 and 2 list some of the parameters involved. In addition to these, we also include cache hierarchy information, PCIe data transfer rates, and the GPU parameters; including the maximum number of concurrent streams, the GPU thread dimension information, the shared memory size, *among others.*
2. Collect runtime profile information from the application. The CPU and GPU ML models are used to predict the heterogeneous performance in a co-running mode.

    - *CPU* Since collecting all the events provided by `Oprofile` [24] is extremely time consuming, all the events are categorized into three groups: (1) cache related events including cache hierarchy and cache misses events; (2) branch

Table 1 CPU hardware features of the experimental platforms

Machines/Param.	Hardware environment				
	CPU parameters				
	Cores	Clock (GHz)	# Socket	L3 size (MB)	CPU mem (GB)
Machine1 (K20)	32	2.6	2	20	64
Machine2 (K20)	40	3	2	25	256
Machine3 (k40)	8	3.4	1	8	16
Machine4 (Titan)	12	3.4	1	12	31

Table 2 GPU hardware features of the experimental platforms with PCIe data transfer rate

Machines/Param.	Hardware environment				
	GPU parameters				PCIe (GB/s)
	# SM	Clock (GHz)	L2 size (MB)	GPU mem (GB)	
Machine1 (K20)	13	0.71	1.25	4.8	6.1
Machine2 (K20)	13	0.71	1.25	4.8	6.1
Machine3 (k40)	15	0.75	1.5	12	10.3
Machine4 (Titan)	14	0.88	1.5	6	11.5

related events; (3) all the other events. Based on the different applications, event groups 1 or 2 or all 3 can be activated. In this paper, we sample event groups 1 and 2.

- *GPU* `Nvprof` [31] provides many options to collect CUDA run time information. For our experiments we used two different categories. (1) `gpu-trace` and `api-trace` (faster, fewer events); (2) `nvprof-metrics` API (time consuming, more events). Using category 1 or 2 depends on the time constraints and accuracy requirements. To reduce the training time, we collect only category 1 of events.

3. *Normalize the collected data* since the collected data refers to many aspects such as cache miss rate, execution time, number of threads, ..., we need to change the numeric columns values to a standard scale without distorting the differences in the ranges of values.
4. *Clusters of features* since a high number of features are collected using `Oprofile` and `Nvprof`, a Hierarchical Agglomerative Clustering algorithm (HAC) is utilized to group correlation similarity features and finally obtain a reduced set of features. We tested sets of 4–12 features. First, a threshold is established with the correlation coefficients between the target variable (execution times) and the other features. Then, a dendrogram is built, using the correlation distance between the final features to clustering them by similarity.
5. We use the gathered information to build a profiler-based ML estimation model for CPU and GPU workloads. The CPU model focuses mainly on performance (execution time), resource utilization and cache issues; the GPU model mainly focuses on data transfers, computations time and the overlapping between them. Especially for GPU, unsuitable workload allocation and the number of concurrent streams will affect the host-device (CPU–GPU) communication-computation ratio to drop down the performance dramatically.
   The ML models utilizes the collected runtime profiler information to help the scheduler distribute fine-grain tasks and improve the total performance. More details can be found in Sect. 4, where we show how the fine-grain tasks are

allocated with the help of ML-based models using two applications as examples. Below, we describe how to create profile-based ML estimation models for CPU and GPU workloads:

- Run a set of ML methods such as linear regression, Support Vector Machine, and random forest model with the grouped features. Specifically, the linear regression model can be shown in two forms: y and $log_2(y)$. We use G to stand for the two forms: $G = y$ and $G = log_2(y)$). $G = \sum_{i=1}^{n} w_i \phi_i(x_i)$. Where $\phi_i(x)$ are functions from the set of x, x^2, x^3, x^4, e^x, $log_2 x$, $x \cdot log_2 x$, $ln\, x$, $x \cdot ln\, x$; x_i are features from last cluster step. Since y is our target variable, transformation is necessary for the logarithm version using 2^G. The reason why we include the Logarithm function is to reduce the non-linearity factors [4] and provide reasonable approximation with the target variable. For the SVM model, we use polynomial and Gaussian kernels.
- *Overfitting* we use 10-fold cross validation and L2 regularity to reduce the overfitting problems.
- *Models evaluation* to evaluate how well the model fits the data, a coefficient of determination, R^2, is used. It is defined as the percentage of the response variation that is explained by a linear model: $R^2 = \frac{\text{Explained variation}}{\text{Total variation}}$, with $0\% \leq R^2 \leq 100\%$. 0% indicates the model explains none of the variability of the response data around its mean. In contrast, 100% says that the model explains all the variability of the response data around its mean.
- *ML estimation model building* an estimation formula of the best matched statistical model can be built to predict an applications performance on this specific heterogeneous platform. The specific parameters used to construct the formula are mentioned in Sect. 4.3.

6. Build a heterogeneous prediction model based on the pure CPU and GPU model. The communication cost between CPUs and GPUs are included in the GPU model. To improve the GPU utilization effectiveness, especially when the workload memory footprint is much larger than the GPUs available global memory, CUDAs concurrent streams are used on GPU, based on Eqs. 1 and 3.

4 Algorithm Implementation and Experiment Results

This section starts by introducing our heterogeneous platform, then presents the two target applications, Stencil and SpMV, as well as their optimized co-running workload partition approaches, finally concludes with the performance analysis of these two applications employing different scheduling algorithms: `CPU-Seq`, `GPU-only`, `DARTS-CPU`, `DARTS-GPU`, `DARTS-DAWL`, `DARTS-Static` and `DARTS-PDAWL`. Specifically, `CPU-Seq` distributes the whole workload onto one single thread (CPU). It is used as the baseline. `GPU-only` distributes all the

workload on GPU using static fine-grain scheduling approach. If the whole workload is less than the available GPU memory, the single-stream approach is employed. If the whole workload is march larger than the available GPU memory, the concurrent stream approach is employed. `DARTS-CPU`,`DARTS-GPU`, `DARTS-DAWL`, `DARTS-static` and `DARTS-PDAWL` are implemented on `DARTS` runtime system, see Sect. 2.1. Specifically, `DARTS-CPU` distributes the whole workload onto CPUs (threads) using static coarse-grain scheduling approach that the whole workload is evenly partitioned and allocated on each thread. `DARTS-CPU` stands for homogeneous multi-thread computing. `DARTS-GPU` distributes all the workload onto GPU using static fine-grain scheduling approach. Different from `GPU-only`, `DARTS-GPU` employs concurrent stream approach all the time. `DARTS-DAWL` distributes the workload onto CPUs and/or GPU based on DAWL, see Sect. 3.3, which is a coarse-grain dynamic task scheduling approach. `DARTS-Static` also distributes workload onto CPUs and/or GPU, but it employs the coarse-grain static partition workloads approahch. `DARTS-PDAWL` distributes workload onto CPUs and/or GPU based on PDAWL, see Sect. 3.4, which is a profile-based fine-grain dynamic task scheduling approach. Aiming at different applications, the scheduling approaches mentioned above may vary. More details will be discussed in the corresponding experiments.

4.1 Experimental Testbed

DARTS already yields high performance on single-node homogeneous many-core systems [3, 18, 37]. As explained in Sect. 2.1, we modified DARTS to be *heterogeneous* and make it *GPU-aware,*. It is capable of scheduling `CPU_codelets` and `GPU_codelets` simultaneously. We ran the experiments on four heterogeneous systems, as shown in Tables 1 and 2. The software environment of these machine is shown in Table 3. Stencil-based computations and Sparse Matrix-Vector multiplication using the Compressed Row Format (SpMV-CSR) were selected to evaluate our DAWL and PDAWL.

4.1.1 Target Application: Stencil Computation

To emphasize a worst-case scenario, we used the Stencil kernels described in [18], without ghost cells, which enhances the need for synchronization. Specifically, we focused on two kernels: a 5-point 2D and a 7-point 3D Stencil, using double

Table 3 Software environment

Machines/Param.	Software environment	
	GCC	CUDA
Machine1 (K20)	v6.2/v8.1	v8.0
Machine2 (K20)	v4.8.5/v6.2	v8.0
Machine3 (k40)	v5.4	v9.0
Machine4 (Titan)	v4.9.2	v9.1

precision values. We fixed the number of time steps to 30, removing the convergence test at the end of each time step for simplification and making it more deterministic. Note that the CPU tasks and GPU tasks within one timestep were independent and that a global barrier was inserted at the end of each iteration. We repeated each experiment 20 times. There are no confidence intervals as the standard deviations were small, the larger one being 5% and the average smaller than 1%.

We follow two partitioning approaches: coarse-grain (DAWL) and fine-grain (PDAWL) are implemented for Stencil computation. As mentioned in Sect. 3.2, the naïve mathematical model is utilized to provide coarse-grain workload partitioning so the whole workload is split into large chunks of rows and/or columns, and then distributed to processing elements. The fine-grain partitioning approach refers to more parameters detailed below.

