J Supercomput (2018) 74:5399–5431
https://doi.org/10.1007/s11227-018-2435-1

E-OSched: a load balancing scheduler for heterogeneous multicores

Yasir Noman Khalid[1] · Muhammad Aleem[1] · Radu Prodan[2] · Muhammad Azhar Iqbal[1] · Muhammad Arshad Islam[1]

Published online: 23 May 2018

Abstract The contemporary multicore era has adhered to the heterogeneous computing devices as one of the proficient platforms to execute compute-intensive applications. These heterogeneous devices are based on CPUs and GPUs. OpenCL is deemed as one of the industry standards to program heterogeneous machines. The conventional application scheduling mechanisms allocate most of the applications to GPUs while leaving CPU device underutilized. This underutilization of slower devices (such as CPU) often originates the sub-optimal performance of data-parallel applications in terms of load balance, execution time, and throughput. Moreover, multiple scheduled applications on a heterogeneous system further aggravate the problem of performance inefficiency. This paper is an attempt to evade the aforementioned deficiencies via initiating a novel scheduling strategy named OSched. An enhancement to the OSched named E-OSched is also part of this study. The OSched performs the resource-aware assignment of jobs to both CPUs and GPUs while ensuring a balanced load. The load balancing is achieved via contemplation on computational requirements of jobs and computing potential of a device. The load-balanced execution is beneficiary in terms of lower execution time, higher throughput, and improved utilization. The E-OSched reduces the magnitude of the main memory contention during concurrent job execution phase. The mathematical model of the proposed algorithms is evaluated by comparison of simulation results with different state-of-the-art scheduling heuristics. The results revealed that the proposed E-OSched has performed significantly well than the state-of-the-art scheduling heuristics by obtaining up to 8.09% improved execution time and up to 7.07% better throughput.

✉ Muhammad Aleem
aleem@cust.edu.pk

[1] Capital University of Science and Technology, Islamabad 44000, Pakistan

[2] Alpen-Adria-Universität, 9020 Klagenfurt, Austria

Keywords Scheduling · Data-parallel applications · Heterogeneous multicores · Load balancing

1 Introduction

Multicore-based execution plays a pivotal role to maintain high speedup for compute-intensive scientific applications. Over the past few years, the paradigm of the single-core era is gradually being shifted toward multi-/many-core devices such as heterogeneous processing devices ranging from smartphones [35] to supercomputers [33]. Heterogeneous processing devices are generally equipped with a general-purpose multicore *Central Processing Unit* (CPU) and a many-core *Graphics Processing Unit* (GPU). Recent advances in computer architecture have modeled the GPUs in such a way that they have been transformed into immensely programmable and proficient to conduct general-purpose computing. GPUs are basically deemed to be extremely efficient in terms of high speedup execution for the scientific applications that involve an excessive amount of parallel computations [2, 30]. To program GPUs, *Open Compute Language* (OpenCL) [29] has emerged as an industry standard for programming data-parallel applications. Due to its portable nature, OpenCL applications can be executed on many types of processors and accelerators including CPUs, GPUs, FPGAs, etc. [34, 40]. An OpenCL program comprises of two parts, (1) *host program*, and (2) *kernel function*. Supervision of kernel function execution on accelerator devices (i.e., CPU, GPU) is the responsibility of a host program that executes on a CPU device [28]. The kernel function deals with the compute-intensive part of the code and is programmed using data-parallel programming model. Within an OpenCL program, buffer creation, data transfer, and kernel mapping to the devices are manually managed by the application developers [36].

The conventional scheduling mechanisms adhere to employ GPUs for the execution of compute kernel whereas the host program's execution is performed by the CPU device [38]. Such scheduling strategies can cause certain problems, for instance, CPU remains idle and underutilized during the kernel function execution causing a longer program execution time and load imbalance. Over the past few years, the research community is eagerly focused to overcome the aforementioned issues via splitting a compute kernel to utilize both the CPU and GPU devices simultaneously within a heterogeneous machine [1, 4–6, 13, 17]. The simultaneous utilization of both CPU and GPU devices for execution often results in load-balanced execution, better device utilization, and reduced execution time.

The mainstream usage of multicore heterogeneous machines stipulates to employ a scheduling support to efficiently utilize the computing resources and to reduce the overall execution time of the compute-intensive data-parallel applications [39]. Moreover, in a data-center or Cloud computing environment, dependence on central scheduler rather than application developer plays a more significant role in improving the resource utilization and reducing the execution time for the job pool [26]. Single job-based[1] scheduling strategies are not appropriate for the scenarios where several

[1] In this research job terminology is used to define an OpenCL application that consists of a host program and kernel functions.

jobs are being submitted by different users. The single job scheduling techniques are often job-oriented and employ kernel profiling and code-splitting to utilize both CPUs and GPUs. In addition to the application code change, devising a generic single job-based scheduling technique is a difficult task. Moreover, the results obtained from the split kernels must be accurately combined to produce the correct results. Machine learning-based scheduling strategies devised for a job pool require kernel profiling to assign a kernel to the device on which the job will execute faster [38, 39]. Moreover, other scheduling schemes [11, 15] require the applications execution provenance to decide the appropriate target device that will enhance overall throughput of the job pool. Most of the job-pool scheduling schemes [15, 20, 38, 39], require extensive offline profiling to schedule compute kernels. Furthermore, these scheduling schemes do not consider load balancing as a major factor to decrease the execution time of the job pool [11, 15, 38, 39]. Therefore, there is a need for job scheduler (to map data-parallel compute-intensive applications) that does not require application code change, balances the load across the employed heterogeneous computing devices to reduce the execution time of the job pool, to increase throughput, and to improve device utilization.