To implement the fine-grain task distribution between CPUs and GPUs, our approach consists of two steps: "Slicing" and "Tiling," respectively. "Slicing," including 2D and 3D-Slicing, means that the workload is partitioned along one dimension, as shown in Fig. 4. Within a slice, "Tiling" (*i.e.,* L1-Tile (L1 cache) for CPU tasks and Block-Tile for GPU tasks) can then be utilized. Figure 4 shows the 2D and 3D Stencil workload partitioning paradigm in the GPU/CPU co-running situation. This paradigm also works for the pure CPU/GPU cases by removing the GPU/CPU from the paradigm. In co-running situations, CPU and GPU "Slicing" may meet at some point.

Correctness and performance are the two main targets for our fine-grain task scheduling and distribution system. The workload allocation parameters should be carefully chosen to avoid computing errors and to avoid dramatically performance fluctuation/declining. In particular, the communication-computation ratio plays a pivotal role for GPU tasks. The parameters affect the ratio, including the number of concurrent streams, the workload (including transformation and computation) for single stream, the size of a block tile, the number of thread block within one block, the total number of thread block, the synchronization between streams, *etc.* The

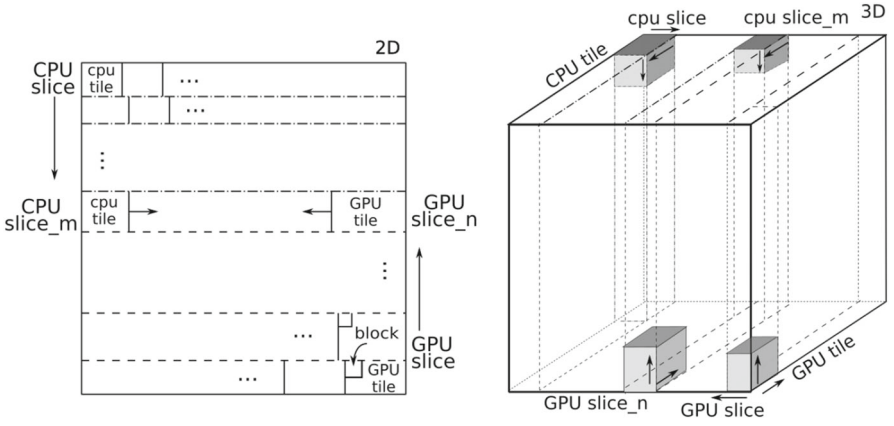


Fig. 4 GPU/CPU hybrid: stencil 2D/3D slicing and tiling

model utilized in DAWL is a coarse-grain model which is incapable to provide these fine-grain parameters. We instead employ an ML model in PDAWL to automatically obtain the correctly matched parameters which can result in a near-optimal compute-communication overlap and maximum both CPUs and GPU utilization. As mentioned in Sect. 3, different systems architectures can yield different parameters for our ML model.

Furthermore, the fine-grain task distribution helps us reduce the ML training time. Combining "Slicing," "Tiling," and the concurrent streams approach can help split the huge workload task into a set of small workload tasks. A small task, owning the feature of fewer data transformation, which is one of the most time consuming tasks, can converge to a near-optimal solution much faster. Since a very large workload can be seen as the combination of small workloads, the ML model trained by small tasks can be utilized to predict the performance of much larger ones on GPUs.

Algorithm 1: Pseudo-code: Stencil (2D/3D) co-running approach

```
1  Function Stencil_Main():
2      S0: init_system();
3      S1: parameters = Estimation_Func(); // MM/ML
4      for it = 0; it < total_Iteration; it++ do
5          S2: Stencil_CPU(parameters);
6          S2': Stencil_GPU(parameters);
7          S3: Sync_All_Resources();
8          S4: parameters = Obtain_best_parameters(history);
9      end
10 Function Stencil_CPE(*parameters):
11     S20: Sync_Remaining_WL();
12     do
13         S21: Run_CPE(parameters) ; //computing
14         S22: Update_Recording_History(parameters);
15         S23: ostatus = Check_OPE_Status(); //OPE
16         S24: Opt_Estimation_Func(history, ostatus, parameters); //MM/ML
17         S25: Sync_Remaining_WL();
18     while Remaining_WL > Total_WL * 10%
19     S26: Run_Remaining();
```

Algorithm 1 shows the Stencil pseudo-code co-running approach using DAWL or PDAWL. It consists of two functions: *Stencil_Main* and *Stencil_CPE*. CPE stands for the Current Processing Elements. OPE stands for the other Processing Elements. If the code is currently run on CPUs, then the CPE stands for CPUs and OPE stands for GPU, and vice-versa. We use one *Stencil_CPE* function to stand for two Stencil computations, respectively running on CPUs (*Stencil_CPU*, homogeneous multi-threads computing) and GPU (*Stencil_GPU*). DAWL and PDAWL share the same framework,[5] see Sect. 3.3, but employs a different performance estimation model corresponding to pseudo-code labels S1 (original model) and S24 (optimization model). The performance estimation model provides the necessary

[5] The pseudo-code mainly shows the DAWL components from item 3 to item 6.

parameters to the *Stencil_CPE* function. For DAWL, the parameter set is simple which includes the number of CPU threads, the workload (the Rows/Columns number) for each CPU threads, the number of GPU concurrent streams and the GPU workload in one round. For PDAWL, the parameter set is complicated which at least includes the number of CPU threads, the number of GPU concurrent streams, the "Slicing" and " Tiling" size for CPU thread (the number and size of slice/L1-Tiles) and GPU (the number and size of slice/block tiles, the number of thread block within one tile, the total number of thread block). Label S2 (*Stencil_CPU*) and S2'(*Stencil_GPU*) are run in parallel with (explicit and hidden) synchronization operations. Label S21 (*Run_CPE*) is the Stencil computation following the partitioning rules described in Fig. 4. The specific CPU multi-threads code can be found in paper [18]. The GPU code is the concurrent stream version of the Stencil-kernel code with the pipeline technique optimization. Label S22 (*Update_Recording_History*) records and updates the CPE (history) information (see DAWL item 3). Before starting the next computing task, it is necessary to check the OPE status (Label S23, *Check_OPE_Status*), to estimate when the OPE computing will be finished if the current status of OPE is running. With the history and the current OPE status information, we can leverage the optimized estimation model (Label S24,*Opt_Estimation_func*, DAWL item 4) to provide a new parameter set for the next computing task. Label S26 (*Run_Remaining*) corresponds to the DAWL item 5 and the parallel computation is also involved in this step. When all the computations within one iteration are finished, there is a an explicit synchronization operation (label S3, *Sync_All_Resources*). It is prepared for the evaluation operation, Label S4. Label S4 (*Obtain_best_parameters*) evaluates all the tasks running on different PEs during this iteration, and then figure the best matched parameters set for PES which can be utilized in the next iteration (corresponding to DAWL item 6).

4.1.2 Target Application: SpMV Computation

We used the SHOC benchmark suite's implementation of SpMV-CSR (Scalar version) [11] *blue*kernel functions, for both the CUDA and C++ sequential versions. We converted the sequential code to parallel code where every CPU's processing element (PE) can calculate one or multiple rows. One PE is in charge of communication and synchronization between the host (CPU) and the accelerator/device (GPU). For the CUDA code, we utilize the concurrent stream technique as an optimization. Just as with Stencil, SpMV has also been implemented in two versions: coarse-grain and fine-grain. The coarse-grain version is the parallel version of SHOC SpMV-CSR mentioned above. The fine-grain version is similar to Stencil, in that it involves fine-grain partitioning of the current/selected workload, the number of GPU concurrent streams, GPU Block-Tile size, *etc.* Furthermore, considering the features of the SpMV algorithm, the SpMV fine-grain approach entails one more step called pre-processing; more detail can be found below.

The performance profile of sparse matrix-based computations vary widely depending on the sparsity of its matrices' rows. A row is either "sparser" (*i.e.*, it contains more zeros than non-zero values) or "denser" (if it conversely contains more non-zero elements). The execution time of tasks on "sparser" and "denser"

rows may vary enormously. If all the rows are evenly allocated on computing resources (PEs), the execution time will depend on the heaviest task which is allocated with the "densest" rows. Since the target application is a sparse matrix, the majority of tasks are just waiting for the completion of the heaviest task. It reduces the computation resource utilization and results in lower performance. We propose to pre-process data at first: extract the "denser" rows as irregular computation tasks. At this point, the majority of sparser rows that are left over can be considered regular computing. Considering the features of CPUs and GPU, GPU will be preferred to run regular computing tasks, while CPUs can run both regular and irregular computing tasks. To split denser and sparser rows, we built up a *SpMV Co-running Model* on SHOC SpMV-CSR. More specific steps are shown below:

1. Analyze and evaluate statistical information, as shown in Table 4, to estimate the sparsity degree of the matrix. NNZ is the total number of non-zero elements. μ is the average number of non-zero elements per row. σ is the variance of the number of non-zero elements per row. CV stands for coefficient of variation per row. MAX: the maximum number of non-zero elements per row.
2. Build priority groups based on collected information (see Fig. 5). The highest priority level contains the maximum non-zero number per row(s); the lowest priority level contains the minimal non-zero number per row(s). On the same level, group members have similar non-zero numbers so they can run in parallel. To simplify the model, we statically set the ratio (30%) [19] as the threshold. The top 30% maximum non-zero number per row(s) will be extracted from the matrix and added to CPUs priority groups.
3. Run irregular and regular computations on CPUs and GPU, in parallel. CPUs will proceed from the highest to the lowest priority level, and GPU will proceed from the lowest priority level. Here, a concurrent stream approach is also utilized in the GPU.
4. Synchronize when all the CPUs and GPU computations are finished.

Matrices Used for Our Experiments We used 50 sparse matrices from the University of Florida Sparse Matrix Collection (UFSMC) [12] to train and 5 matrices (see Table 4) to evaluate our DAWL/PDAWL.

Table 4 Matrices for SpMV

Name	Dimension (M)	NNZ (M)	μ	σ	cv	MAX
circuit5M	5.56	59.52	10.71	1356.62	126.68	1,290,501
eu-2005	0.86	19.24	22.30	29.33	1.32	6985
in-2004	1.38	16.92	12.23	37.23	3.04	7753
FullChip	2.99	26.62	8.91	1806.80	202.73	2,312,481
kmer_U1a	67.7	138.8	2.05	0.37	0.18	35

NNZ: total of non-zero elements

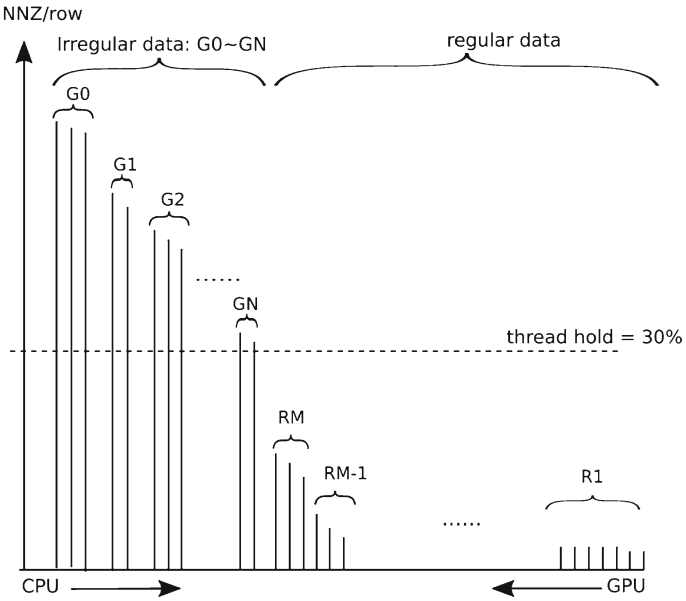


Fig. 5 SpMV: GPU/CPU priority groups

Algorithm 2: Pseudo-code: SpMV co-running approach

```
1  Function SpMV_Main():
2      S0: Spmv_Config_Info = Obtain_SpMV_Info();
3      S1: Priority_Group = Build_Priority_Group(Spmv_Config_Info); // pre-process
4      s2: parameters = Estimation_Func(Priority_Group); // MM/ML
5      S3: SpMV_CPU(parameters);
6      S3': SpMV_GPU(parameters);
7      S4: Sync_All_Resources();
8  Function SpMV_CPE(*parameters):
9      do
10         S31: Run_CPE(parameters) ;
11         S32: Update_Recording_History(parameters);
12         S33: ostatus = Check_OPE_Status(); //OPE
13         S34: Opt_Estimation_Func(history, ostatus, parameters); // MM/ML
14     while Remaining_Rows > 0
```

Algorithm 2 shows the SpMV pseudo-code co-running approach using DAWL or PDAWL. Just as with Stencil computing, it consists of two functions: *SpMV_Main* and *SpMV_CPE*. *SpMV_CPE* stands for two parallel running functions, *SpMV_CPU* and *SpMV_GPU*. Based on the features of SpMV, no iteration operation is involved and no last step fine grain optimization (see DAWL item 5) is utilized. Label S0 (*Obtain_SpMV_Info*) is to obtain the configuration information of the current sparse matrix and vector including the total number of rows and columns, the pointers to the CSR format SpMV matrices and vector, NNZ, MAX, *etc.* listed in Table 4. Label S1(*Build_Priority_Group*) is meant to build a priority group described in SpMV Co-Running Model item 2 and Fig. 5. This is the pre-processing step whose

purpose is to build two new matrices representing regular and irregular groups. The performance estimation model is based on these new matrices. Label S3 (*SpMV_CPU*) and S3'(*SpMV_GPU*) are run in parallel on CPUs and GPU with (explicit and hidden) synchronization operations. To avoid repetitions, we will not describe the detail of the other functions since they are similar to those in the Stencil pseudo-code.

4.1.3 Experiments Hardware Parameter Space Configuration

We used `numactl` to allocate memory in a round-robin fashion and avoid NUMA-related issues.[6] All systems were configured so that only 2 GB were seen as available by the runtime system, which has the effect of reducing the parameters space to explore. Figure 6 shows the same "drop-off" trend when using a 4 GB memory threshold which indicated that the artificial constraint we put on the GPU DRAM capacity does not impact the overall methodology nor its results. The initial workload is important for our workload distributed algorithm (DAWL) running in the co-running mode, as shown Fig. 6. The suffixes, "-1" and "-2," stand for different initial workloads. Even though the final speedup will converge when the total workload is large enough, during the whole process, especially in the first stage of co-running, an unsuitable initial workload will cause performance fluctuations.

We used two different mapping policies to pin DARTS threads to physical processing elements: `spread` and `compact` which roughly behave as OpenMP 4.5's `spread` and `close` thread configuration on the target device. The `spread` policy attempts to map DARTS threads to processing elements as far apart as possible physically on the underlying hardware. On the contrary, `compact` attempts to map DARTS threads as closely as possible on the available processing elements.

4.2 DAWL: Performance Analysis

To comprehensively characterize DAWL, we performed a series of workload performance analysis. We compared the `DARTS-DAWL` performance with `GPU-Only`, `CPU-Seq`, `DARTS-CPU`, and `DARTS-GPU` (see Table 5 for details).

Figure 7 shows the experimental results[7] that the Stencil kernels do not always scale well over multiple cores and nodes. Considering the Stencil features, such as data dependence , and the communication cost between CPUs/sockets, using more computing resources will not guarantee higher performance. The memory/cache conflicts and synchronization [18] issues incur quite a large overhead. Matched workload and computing resources is what is essential to obtain high performance.

[6] This ensures a stable DRAM access latency, and thus allows us to remove one parameter from the search space.

[7] The four machines onto which we experimented behave similarly in that respect. Hence we only show the machine 1 case.

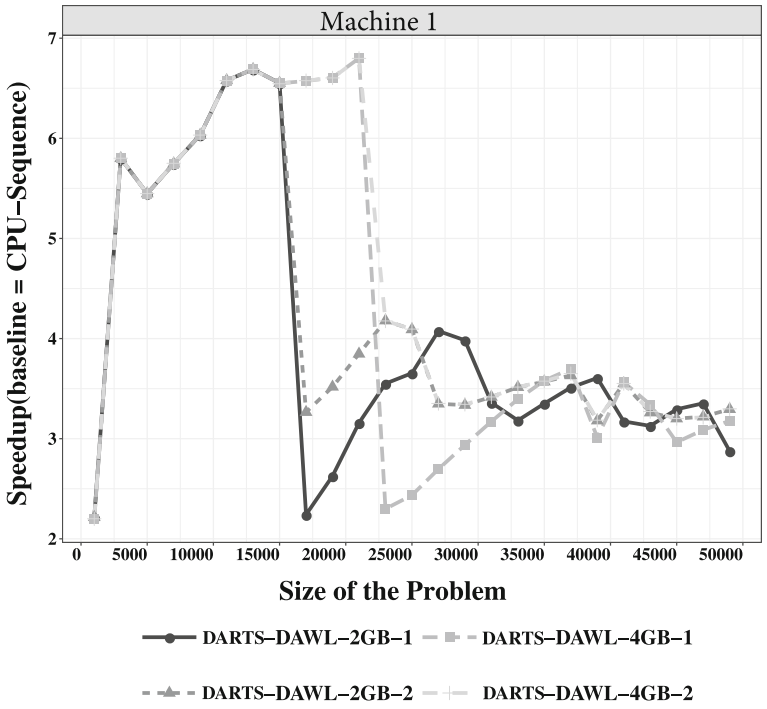


Fig. 6 Stencil 2D: speed up when GPU memory is 2 and 4 GB with different initial workload (GPU = CPU): the performance vary with initial workload

Table 5 Stencil kernel implementation

Implementation	Illustration
`CPU-Seq`	Sequential c++ code
`GPU-Only`	CUDA code
`DARTS-CPU`	Multi-threads c++ code
`DARTS-GPU`	CUDA code on DARTS (concurrent streams)
`DARTS-DAWL`	DAWL hybrid code on DARTS
`DARTS-Static`	DAWL hybrid static partition code on DARTS

Figure 8 shows the speedup of different variants for the 2D Stencil.[8] Here, `DARTS-GPU` uses concurrent streams at all times. Whereas, `GPU-Only` is slightly optimized comparing to the traditional way. When the problem size is smaller than the GPU memory capacity, we use the single-stream method to avoid superfluous synchronizations between the host and device. We use concurrent streams when the problem size is larger than the GPU's memory capacity to overlap communication

[8] The 2D and 3D cases behave similarly in that respect. Hence we only show the 2D case.