Moreover, such scheduler should also consider that the CPU device executes data-parallel applications much slower than the GPU device (due to the less parallelism available in CPUs). However, applications with the small data foot-prints get efficiently executed on a CPU device as compared to a GPU device due to the data-transfer overheads, underutilization of GPU resources, and small computation to communication ratio for a GPU device.

To highlight the discussed scenario, Fig. 1 shows the execution of *Bitonic sort* application from AMD [3]. We have executed Bitonic sort application using 09 different input data sizes and application is mapped to both CPU and GPU, separately. Figure 1 shows that with the small data size (less number of values to be sorted) the CPU and the GPU-based executions have exhibited the similar performances. Whereas with the increased data size, the execution time of the CPU-based application has increased exponentially, while the GPU-based execution has shown a linear increase in the execution time. As shown in Fig. 1, for the largest data size (i.e., 268,435,456 values) the GPU-based execution has resulted in 05 times reduced execution time as compared to the CPU-based execution of the application. Therefore, such heterogeneous scheduler should be designed that maps the small data-sized jobs on CPUs while the jobs based on large data size should be mapped on GPUs. In addition to that, the load balance factor should not be ignored at all so that all the employed devices (i.e., CPUs and GPUs) accomplish the execution of assigned jobs within the approximately same time duration. A load-balanced and application suited mapping (small jobs on CPUs and large on GPUs) often results in reduced execution time for the job pool, improved resource utilization, and higher throughput.

Therefore, we propose an *OpenCL Scheduler* (OSched) that considers jobs' computing requirement, processing capabilities of the devices, and jobs' data size to balance the load across the heterogeneous computing devices. The load-balanced execution by the OSched results in a reduced execution time of the job pool, maximized throughput, and increased resource utilization (as evident by our experiments). In general, the applications with lower computation requirements tend to schedule to *slower*

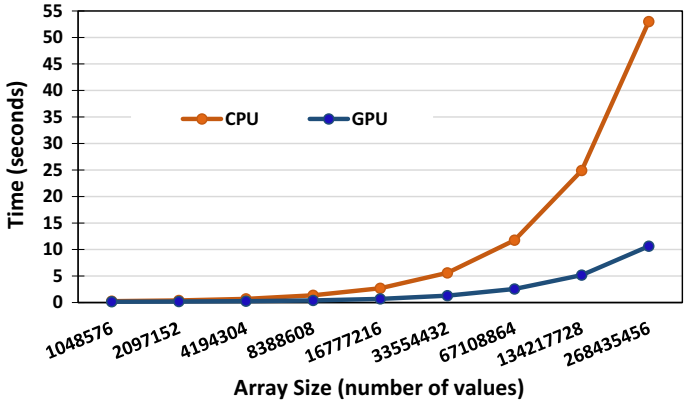

Fig. 1 Execution of bitonicsort—CPU versus GPU

processing devices and vice versa. Moreover, a certain compute device (a CPU or a GPU) is assigned with computations according to its computing capability while maintaining a load balance execution within the heterogeneous multicore machine. The load-balanced assignment of jobs ensures the reduced execution time, higher throughput, and better device utilization. Moreover, an enhancement of the proposed OSched scheduler named as *Enhanced-OSched* (E-OSched) is also proposed to further reduce the execution time (of the job pool) and increase the throughput via mitigation of memory contention.

The rest of the paper is organized as follows. Section 2 presents the related work and the proposed methodology is detailed in Sect. 3. Section 4 presents the experimental setup, the experimented scheduling policies, the evaluation metrics, and the experimental results. Section 5 concludes the paper.

2 Related work

Currently, there exists a plethora of scheduling techniques for heterogeneous multicore machines [5, 11, 15, 17, 23, 24, 27, 38]. A few of these scheduling techniques [5, 23, 24, 27] split either a single application kernel among CPUs and GPUs or attempt to schedule a pool of applications [11, 15, 38] to diminish the overall execution time of the jobs and to improve the device utilization.

In Grewe and O'Boyle [17], *int operations*, *float operations*, *barriers*, *work items* etc., based code features have been extracted from the OpenCL kernels at compile time. These extracted features are then passed to machine learning-based predictive model, i.e., *Support Vector Machine* (SVM) to make the scheduling decisions (whether to map a kernel to a CPU, GPU, or to partition the kernel among available computing devices). In contrast to our proposed work, the authors have presented an algorithm for scheduling a single OpenCL kernel across heterogeneous devices; whereas, our proposed approach is employed to schedule job pool of applications. The authors' work has further been extended by Ghose et al. [13], wherein a machine learning-based

scheduling approach is proposed. The extended scheduling mechanism works on the basis of *branch divergence* (i.e., *if, else, statements,* etc.) for optimal job scheduling. Authors have emphasized that the branch divergence plays a critical role in application execution; therefore, this code feature should be employed while building a machine learning-based scheduling model. In contrast to the proposed scheduler by the authors, our technique does not require offline profiling and training. Moreover, the proposed techniques by the authors do not ensure a load-balanced scheduling of tasks.