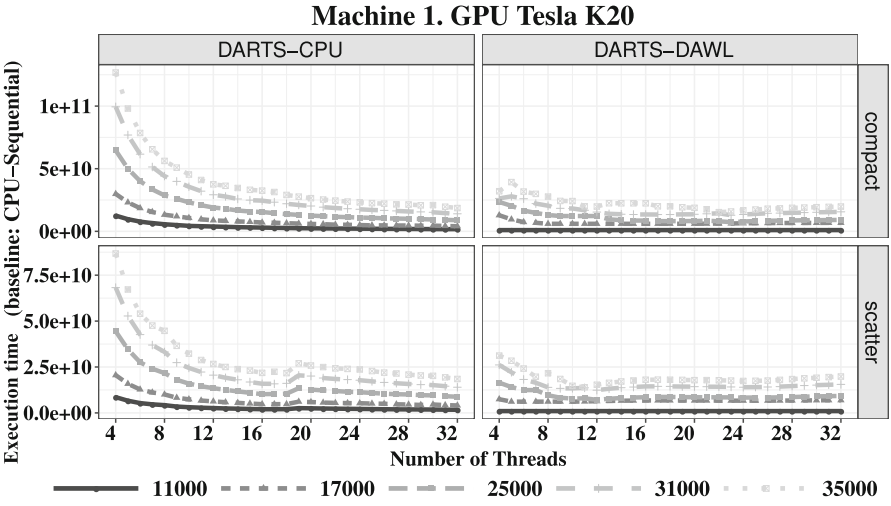


Fig. 7 Stencil 2D: performance with a varying number of HW threads. Time in nanoseconds

and computation. For `DARTS-DAWL`, we tried different initial workloads and chose the best set for CPUs and GPU.

Figure 8 verifies our baseline mathematical model. With 30 iterations constraints on Stencil kernels, and when the workload's memory footprint is smaller than the available device memory, $r \gg 1$ as described in Eq. 3, and the application allocates the full workload to the device to get maximum performance. When the memory footprint is bigger than the available device memory, it is allocated to both the host and the device. Considering the cost of communication and synchronization between these two resource types, the total performance ultimately drops. The speedup ratios are quite different on different systems, which is due to the differences in hardware. *e.g.,* the GPU of machine 3 is a Tesla-K40, which has a higher clock and memory frequency than Tesla-K20.

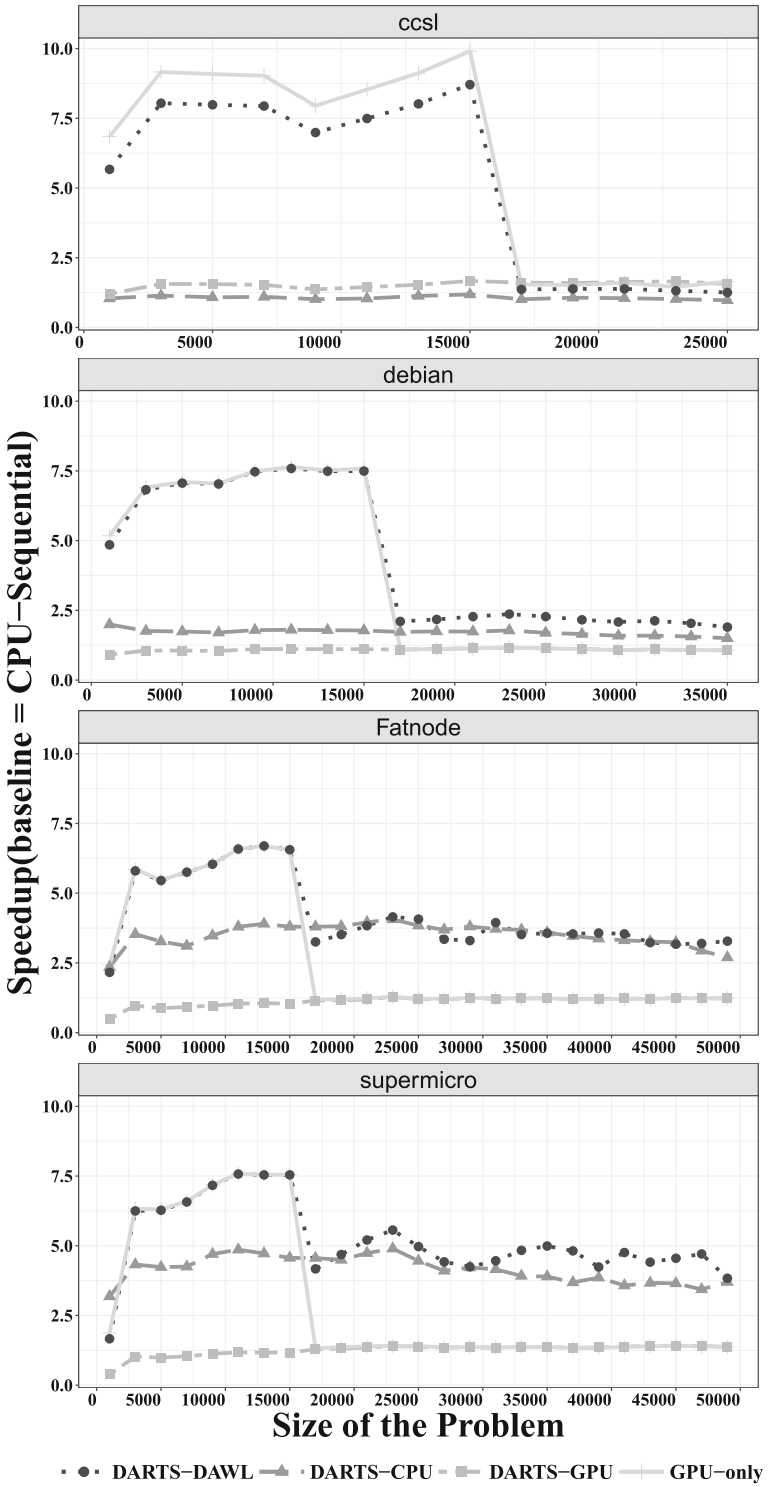
`DARTS-DAWL` on machine 3 should run in pure GPU mode based on Eqs. 1–3. Here, `DARTS-DAWL` is hard coded to use the co-running mode to prove that our ML approach can still improve performance even in the worst case.

4.3 Profile-Based Estimation Model, Analysis and Results

In this section, we first discuss why we employ the profile-based dynamic fine-grain workload partition approach instead of the simple static fine-grain workload partition approach for our experiments. Then, we analyze in detail the results of the experiments related to the Profile-based Estimation Model.

4.3.1 Dynamic Versus Static Workload Partition

As we discussed in Sect. 4.2, the coarse-grain tasks scheduling approach used in DAWL presents essential weaknesses. In Sects. 4.1.1 and 4.1.2, we list the benefits


ccsl
debian
Fatnode
supermicro
Speedup(baseline = CPU−Sequential)
Size of the Problem
DARTS−DAWL
DARTS−CPU
DARTS−GPU
GPU−only

◀**Fig. 8** Stencil 2D: speedup of the different versions

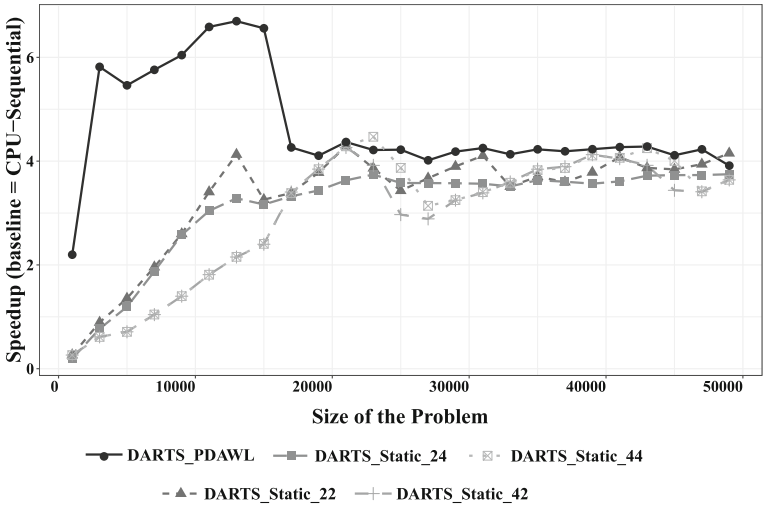


Fig. 9 Stencil 2D: dynamic and static workload partitions onto CPUs and GPU. 24 means CPU workload is 2000 × 2000, GPU workload is 4000 × 4000

of fine-grain task scheduling for the Stencil and SpMV applications. The next question will be why we train the profile-based ML model to estimate the fine-grain task performance and then dynamically and adaptively approach these tasks based on the runtime situation? Why not just utilize a simple static fine-grain tasks scheduling approach? Figure 9 provides the answers. It shows the dynamic and static workload partitions onto the CPUs and GPU. The static fine-grain approach can finally converge to an optimal state if the workload is large enough which means it takes a long time to converge. Furthermore, the performance fluctuates much during the whole process and it varies with different partitions size. On the contrary, PDAWL , which employs a profile-based ML model and can dynamically adjust the workload based on the runtime situation, always reaches the optimal performance.

4.3.2 Profile-Based Estimation Model

Section 3.4 shows how we used the performance of pure CPU/GPU versions to predict co-running executions. Our training/validation/test sets are split between CPU and GPU.

The "CPU set" is to build a CPU performance-resource estimation model which can provide the "best" scheduler using minimum computing resources to obtain the maximum performance (shortest execution time) for a specific workload. As described in Sect. 4.1.3, combining `spread` and `compact` mapping policies, we

run experiments with different active CPU threads number (*e.g.* 2, 4, 8, 16...) to obtain the necessary run time information by using Oprofile [24]. Our experiments (Fig. 7) show that when the CPU threads number reaches a given threshold, increasing the number of threads does not improve performance—which is to be expected because of memory conflicts. Furthermore, PDAWL utilizes this information to provide an accurate prediction model even when *e.g.*, some PEs are suddenly turned off because of power issues.

The "GPU set" is used to build a GPU communication-computation overlap model, to estimate data transfer and execution time. In particular, the right Block-Tile size, the number of thread block, and the number of concurrent streams can perfectly overlap communication and computation on a system; and yet, the overlap ratio may be very low on other systems since the available SM, PCIe throughput, *etc.*, are totally different. This is particularly true if the Block-Tile size is too small to cover the CUDA runtime API launching time: there will be no overlap between computation and communication. Too many concurrent streams will increase the pressure on the scheduler, the pre/post-processing time and even increase the amount of data transfers. `nvprof` [31] can be employed to obtain related information. Specifically, the estimation model consists of: API launching, events, metrics, data transfer between host and device and device computation parts. We run the two versions of the GPU code, with/without a different number of concurrent streams, combining with different Block-Tile size.

The information collected by the runtime system helps gather more than two hundreds features for each type of device. Features are computation and communication metrics and events. To obtain a reasonable group of features, a correlation analysis and cluster algorithm, Hierarchical Agglomerative Clustering algorithm (HAC), was employed. For instance, Fig. 10 shows one dendrogram describing a matrix correlation of different GPU parameters and a grouping of the set of features. rows and columns share the same variables. Each cell in the table shows the correlation between two variables. In this figure, blue and red are minimum and maximum correlations, respectively. First, features with a high correlation with the execution time are selected. Second, selected features are analyzed with a hierarchical clustering and different sets of features are created.

Figure 11 shows 5 clusters. If the height of the threshold is increased (see red dash line), then fewer clusters are created. This means that the threshold (the red dashed line) in the dendrogram determines the number of clusters.

From the other side, these grouped features obtained from the Hierarchical Agglomerative Clustering algorithm (HAC) point to the most important aspects which affect the performance of applications on the heterogeneous system. For example, if the feature is related to the threads number (CPU) or global_store_-transactions (GPU), then the application will be more sensitive to computation (CPU) or the communication between host and GPU (GPU). Based on this information, the grouped features can provide optimization suggestions for current application optimization.

After the data is collected, various ML algorithms see Sect. 3.4, PDAWL item 5 are run and the one that fits the model best is selected.

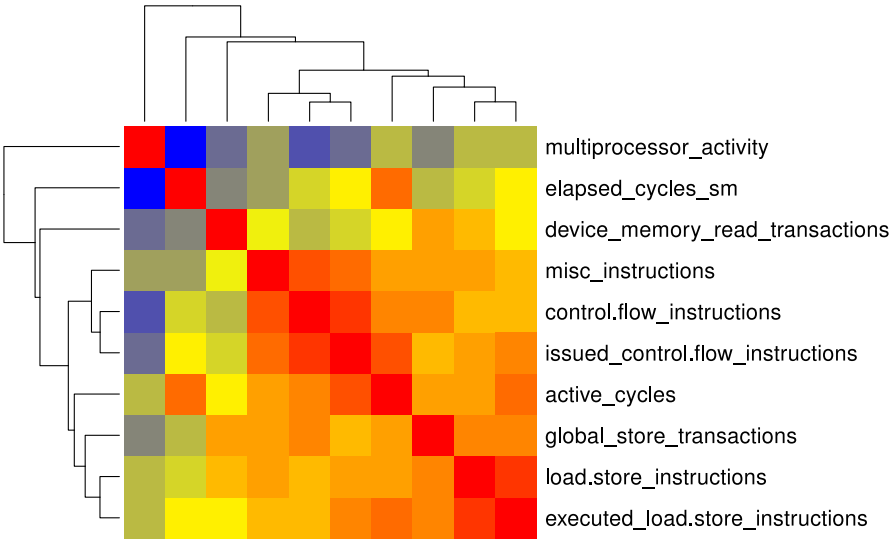


Fig. 10 Built dendrogram from a matrix correlation of the set of GPU features coming from nvprof (Color figure online)

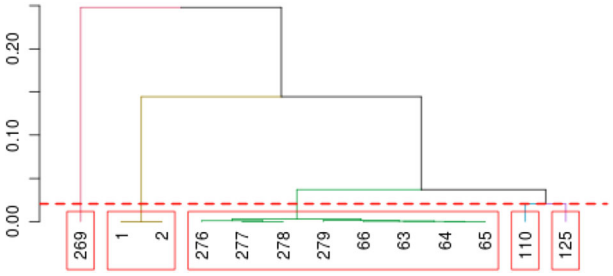


Fig. 11 Dendrogram generated from a set of CPU features coming from `OProfile`. The numbers on the x axis represent the features assigned by `OProfile`. The numbers on the y axis represent thresholds (Color figure online)

Here, we use two simple linear regression functions as an example[9] to explain how ML algorithms were chosen (PDAWL item 5). If five features (x_1 to x_5) are selected by the HAC clustering algorithm (PDAWL item 4), the two possible linear regression functions (which are randomly picked from our linear regression model set) can be: 1. $G = w_1 * (x_1) + w_2 * (x_1)^2 + w_3 * (x_2) + w_4 * (x_2)^2 + w_5 * (x_3) + w_6 * (x_3)^2 + \quad w_7 * (x_4) + w_8 * (x_4)^2 + w_9 * log_2 x_5 + w_{10} * (x_5) * log_2 x_5$; 2. $G = w_1 * (x_1) + w_2 * (x_1)^2 + w_3 * (x_2) + \quad w_4 * (x_2)^2 + w_5 * (x_3) + w_6 * (x_3)^2 + w_7 * (x_4) + w_8 * (x_4)^2 + w_9 * x_5 + w_{10} * log_2 x_5$; Then, traditional ML training and

[9] Here we randomly pick two from the whole linear regression model set as an example. May neither one be the best fit function in the experiment.

validation methods, such as the use of 10-fold cross validation and L2 regulation to avoid overfitting, can be utilized on these two linear regression models to obtain the optimum weights (w_i). When the weights for each function are obtained, we transfer G to y, since y is our target function, as we described in PDAWL item 5. The two forms are $G = y$ and $G = log_2(y)$. Then the transformation will be $y = G$ and $y = 2^G$. The reason why we include the logarithm function in our model is that logarithmic scale function can reduce the non-linearity factors and provide reasonable approximations [4]. R^2 is utilized to evaluate which is the best fit function and will be selected as our finally performance estimation model. Beside the accuracy, the computation complexity of the model is considered when chosen as the best fit function. For example, if one of the linear model and SVM model have similar R^2, such that the difference is less than 0.01%, then the linear model will be chosen at the end since the computation complexity of SVM is much higher than that of the linear model.

In our experiments, when training/validation all of our linear models, Random Forest and Support Vector Machine (SVM) models on four machines, we find that the majority of the best matches both for Stencil and SpMV computation is given by *linear regression* , and that its R^2 is $0.93 \leq R^2 \leq 0.94$. Linear regression is also highly efficient for training and testing evaluations. In the future, more ML models and approaches will be added to our experiments.