Chen and Marculescu [10] presented *Choose-between-Accelerate-the-fastest-and-Best-fit* (CAB) and *Greedy-Increase* (GrIn). The *CAB* is a mathematical model that considers performance and energy constraints for optimal task scheduling on a heterogeneous multicore system. *GrIn* algorithm is an implementation of the general case of *CAB* mathematical model. According to the authors, each application program consists of a set of tasks (both *serial* and *parallel*). First, the minimum and the maximum throughput for each task-type (for all the heterogeneous processors) is calculated using the *CAB* model. The tasks consuming the minimum system energy and having the minimum *Energy Delay Product* for a processor represents the highest throughput for that processor. In order to maximize system throughput, the *GrIn* scheduling algorithm assigns a task (using a greedy approach) to a processor. This research is analogous to our work for considering processing speed of processors; however, memory contention aspect has been ignored.

In Luk et al. [27], a system named *Qilin* for programming heterogeneous devices has been presented. At first, the *Qilin* system partitions the kernel data into two parts, which are further mapped onto CPU and GPU devices. The *Qilin* stores the execution performance of an application in a database. Afterward, the *Qilin* uses the previously stored execution performance data of a certain application to project current execution performance to form scheduling decisions accordingly. In case of any system hardware changes, the *Qilin* initiates a new training session for the altered hardware configurations. In contrast to the *Qilin*, our scheduling algorithm does not require offline profiling and code change for the OpenCL applications.

In Albayrak et al. [1], a profiling-based kernel scheduling method is proposed in which offline profiling has been performed to obtain execution time and data dependencies of an OpenCL kernel. Afterward, a greedy algorithm has been employed to schedule the kernel on a computing device (i.e., CPU or GPU) that resulted in the least execution time without dependency violations. However, the proposed algorithm only schedules kernels of a single application. Whereas, our proposed scheduler does not require offline code profiling and assigns a pool of jobs to CPUs and GPUs.

StarPU [4] is a runtime system that provides a unified execution environment for executing numerical kernels on heterogeneous architectures. The StarPU adopts simple scheduling policies for tasks distribution on heterogeneous architecture. The employed *Greedy policy* (priority-based) assigns a task to a processor as soon as it becomes idle. The tasks get executed based on their priority-preference (i.e., high priority task will be executed first). *No-prior* policy [4] is same as the greedy approach; however, it does not consider task priority while assigning a task to the processor. The employed *Ws-policy* also assigns task greedily using the work-stealing mechanism. In *w-rand policy*, each worker-device (i.e., processor) is assigned an acceleration factor. A task is assigned to that worker-device whose probability is proportional to acceleration

factor ratio. In *heft-tm policy*, a task is assigned to a computing unit which minimizes the task's execution time (considering the already assigned tasks on that compute-unit). The StarPU's scheduling policies only consider *numeric kernels* (such as Matrix Multiplication, LU decomposition), while our proposed scheduling heuristic is capable to schedule different application kernels and employing multi-GPU configurations.

In Becchi et al. [5], a runtime system has been proposed that schedules *legacy kernels* (compute-intensive part of applications) on a heterogeneous machine considering both the execution history and data-transfer overheads. Authors have proposed a runtime system that intercepts function-calls of a kernel and schedules them on either a CPU or a GPU device. Our proposed approach is distinguished as compared to the scheduler presented in Becchi et al. [5] in terms of no overhead of code profiling. Moreover, our proposed scheduler is capable of scheduling several data-parallel applications as compared to the single application-based scheduling approach adopted by Becchi et al. [5].

HDSS [6] scheduling mechanism improves the execution time of a kernel via partitioning the workload among CPU and GPUs. Initially, a profiling phase is utilized to learn the computing power of each processor (by assigning a small number of loop iterations). In the adaptive phase, remaining loop iterations are assigned to each processor according to their processing speed. As a result, the HDSS mechanism ensures a load-balanced execution on heterogeneous machines. In contrast to the HDSS, our approach does not require job splitting and code change.

In Binotto et al. [7], a runtime system has been proposed that allocates OpenCL-based data-parallel tasks to a CPU or a GPU device. In this system, a kernel is divided into several *tasks*. Afterward, a profiler is employed to record the execution time of the scheduled tasks against each processor. Whenever a task arrives for execution, it is scheduled to either a CPU or a GPU device based on the stored performance profile. However, this system requires profiling and code changes which is not the case with our proposed approach.

The algorithm (based on single kernel) presented in Boyer et al. [8] divides workload into *chunks* and schedule these to a CPU and a GPU device. The scheduler [8] starts with the assignment of uniform small-sized chunks for each device. The chunk sizes are increased/decreased exponentially according to the previous executions. In such a way, the faster computing devices are loaded with large-sized chunks and slower devices are loaded with small-sized chunks which ensure a good load balance. In contrast to this approach, our scheduler is memory contention aware that ensures load-balanced execution with low execution time.

In Choi et al. [11], *Estimated Execution Time* of an application is used to decide allocation to a CPU or a GPU. This technique requires a training period in which the execution history of an application is observed. When an application arrives for execution, it is mapped to a device which is capable to complete this job earlier. Application completion time is estimated by considering the total execution time of the application and the execution time of the prior scheduled applications on that device. In contrast, our proposed schedule scheme does not require offline profiling. Moreover, our proposed technique supports multi-device configuration which is not provided by this heuristic [11].

In Gregg et al. [15], historical runtime data is used to schedule an application to a suitable device that can execute the job earlier. If that suitable device is busy executing other applications then the application is scheduled on a slower device. In contrast to this approach, our proposed scheduler is capable of employing multi-GPU configurations.

Gregg et al. [16] proposed an algorithm that schedules applications on heterogeneous multicores by considering the *system specifications*, the historical *runtime data*, and *current system state*. In contrast to this approach, our scheduler considers both the computational requirements of an application and computing capabilities of devices. Moreover, our proposed approach schedules pool of applications to ensure load balancing and memory contention-free execution.

In Jiménez et al. [20], three different scheduling algorithms have been proposed. Two algorithms are based on different variants of *First Free* and *First Come First Serve* heuristics. The third algorithm assigns tasks to a processor via considering the execution history. The performance history is then harnessed to predict waiting time for a task on a certain processor. On the other hand, our proposed scheduler minimizes the memory contention issue and does not require changes in code.

In Kofler et al. [22], an offline prediction model is proposed to dynamically partition tasks between a CPU and a GPU. Machine learning techniques, based on *Artificial Neural Network* (ANN), are used to derive prediction model for task partitioning using the *Insieme* [19] runtime system. This model depends on static code features (e.g., *OpenCL built-in functions*) and dynamic input sensitive features (e.g., *data-transfer size* of the split-able buffer) for the training phase. The *Principal Component Analysis* procedure is then used to further optimize task partitioning. After that, the partitioned tasks are assigned to CPU and GPU for execution. In contrast to this approach, our proposed scheduler does not require data-splitting and offline training.

Lee et al. [25] presented a *Single Kernel Multiple Data* scheduling heuristics that employs splitting of a kernel across all available devices. For partitioning, the data represented by the *ND Range* is flattened and its subset is assigned to each computing device. The partial subset of results are obtained and merged in a seamless manner to produce the output. To ensure load-balanced distribution across multiple heterogeneous devices, the execution speed and data-transfer cost to each device are considered. However, our proposed scheduler does not require data-splitting and is capable of scheduling a pool of jobs considering both the application and device's computing requirements.

Pandit et al. [31] presented an OpenCL runtime system called *FluidiCL* that is capable of distributing and executing an OpenCL program using both CPU and GPU devices. This scheme does not require any prior offline training. It automatically handles data-transfers and results aggregation without involving the programmer. For data-distribution, the n-dimensional *ND Range* of a workgroup is flattened and used as a unit of allocation for execution. A kernel mapped on a GPU device starts executing the flattened workgroup from one end while on a CPU device a sub-kernel starts executing another part of that workgroup. In contrast to this approach, our proposed scheduler is capable of executing multiple jobs simultaneously (using CPUs and GPUs) providing a better application response time.

Lee et al. [23] presented a two-phase scheduling algorithm named *Multi Kernel on Multi Devices* (MKMD) to schedule multiple kernels of an application. In the first phase, a kernel is assigned to the device that minimizes the kernel's execution time and data-transfer cost. In the second phase, a kernel is split into sub-kernels and re-scheduled to heterogeneous processors to improve device utilization. The MKMD heuristic builds a regression model for each kernel considering different input sizes and device mappings. The regression model is then used to decide whether a kernel should be split or mapped completely on a certain compute device. However, our proposed scheduler does not split kernels across heterogeneous devices and does not require offline profiling.

In Kaleem et al. [21], two scheduling strategies are proposed to partition the kernel workload between a CPU and a GPU. In naïve profiling step, a small portion of the work is assigned to both CPU and GPU devices and the execution performance of both devices is analyzed. Afterward, the collected profiling data is utilized for further job assignments. On the other hand, our scheme does not require kernel splitting and can efficiently schedule pool of kernel jobs.

Wen et al. [39] presented a machine learning-based task scheduling scheme to schedule multiple kernels from different programs. The focal aspect of this technique is to enhance the system throughput and decrease the average turn-around time. The distinguishing factor of this scheme is that authors have contemplated scheduling of multiple OpenCL applications on a heterogeneous platform. It considers (a) *static code features* such as a number of instructions, load/store operations, etc.; and (b) *runtime features* such as input size. Using both the static/dynamic features and a predictive model, the data-parallel programs are categorized into high and low speedup classes. The high speedup programs are scheduled on a GPU device and the low speedup programs are scheduled on CPU. In contrast to this technique, our proposed scheduler assigns OpenCL kernels on the basis of application's computing requirements and the device's computing capabilities.

In Ravi and Agrawal [32], a dynamic scheduling scheme is presented for applications that are characterized by (*a*) *generalized loop reduction* and (*b*) *structured grid computation*. These applications consist of data-parallel loops, which are divided into *chunks*. These chunks are further divided into *chunk-lets* that represent a basic unit of assignment to a CPU or a GPU device. Using the FCFS-based mapping, a CPU is assigned to a single chunk-let at a time while several chunk-lets are combined and assigned to a GPU. Employing this methodology, the faster computing devices (such as GPUs) are loaded with more tasks as compared to the slower computing devices (i.e., CPUs). This task-mapping scheme ensures a good load balance across the computing devices. In contrast to this technique [32], our proposed scheduling scheme does not require kernel splitting and is used to schedule pool of jobs.