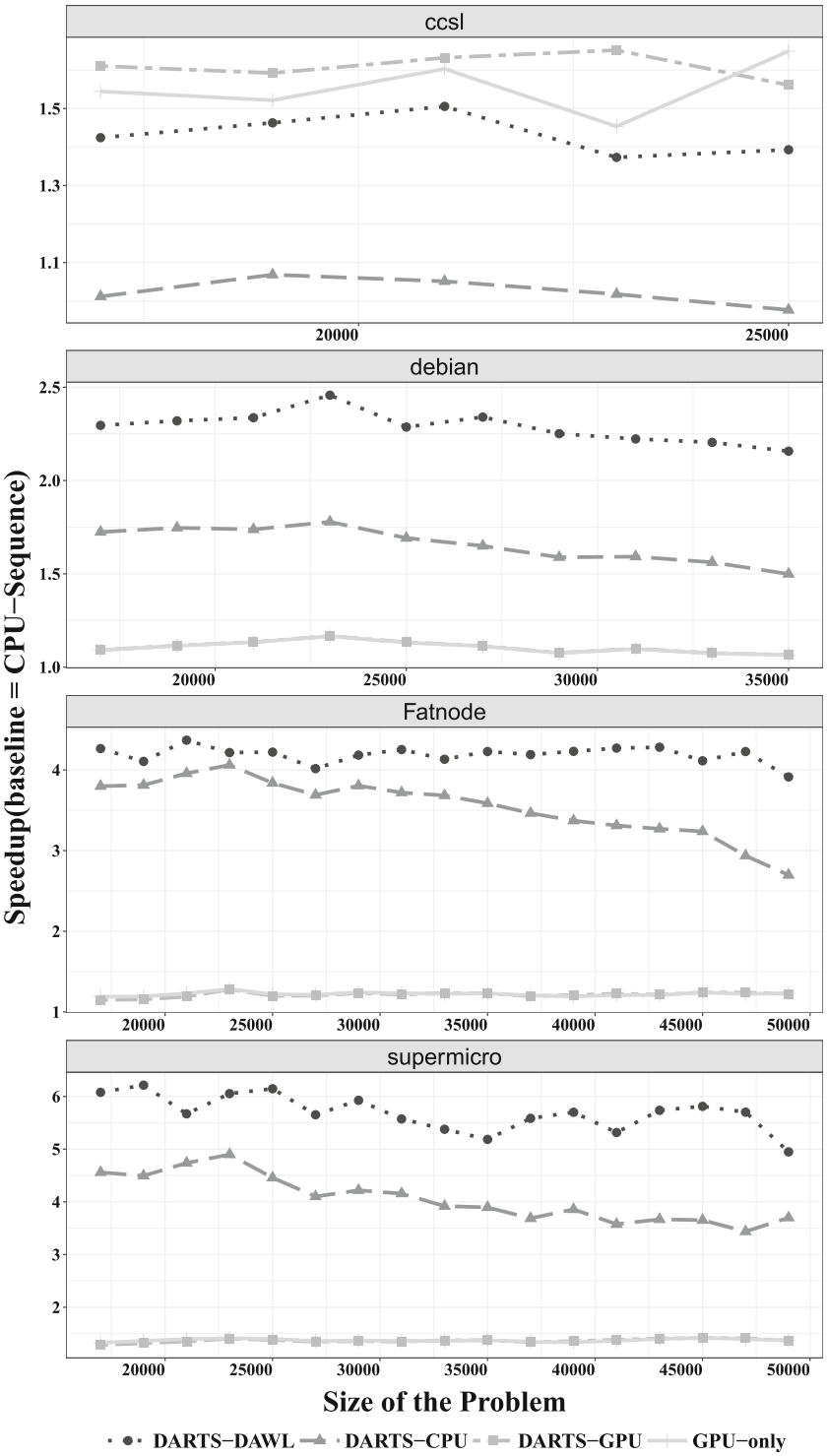
The Mean Absolute Percentage Error (MAPE) is utilized to measure the accuracy of our prediction model. Table 6 shows the MAPE of the linear model for each machine in the Stencil 3D experiments.

Figures 12 and 13 show the results for PDAWL. Compared to `DARTS-CPU`, the number of PEs changes with runtime. Our scheduler can reach up to 6× speedups compared to sequential runs, 1.6× speedup compared to the multiple core version, and 4.8× speedup compared to the pure GPU version in the 2D Stencil. In the 3D Stencil, `DARTS-PDAWL` reaches speedups up to 9× compared to the sequential version, 1.8× against multi-cores, and 3.6× against a pure GPU version. Figures 12 and 13 that profiling does not always yield significant speedups. This is especially true around *drop points*, *i.e.*, unstable points which are affected by multiple co-running hardware/software conflicts parameters, which our machine learning estimation model did not take into consideration.

Figure 14 compares the SpMV of the five Matrices listed in Table 4 on Machine 1. *DARTS-CPU* (pure CPU) employs the coarse-grain task scheduling approach which evenly distributes all rows onto multheeads (CPUs). *DARTS-GPU* (pure GPU) employs the fine-grain task scheduling approach which evenly distributes all rows onto thread blocks (GPUs). Considering the features of sparse matrices, the

Table 6 Stencil 3D: mean absolute percentage error of the performance prediction with linear regression models for each machine

Machines	#1	#2	#3	#4
MAPE	6.43%	7.41%	3.45%	1.68%


ccsl
debian
Fatnode
supermicro
Speedup(baseline = CPU−Sequence)
Size of the Problem
DARTS−DAWL
DARTS−CPU
DARTS−GPU
GPU−only

◀**Fig. 12** Stencil 2D: speedup when matrices are larger than 17K × 17K (PDAWL)

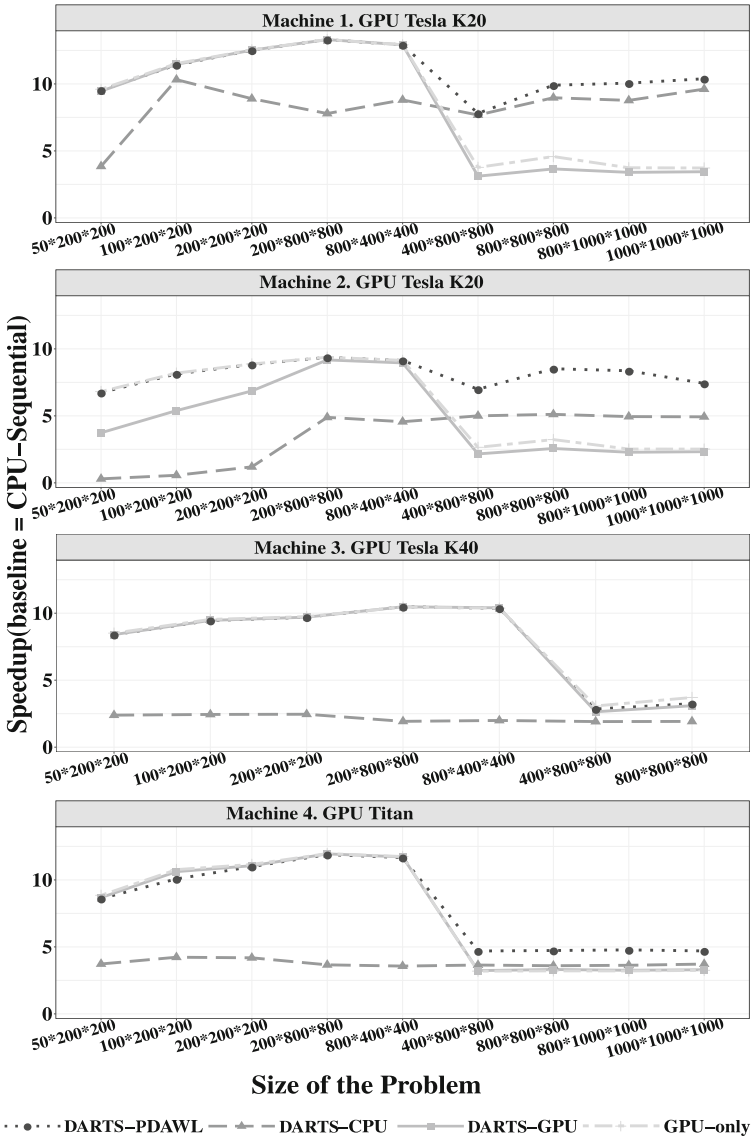


Fig. 13 Stencil 3D: speedup (PDAWL)

non-zeros number per rows varies enormously. If the workload is split in rows, then the execution time of each partition will varies greatly. The totally execution time depends on the partition with the maximum total of non-zeros elements. Resource under utilization issues exist in both *DARTS-CPU* and *DARTS-GPU*. Compared to

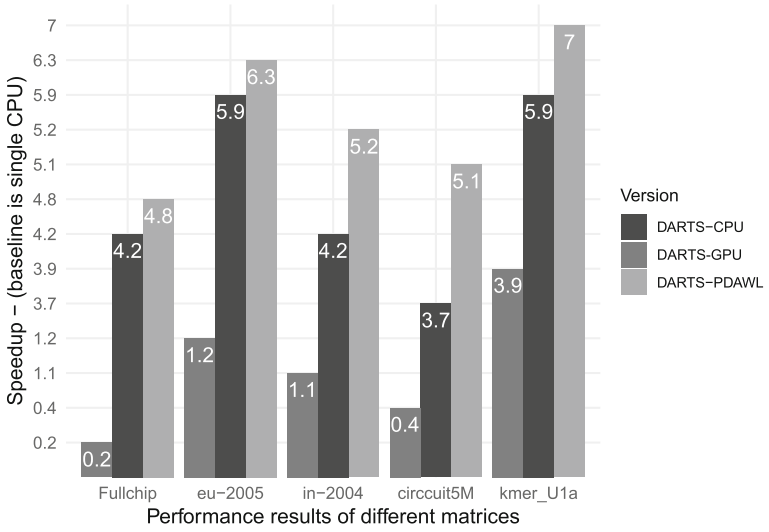


Fig. 14 SpMV performance(SpeedUP)

DARTS-GPU, the performance of *DARTS-CPU* is much higher than that for the *DARTS-GPU*. This is because CPUs are better at processing complex data structure which is the feature SpMV-CSR format. *DARTS-DAWL* is not shown in the figure since it overlaps with *DARTS-CPU* with very tiny differences which are not large enough to be visible. So, *DARTS-CPU* also stands for *DARTS-DAWL* in this figure. *DARTS-PDAWL* first transforms sparse matrices into dense matrices using the approach described in Fig. 5, and then utilize the ML model to find the best matched parameter set for both CPU and GPU to improve computing resource utilization. `DARTS-PDAWL` executes up to 30.5× faster than the GPU version and 1.37× faster than the multi CPU version. The speedup depends on the degree of sparsity in the tested matrices. As shown in Fig. 14, the speedup of `DARTS-PDAWL`/`DARTS-GPU` of Fullchip and circuit5M are far larger than the others. Furthermore, our optimized SpMV approach, mentioned in Sect. 4.1.2, can run in parallel a regular computation group on GPU and an irregular one, and perhaps also include part of a regular, computation group on the CPU. 30% of threshold is a reasonable value [19] for an SpMV computation. Choosing a more suitable threshold, using ML algorithms, in order to further to improve the performance of `DARTS-PDAWL` for sparse matrices computing will be one of our future tasks.