In Wang et al. [37], a two-phase scheduling scheme called CAP is proposed for heterogeneous parallel machines. In the first phase, a static partitioning method is used to distribute a small portion of workload equally to both a CPU and a GPU device. The execution time of the assigned workload is analyzed. Considering the execution time of the previous assignment, the amount of work is increased twofold on the faster device. The scheduling with the increased workload is performed until the variance between current and the previous executions becomes less than a predefined threshold. In the

second phase, the remaining workload is divided among compute devices according to the sampling done in the first phase. In contrast to this scheme, our proposed scheduler does not require prior execution analysis of the workload.

In Wen et al. [38], a machine learning-based heuristic is proposed that employs OpenCL code features (such as *instructions*, *blocks*, *math functions,* etc.) to determine device suitability. Moreover, certain code features (e.g., *branch ratio*, *data size*, etc.) are utilized to determine whether to schedule a kernel in isolation (to a GPU) or to combine it with other kernels to improve execution performance. In contrast, our proposed scheduler considers devices' computing capabilities to balance the load of job pool with memory contention-free execution of multiple data-parallel applications.

In summary, most of the contemporary state-of-the-art techniques are either concerned with the single kernel-based scheduling or limited to the scheduling of certain kind of applications. Several existing heuristics require an offline training or profiling along with some code changes. To the best of our knowledge, there does not exist any technique that emphasizes on the load-balanced scheduling of job pool without requiring profiling and offline training. Considering all aforementioned deficiencies, our proposed scheduling heuristic contemplates both the computational requirements of a job, computing capabilities of devices, and memory contention-free mapping of OpenCL jobs. Table 1 presents the summary of all the scrutinized state-of-the-art scheduling techniques.

In a more coherent way, the contributions of the proposed scheme are as follow:

- Significant scrutinization of contemporary state-of-the-art scheduling techniques for heterogeneous machines that provides a comprehensive outline to understand the shortcomings of the existing scheduling heuristics;
- Two novel scheduling schemes *OSched* and *E-OSched*, performing the resource-aware assignment of data-parallel jobs on CPUs and GPUs to reduce the make-span of the job pool, to increase throughput, to increase device utilization, and to reduce memory contention;
- Mathematical modeling of the proposed *OSched* and *E-OSched* algorithms;
- Experimental evaluation to justify the concept of the proposed *OSched* and *E-OSched* algorithms in terms of load-balanced execution with lower execution time, higher throughput, and improved resource utilization as compared to the state-of-the-art scheduling schemes.

3 OpenCL Scheduler

We propose *OpenCL Scheduler* (OSched) that assigns jobs to CPUs and GPUs in load-balanced manner to improve device utilization, increase throughput, and reduce the execution time of a job pool. The OSched maps jobs in a load-balanced manner by contemplating the computational requirements of jobs and processing capabilities of the devices. All the submitted jobs are arranged in the job pool according to their computational requirements (i.e., *smaller size jobs first*) where the first half (of the job pool) contains less computational intensive jobs and the second half comprises of jobs requiring high computation power. Jobs involving low computational requirements (i.e., first pool segment) are mapped to CPU (having low computing power) while the

Table 1 Summary of the related work

References	Attributes							
	Scheduling type	Job allocation	Load balancing	Device migration support	Code change	Resource-aware	Application-aware	Memory contention aware
OSched	Static	Job pool	Yes	No	No	Yes	No	No
E-OSched	Static	Job pool	Yes	No	No	Yes	No	Yes
Grewe and O'Boyle [17]	Static	Single job	No	No	No	No	Yes	No
Chen and Marculescu [10]	Static	Job pool	Yes	No	No	Yes	No	No
Ghose et al. [13]	Static	Single job	No	No	No	No	Yes	No
Luk et al. [27]	Dynamic	Single job	Yes	No	Yes	Yes	No	No
Albayrak et al. [1]	Dynamic	Single job	No	Yes	No	Yes	No	No
Augonnet et al. [4]	Dynamic	Single job	Yes	No	Yes	Yes	No	No
Becchi et al. [5]	Dynamic	Single job	No	Yes	Yes	No	No	No
Belviranli et al. [6]	Dynamic	Single job	Yes	No	Yes	Yes	No	No
Binotto et al. [7]	Dynamic	Single job	Yes	Yes	Yes	Yes	No	No
Boyer et al. [8]	Dynamic	Single job	Yes	No	Yes	Yes	No	No
Choi et al. [11]	Dynamic	Single job	No	No	No	Yes	No	No
Gregg et al. [15]	Dynamic	Job Pool	Yes	No	No	Yes	No	No
Gregg et al. [16]	Dynamic	Single job	No	No	No	Yes	No	No
Jiménez et al. [20]	Dynamic	Job Pool	Yes	No	Yes	No	No	No

Table 1 continued

References	Attributes							
	Scheduling type	Job allocation	Load balancing	Device migration support	Code change	Resource-aware	Application-aware	Memory contention aware
Kofler et al. [22]	Dynamic	Single job	No	No	Yes	No	Yes	No
Lee et al. [25]	Dynamic	Single job	Yes	No	No	Yes	No	No
Pandit and Govindarajan [31]	Dynamic	Single job	No	No	Yes	No	No	No
Lee et al. [23]	Dynamic	Single job	Yes	Yes	Yes	No	No	No
Kaleem et al. [21]	Dynamic	Single job	Yes	No	Yes	Yes	No	No
Wen et al. [39]	Hybrid	Job Pool	No	No	No	No	Yes	No
Ravi and Agrawal [32]	Hybrid	Single job	Yes	No	No	Yes	No	No
Wang et al. [37]	Hybrid	Single job	Yes	No	No	Yes	No	No
Wen and O'Boyle [38]	Hybrid	Job Pool	No	No	No	No	Yes	No

Table 1 continued

References	Attributes						
	Implementation	Data size consideration	Accelerator vendor support	Multi-GPU support	Throughput maximization	Requires profiling	Provenance data usage
OSched	Runtime system	Yes	Any	Yes	Yes	No	No
E-OSched	Runtime system	Yes	Any	Yes	Yes	No	No
Grewe and O'Boyle [17]	Runtime system	Yes	Any	No	NA	Yes	No
Chen and Marculescu [10]	Runtime system	No	Any	Yes	Yes	No	No
Ghose et al. [13]	Runtime system	Yes	Any	No	NA	Yes	No
Luk et al. [27]	API	Yes	NVIDIA	No	NA	Yes	Yes
Albayrak et al. [1]	Runtime system	No	Any	No	NA	Yes	Yes
Augonnet et al. [4]	API	No	NVIDIA	No	NA	Yes	Yes
Becchi et al. [5]	Runtime system	Yes	NVIDIA	No	NA	Yes	Yes
Belviranli et al. [6]	API	No	NVIDIA	No	NA	No	Yes
Binotto et al. [7]	Library	Yes	Any	Yes	NA	Yes	Yes
Boyer et al. [8]	Manual	No	Any	Yes	NA	Yes	Yes
Choi et al. [11]	Runtime system	No	NVIDIA	No	No	Yes	Yes
Gregg et al. [15]	Runtime system	Yes	Any	No	Yes	Yes	Yes
Gregg et al. [16]	Runtime system	No	Any	No	NA	Yes	Yes

Table 1 continued

References	Attributes						
	Implementation	Data size consideration	Accelerator vendor support	Multi-GPU support	Throughput maximization	Requires profiling	Provenance data usage
Jiménez et al. [20]	Runtime system	No	NA	No	No	Yes	Yes
Kofler et al. [22]	Runtime system	Yes	Any	Yes	NA	Yes	No
Lee et al. [25]	Middleware	Yes	Any	Yes	NA	Yes	No
Pandit and Govindarajan [31]	Runtime system	Yes	Any	No	NA	Yes	No
Lee et al. [23]	Library	Yes	Any	No	NA	Yes	Yes
Kaleem et al. [21]	Runtime system	No	Intel	No	NA	Yes	Yes
Wen et al. [39]	Runtime system	Yes	Any	No	Yes	Yes	No
Ravi and Agrawal [32]	Runtime system	No	NA	No	NA	Yes	No
Wang et al. [37]	Runtime system	No	NVIDIA	No	No	Yes	Yes
Wen and O'Boyle [38]	Runtime system	Yes	Any	No	Yes	Yes	No

jobs having high computational requirements (second pool segment) are scheduled on a GPU device. Each device (either a CPU or a GPU) is assigned to the pool segment considering the device's computational share or capability. This load-balanced mapping of jobs ensures least execution time for the job pool, higher throughput, and improved device utilization.

3.1 System architecture

The *OSched*-based scheduling system comprises three layers: (1) *Hardware*, (2) *System Software*, and (3) *the OSched* scheduling layers. The OSched-based scheduling system is depicted in Fig. 2. The *hardware layer* is the bottom-most layer that contains a heterogeneous multicore machine based on CPUs and GPUs. At top of the *hardware layer* is the *System Software* layer that consists of *Operating System* and *OpenCL Runtime* [28]. The OSched scheduler is at the top-most layer of the heterogeneous system. The user of a system submits jobs of varying sizes (depicted by the different sized circles in Fig. 2). Afterward, the computational requirements of each submitted job is calculated by harnessing the job's computational complexity.

For instance, a matrix multiplication job completion requires $2\ N^3$ operations to complete, where N is a dimension of a square matrix. All the submitted jobs are arranged in ascending order (smaller size jobs first) in the job pool considering their computational requirements. The *Resource Manager* (shown in Fig. 2) is responsible for extraction of the hardware resource information (i.e., *processors detail*) of the machine. The *processing speed* of a processor (based on multiple computing cores) is measured in *FLOPS*[2] that is calculated using following equation [12]:

$$Processing\_Speed = number\_of\_cores \times \frac{cycle}{second} \times \frac{flops}{cycle},$$

where, *number_of_cores* represents total cores of a processor, *cycles/second* represents clock frequency of a core (in Hz), and *flops* represents total number of floating point operations. The *Job scheduler* divides the job pool into *CPU job queue* (represented as CJQ in Fig. 2) and *GPU job queues* (represented as $GJQ_{1\_to\_n}$). The reason for a single *CJQ* is that the OpenCL, by default, considers a CPU (even if there are multiple cores) as a single device. The decision of the job-pool division (into CJQ and $GJQ_{1\_to\_n}$) and device selection is determined pursuant to the computational requirements of jobs and the computing power of each processor. After that, the jobs from CJQ and $GJQ_{1\_to\_n}$ are assigned to the respective processor for execution (based on *First In First Out* (FIFO)).