To summarize, based on the experiment results analysis (both Stencil and SpMV), PDAWL can adaptively schedule regular and irregular workloads based on the system hardware architecture. The reasons why PDAWL outperforms pure GPU, pure CPU and DAWL are that it can fully utilize the available computing resources, can find the suitable synchronization way to guarantee that all the resources are co-running all the time, and can run on different architectures without considering the differences of hardware architecture, initial workload allocation and workload update ratio. Furthermore, PDAWL can also obtain relatively high

performance when tasks are forced to a (not best matched) computing resources (see machine 3 in Figs. 8 and 12). Even though, when facing the totally new applications and hardware environments, it may take times to re-training profile-based ML estimation model, where the re-training time depends on the complexity of system architectures and applications, PDAWL can still be used as a general approach for co-running applications, such as linear algebra applications, to obtain a relatively better performance during the test for different systems. Furthermore, since we have collected the important features based on the HAC algorithm, if the changes of hardware do not effect the important features, there is no need to re-training for the same/similar applications.

5 Related Works

The main challenge of the load-balancing mechanism is to divide the workload into processing units precisely. A simple heuristics division approach may result in worse performance than a simple uniform division. Machine-learning-based prediction mechanism or/and online profiling-based scheduling algorithms have been deployed to determine the workload partitioning decision on many-core homogeneous/heterogeneous systems.

Luk *et al.* [26] proposes an empirical adaptive mapping, a fully automatic technique to map computations to processing elements on heterogeneous multiprocessors. Wang et al [42] utilizes an ML approach to decide whether to parallelize a loop and how to schedule candidates on multi-core platforms. Memeti and Pllana [30] combined optimization and machine learning to statically distribute work between the host and device of heterogeneous computing systems to minimize the overall application execution time. Belviranli [5] performs a dynamic load-balancing algorithm (Heterogeneous Dynamic Self-Scheduler-HDSS) for heterogeneous GPU clusters. Teodoro [38] performs a performance variation-aware scheduling technique along with an estimation optimization model to collaboratively use CPUs and GPUs on parallel systems. Sant'Ana *et al.* [34, 35] implement two profile-based load-balancing algorithms named PLB-Hec and PLB-HAC for data-parallel applications in heterogeneous CPU–GPU clusters. The ML approach is utilized to predict the best distribution of data block size among different processing units. Zhang *et al.* [46] performs a series of workload characterization analysis to understand the co-running behaviors on integrated CPU/GPU architecture. The main factors affecting the co-running performance: the architectural differences between CPUs and GPUs and the limited shared memory bandwidth. Based on this information, an ML model can be built to predict coarse-grain workload partitioning on a co-running device before porting the program. Zhang *et al.* [45] proposes a fine-grain workload reshaping approach which combines performance prediction, from an ML model, and partitioning threshold, from an online-turning model, to partition the workload between CPU and GPU on integrated architectures. When the workload is lower than the threshold, it will be executed on GPUs; otherwise, CPUs will be employed. Margiolas *et al.* [28] and Boyle *et al.* [32] focus on the accelerator sharing control for multiple kernels and propose to use ML to determine

whether to run OpenCL code on GPU or OpenMP code on multi-core CPUs. Wen *et al.* [43] use ML to decide whether to merge or to separate multi-user OpenCL tasks running the most suitable devices in CPU–GPU systems.

To avoid extensive offline ML training, Laleem *et al.* [20] presents an adaptive online profiling based scheduling technique. Cho *et al.* [8] reshapes the workload on CPU/GPU based on online profile information generated at runtime. Zhang *et al.* As with [45], a threshold is employed by Cho *et al.* [8].

Except for architectural differences, communication between CPUs, GPUs, and the memory has a pivotal role. Chen *et al.* [7], Zhang *et al.* [46], Yang et al [44]. Van Craeynest *et al.* [40] and Garcia *et al.* [16] propose an analytical performance model that includes PCIe transfers and overlapping computation and communication. Lutz *et al.* [27] proposes PARTANS, an autotuning framework for CPUs and GPUs to execute Stencil computations over two nodes with multiple GPUs. Data transfer on the PCIe bus plays a crucial role in determining the number of GPUs to be utilized. To handle the communication-synchronization problem between CPUs and GPUs, Lee *et al.* [22] proposes SKMD (Single Kernel Multiple devices) to transparently orchestrate single kernel execution across asymmetric heterogeneous devices regardless of memory access patterns.

Most of these are aimed at static, coarse-grain workload distribution, and loosely synchronized parallel workloads where specific tasks are often run only a specific type of processing element (*e.g.*, CPU or GPU). Zhang *et al.* [45] works for fine-grain partitioning, but employs an inherently rigid static workload partition. Furthermore, the precision of the ML model determines the efficiency of the workload partitioning approach. The hardware change during runtime may have a catastrophic effect on the performance. At the same time, hardware changes during runtime may happen frequently, and as much as half of the CPU cores may be turned off because of power issues.

Our work focuses on dynamic, fine-grain workload distribution and tight synchronization between CPUs and GPUs. To adapt to the real time hardware situation, fully utilize the available computing resources and reduce the communication cost between CPUs and GPUs, we combine online scheduling and offline machine learning.

6 Conclusions and Future Work

We have presented a profile-based AI-assisted dynamic scheduling approach for heterogeneous architectures. To fully utilize the computing resources and improve performance in the processing of scientific applications, we have focused on the workload balance aspect. PDAWL, an iterative event-driven scheduling algorithm has been designed to load balance better tasks in a heterogeneous system. It leverages a naïve hardware resources mathematical model, combining offline profile-based machine learning and an online scheduling approach. Our model determines how the workload should be allocated to the heterogeneous computing resources. The profile-based Machine-Learning estimation model can help build an estimation model in a heterogeneous resource context. It consists of a CPU model

and a GPU model. We used ML to find the best workload-resource match to improve the CPUs utilization rate, and the optimal estimation model to improve GPU performance since building an accurate mathematical general-purpose GPU performance model is nigh-impossible, as the search space is too large. Furthermore, the cluster algorithm (HAC) within the ML model can provide optimization suggestions of the current application to improve the application performance on heterogeneous systems. An online event-driven scheduling can make up for the inflexibility of offline machine learning and increase accuracy of scheduling. Our approach is suitable for a very dynamic hardware environments where the computing resources can be turned off/on during run-time. Furthermore, our approach can be used in the presence of huge workloads that exceed that capacity of available device memory. The advantage of our approach is that the total machine learning training time will not increase much since we train with small workloads to predict the performance with very large workloads.

Two applications, Stencil and SpMV, have been chosen to evaluate our approach. Experiments with Stencil 2D, Stencil 3D, and SpMV show that PDAWL yields speedups up to 1.6×, 1.8×, and 1.37× for a multi-core baseline, 4.8×, 3.6×, and 30.5× for pure GPU execution.

Future work includes augmenting our model with power consumption parameters to enrich PDAWL and determining the right trade-offs between performance and power on heterogeneous architectures. Online learning algorithms, deep neural networks and other Machine Learning algorithms will be integrated into PDAWL. We will also employ meta-learning to reduce the training time when running our PDAWL on other hardware environment configurations.

Acknowledgements This work was partially supported by the National Science Foundation, under award CCF-1763793 and by FAPESP (São Paulo Research Foundation, Grant #2012/23300-7).

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the plasma and magma projects. J. Phys. Conf. Ser. **180**, 012037 (2009)
2. Amaris, M., Cordeiro, D., Goldman, A., De Camargo, R.Y.: A simple BSP-based model to predict execution time in GPU applications. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 285–294 (2015). https://doi.org/10.1109/HiPC.2015.34
3. Arteaga, J., Zuckerman, S., Gao, G.R.: Generating fine-grain multithreaded applications using a multigrain approach. ACM Trans. Archit. Code Optim. **14**(4), 47:1-47:26 (2017). https://doi.org/10.1145/3155288
4. Barnes, B.J., Rountree, B., Lowenthal, D.K., Reeves, J., de Supinski, B., Schulz, M.: A regression-based approach to scalability prediction. In: Proceedings of the 22Nd Annual International Conference on Supercomputing, ICS '08, pp. 368–377. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1375527.1375580
5. Belviranli, M.E., Bhuyan, L.N., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim. **9**(4), 57:1-57:20 (2013). https://doi.org/10.1145/2400682.2400716
6. Chen, J., Choudhary, A., Feldman, S., Hendrickson, B., Johnson, C., Mount, R., Sarkar, V., White, V., Williams, D.: Synergistic Challenges in Data-Intensive Science and Exascale Computing: DOE ASCAC Data Subcommittee Report. Department of Energy Office of Science (2013). Type: Report