The complete mechanism of the *OSched* is illustrated in Fig. 3. The first step involves calculation of computational requirements (of jobs) and computing power of devices. Next, the *CPU computational share* (depicted as CS_{cpu} in Fig. 3) is calculated. The first segment of the pool (consisting of smaller size jobs) is allocated to a CPU and the second segment (larger size jobs) is allocated to GPUs. First, the CPU-based

[2] FLOPS = Floating Point Operations Per Second.

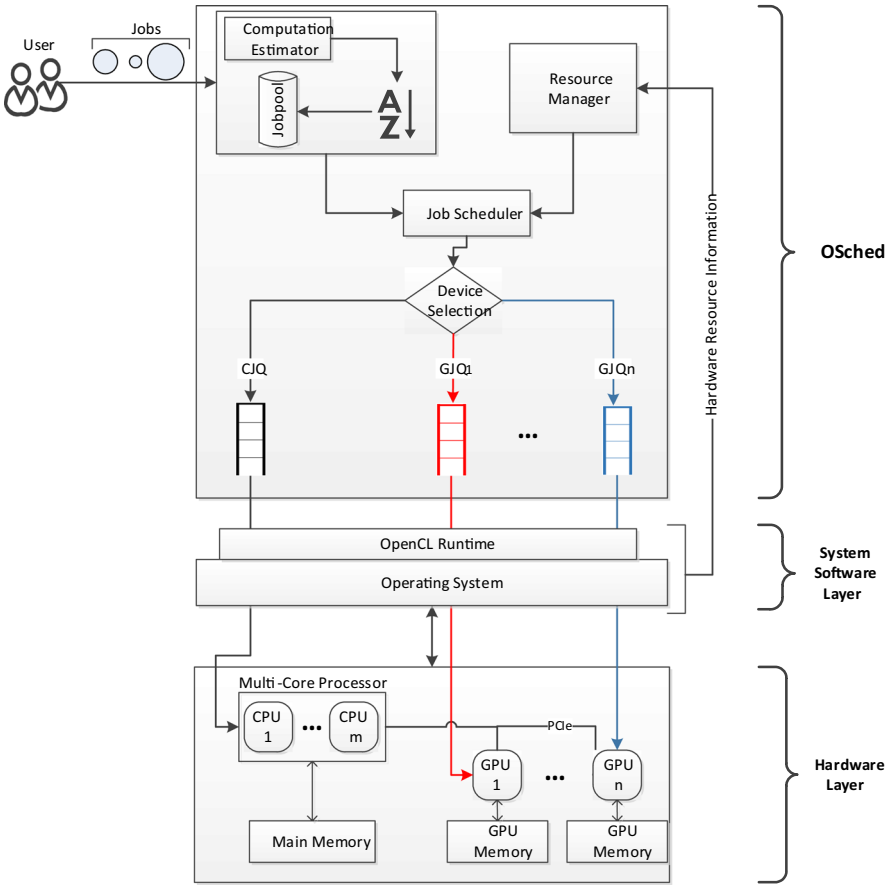

Fig. 2 System architecture of the OSched

scheduling is performed by mapping $m-1$ jobs (where m represents the last job that can be assigned to a CPU) such that all the mapped jobs' computing requirement is less than or equal to the CPU's computational share. The mth job is mapped to the CPU only if the CPU's computational share is higher than the mapped $m-1$ jobs' computing requirements. In addition to that, the CPU's share must not exceed CPU's total computational requirement (after adding 2/3rd computational requirements of mth job to the CPU's share). Once all the jobs of the first segment are mapped to CPU device, the mapped jobs are removed from the job pool. Subsequently, the GPUs job assignment phase commences in a similar fashion. The exception is for last GPU (GPU_N or only GPU device on the machine) in a heterogeneous machine wherein all the remaining jobs of the pool are assigned to it. The dotted rectangle (depicted in Fig. 3) represents an optimization (discussed in Sect. 3.4) to the basic *OSched* heuristic.

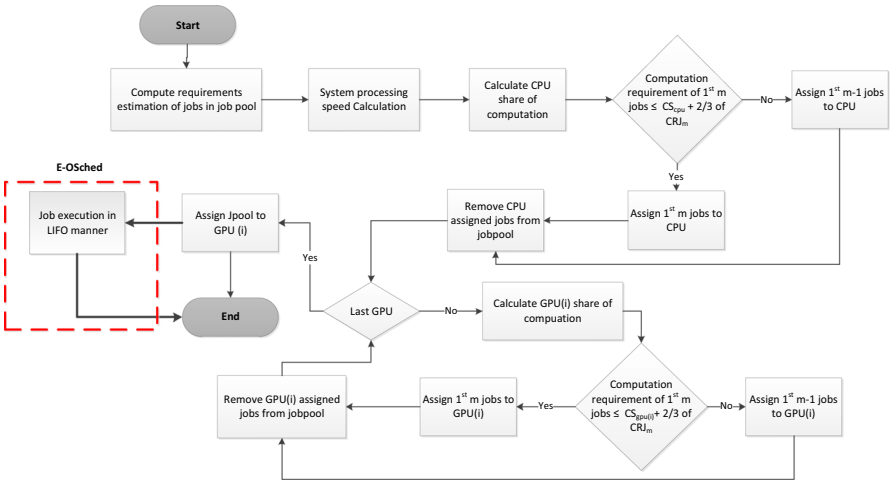

Fig. 3 OSched job scheduling flowchart

3.2 OSched system model

This section presents the mathematical model of the proposed *OSched* scheduling technique. Table 2 lists the terminologies used for the description of the OSched mathematical model.