7. Chen, Q., Guo, M.: Contention and locality-aware work-stealing for iterative applications in multi-socket computers. IEEE Trans. Comput. **67**(6), 784–798 (2018). https://doi.org/10.1109/TC.2017.2783932
8. Cho, Y., Negele, F., Park, S., Egger, B., Gross, T.R.: On-the-fly workload partitioning for integrated cpu/gpu architectures. In: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18, pp. 21:1–21:13. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243176.3243210
9. Chow, E., Anzt, H., Scott, J., Dongarra, J.: Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. J. Parallel Distrib. Comput. **119**, 219–230 (2018)
10. Cole, S.V., Buhler, J.: Mercator: a GPGPU framework for irregular streaming applications. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 727–736 (2017)
11. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3, pp. 63–74. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1735688.1735702
12. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1-1:25 (2011). https://doi.org/10.1145/2049662.2049663
13. De Raedt, H., Jin, F., Willsch, D., Willsch, M., Yoshioka, N., Ito, N., Yuan, S., Michielsen, K.: Massively parallel quantum computer simulator, eleven years later. Comput. Phys. Commun. **237**, 47–61 (2019)
14. Dehne, F., Hutchinson, D., Maheshwari, A., Dittrich, W.: Reducing I/O complexity by simulating coarse grained parallel algorithms. In: Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings, pp. 14–20 (1999). https://doi.org/10.1109/IPPS.1999.760428
15. Franchetti, F., Low, T.M., Popovici, D.T., Veras, R.M., Spampinato, D.G., Johnson, J.R., Püschel, M., Hoe, J.C., Moura, J.M.F.: Spiral: extreme performance portability. Proc. IEEE **106**(11), 1935–1968 (2018)
16. García, V., Gomez-Luna, J., Grass, T., Rico, A., Ayguade, E., Pena, A.J.: 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), pp. 1–10 (2016). https://doi.org/10.1109/IISWC.2016.7581277
17. Gaster, B.R., Howes, L.: Can GPGPU programming be liberated from the data-parallel bottleneck? Computer **45**, 42–52 (2012). https://doi.org/10.1109/MC.2012.257
18. Geng, T., Zuckerman, S., Monsalve, J., Goldman, A., Habib, S., Gaudiot, J.L., Gao, G.R.: The importance of efficient fine-grain synchronization for many-core systems. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 203–217. Springer (2016)
19. Guo, P., Wang, L., Chen, P.: A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. IEEE Trans. Parallel Distrib. Syst. **25**(5), 1112–1123 (2014). https://doi.org/10.1109/TPDS.2013.123
20. Kaleem, R., Barik, R., Shpeisman, T., Hu, C., Lewis, B.T., Pingali, K.: Adaptive heterogeneous scheduling for integrated GPUs. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 151–162 (2014). https://doi.org/10.1145/2628071.2628088
21. Leandro Nesi, L., da Silva Serpa, M., Mello Schnorr, L., Navaux, P.O.A.: Task-based parallel strategies for computational fluid dynamic application in heterogeneous CPU/GPU resources. Concurr. Comput. Pract. Exp. **32**(20), e5772 (2020). https://doi.org/10.1002/cpe.5772
22. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU–GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, pp. 245–256. IEEE Press, Piscataway, NJ, USA (2013)
23. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pp. 451–460. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1815961.1816021
24. Levon, J., Elie, P., Johnson M.: Oprofile: a system profiler for Linux (2004). https://oprofile.sourceforge.io/about/. Accessed 20 June 2020
25. List, T.S.: http://www.top500.org (2017)

26. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1669112.1669121
27. Lutz, T., Fensch, C., Cole, M.: Partans: an autotuning framework for stencil computation on multi-GPU systems. ACM Trans. Archit. Code Optim. (TACO) **9**(4), 59 (2013)
28. Margiolas, C., O'Boyle, M.F.P.: Portable and transparent software managed scheduling on accelerators for fair resource sharing. In: 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 82–93 (2016)
29. Martinasso, M., Kwasniewski, G., Alam, S.R., Schulthess, T.C., Hoefler, T.: 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, pp. 63:1–63:11. IEEE Press, Piscataway, NJ, USA (2016). http://dl.acm.org/citation.cfm?id=3014904.3014989
30. Memeti, S., Pllana, S.: Combinatorial optimization of work distribution on heterogeneous systems. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pp. 151–160 (2016)
31. NVIDIA: CUDA C: Programming Guide, Version 10.0. (2019)
32. O'Boyle, M.F.P., Wang, Z., Grewe, D.: Portable mapping of data parallel programs to opencl for heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO '13, pp. 1–10. IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/CGO.2013.6494993
33. Reed, D.A., Dongarra, J.: Exascale computing and big data. Commun. ACM **58**(7), 56–68 (2015). https://doi.org/10.1145/2699414
34. Sant'Ana, L., Cordeiro, D., Camargo, R.: PLB-HeC: a profile-based load-balancing algorithm for heterogeneous CPU–GPU clusters. In: 2015 IEEE International Conference on Cluster Computing, pp. 96–105 (2015). https://doi.org/10.1109/CLUSTER.2015.24
35. Sant'Ana, L., Cordeiro, D., de Camargo, R.Y.: Plb-hac: dynamic load-balancing for heterogeneous accelerator clusters. In: European Conference on Parallel Processing, pp. 197–209. Springer (2019)
36. Souravlas, S., Anastasiadou, S.: Pipelined dynamic scheduling of big data streams. Appl. Sci. (2020). https://doi.org/10.3390/app10144796
37. Suettlerlein, J., Zuckerman, S., Gao, G.R.: An implementation of the codelet model. In: Proceedings of the 19th International Conference on Parallel Processing, Euro-Par'13, pp. 633–644. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_63
38. Teodoro, G., Kurc, T.M., Pan, T., Cooper, L.A.D., Kong, J., Widener, P., Saltz, J.H.: Accelerating large scale image analyses on parallel, CPU–GPU equipped systems. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 1093–1104 (2012). https://doi.org/10.1109/IPDPS.2012.101
39. Tribbey, W.: Modern database systems. In: Kim, W. (ed.) Modern Database Systems, Chap. Numerical Recipes: The Art of Scientific Computing (3rd Edition) is Written by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, and Published by Cambridge University Press, 2007, Hardback, ISBN 978-0-521-88068-8, 1235 pp., pp. 30–31. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995). https://doi.org/10.1145/1874391.187410
40. Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., Emer, J.: Scheduling heterogeneous multi-cores through performance impact estimation (pie). SIGARCH Comput. Archit. News **40**(3), 213–224 (2012). https://doi.org/10.1145/2366231.2337184
41. van Werkhoven, B., Maassen, J., Seinstra, F.J., Bal, H.E.: Performance models for CPU–GPU data transfers. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 11–20 (2014). https://doi.org/10.1109/CCGrid.2014.16
42. Wang, Z., Tournavitis, G., Franke, B., O'boyle, M.F.P.: Integrating profile-driven parallelism detection and machine-learning-based mapping. ACM Trans. Archit. Code Optim. **11**(1), 1–26 (2014). https://doi.org/10.1145/2579561
43. Wen, Y., O'Boyle, M.F.: Merge or separate?: multi-job scheduling for opencl kernels on CPU/GPU platforms. In: Proceedings of the General Purpose GPUs, GPGPU-10, pp. 22–31. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3038228.3038235

44. Yang, C., Wang, F., Du, Y., Chen, J., Liu, J., Yi, H., Lu, K.: Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: IEEE International Conference on Cluster Computing, pp. 19–28 (2010). https://doi.org/10.1109/CLUSTER.2010.12
45. Zhang, F., Wu, B., Zhai, J., He, B., Chen, W.: Finepar: irregularity-aware fine-grained workload partitioning on integrated architectures. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 27–38 (2017). https://doi.org/10.1109/CGO.2017.7863726
46. Zhang, F., Zhai, J., He, B., Zhang, S., Chen, W.: Understanding co-running behaviors on integrated CPU/GPU architectures. IEEE TPDS **28**(3), 905–918 (2017). https://doi.org/10.1109/TPDS.2016.2586074
47. Zhong, Z., Rychkov, V., Lastovetsky, A.: 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) (2012)
48. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a "codelet" program execution model for exascale machines: position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-tuning Computing Systems for the Exaflop Era, EXADAPT '11. ACM, New York, NY, USA (2011)

Authors and Affiliations

Tongsheng Geng[1] · Marcos Amaris[2] · Stéphane Zuckerman[3] · Alfredo Goldman[4] · Guang R. Gao[5] · Jean-Luc Gaudiot[1]

✉ Tongsheng Geng
tgeng@uci.edu

Marcos Amaris
amaris@ufpa.br

Stéphane Zuckerman
stephane.zuckerman@cyu.fr

Alfredo Goldman
gold@ime.usp.br

Guang R. Gao
ggao.capsl@gmail.com

Jean-Luc Gaudiot
gaudiot@uci.edu

[1] University of California Irvine, Irvine, CA, USA

[2] Federal University of Pará, Tucuruí, PA, Brazil

[3] Laboratoire ETIS, UMR 8051, CY Cergy Paris Universités, ENSEA, CNRS, 95000 Cergy, France

[4] University of São Paulo, São Paulo, Brazil

[5] University of Delaware, Delaware, USA