Concerning jobs execution in job pool J, where $J = \{J_1, J_2, \ldots, J_k | CRJ_i \leq CRJ_{i+1}\}$ is a set of k jobs, in a sorted order concerning the *computation requirement of a job* (represented as *CRJ*). To assign jobs to processors, let $P = \{P_1, P_2, \ldots, P_m | PS_i \leq PS_{i+1}\}$ be a set of m processors, in sorted order with respect to the *processing speed* (*PS*).

In order to determine job assignment to a CPU and GPUs, *total computation requirement* (J_{CR}) of all the jobs in J and *total processing speed of all processors* (T_{CP}) is required (that is calculated through following equations):

$$J_{CR} = \sum_{i=1}^{n} CRJ_i \tag{1}$$

$$T_{CP} = \sum_{i=1}^{m} PS_i \tag{2}$$

where, m represents the processing speed of m computing devices (a multicore CPU and GPUs). From Eq. (1) and Eq. (2), the *computational share of CPU* (CS_{cpu}), which is the portion of computation that is assigned to a CPU from J_{CR}, can be calculated using the formula given in Eq. (3).

$$CS_{cpu} = \left(\frac{P_C}{T_{CP}} \times J_{CR} \right) - \alpha \tag{3}$$

Table 2 List of notations used in OSched system model

Notations	Description
J	Set of jobs
CRJ	Computation requirement of a job
P	Set of processors
PS	Processing speed
J_{CR}	Total computation requirement of all the jobs in J
T_{CP}	The total processing speed of all processors
CS_{cpu}	The computational share of CPU
P_C	The processing speed of a CPU
α	Adjustment factor
J_{cpu}	A subset of 1st q jobs from job pool J
$Ceiling_{cpu}$	Upper boundary for the number of jobs in J_{cpu}, which are assigned to CPU
CJQ	CPU Job Queue
J_{update}	A subset of J, formed by subtracting CJQ from J
GJQ	GPU Job Queue
$CS_{gpu}(i)$	The computational share of ith GPU
$P_G(i)$	Processing speed of ith GPU
J_{gpu}	A subset of 1st r jobs from J_{update}
$Ceiling_{gpu}(i)$	The upper boundary for the number of jobs from J_{gpu}, which are assigned to ith GPU
Q	A family of set in which each element is a set containing elements from J

where, P_C represents *processing speed of a CPU* and α is the *adjustment factor* to balance the load across CPU and GPUs. The value of P_C is calculated using the processing speed formula given in Sect. 3.1. The value of α is obtained empirically, which is explained in Sect. 4.2.1.

From Eq. (3), to determine jobs that are assigned to *CJQ*, let $J_{cpu} = \{J_1, J_2, \ldots, J_q\}$ be a subset of 1st q jobs from job pool J. Here, $Ceiling_{cpu}$ is required to represent the upper boundary for number of jobs in J_{cpu} that are assigned to CPU. The value of $Ceiling_{cpu}$ is calculated as the sum of CS_{cpu} and 2/3rd of *computation requirement of* qth *job* (CRJ_q):

$$Ceiling_{cpu} = CS_{cpu} + \frac{2}{3} CRJ_q \tag{4}$$

where, the ratio 2/3 in Eq. (4) guarantees that qth job is mapped to CPU only if most of its computation (i.e., 2/3) falls within the limit of CS_{cpu}. Mapping the qth job to CPU (where 2/3rd part can be accommodated within the CPU share) cause a minor load imbalance as compared to mapping the job to a GPU.

Using Eq. (4), the number of jobs that are assigned to *CJQ* is given by:

$$\text{CJQ} = \begin{cases} J_{\text{cpu}} & \sum_{i=1}^{q} \text{CRJ}_i \leq \text{Ceiling}_{\text{cpu}} \\ J_{\text{cpu}} - J_q & \sum_{i=1}^{q} \text{CRJ}_i > \text{Ceiling}_{\text{cpu}} \end{cases} \tag{5}$$

After job assignment to *CJQ*, another set J_{update} is formed by subtracting *CJQ* from *J*:

$$J_{update} = J \backslash CJQ = \{job | job \in J \; and \; job \notin CJQ\} \tag{6}$$

Similarly, for job assignment to *GJQ*(*s*), Eq. (1) and Eq. (2) are used to calculate $CS_{gpu}(i)$, which represents the *computational share of* i*th GPU*. $CS_{gpu}(i)$ is calculated by the relationship given below:

$$CS_{gpu}(i) = \left(\frac{P_G(i)}{T_{CP}} \times J_{CR} \right) + \frac{\alpha}{g} \tag{7}$$

where, $P_G(i)$ represents *the processing speed of* i*th GPU* and *g* is the *number of GPUs* in the system. The $P_G(i)$ is calculated using the *processing speed* formula given in Sect. 3.1. From Eq. (7), to determine jobs that are assigned to *GJQ*(*i*), let $J_{gpu} = \{J_1, J_2, \ldots, J_r\}$ be a subset of 1st *r* jobs from J_{update}. Here, $Ceiling_{gpu}(i)$ represents the upper boundary for number of jobs from J_{gpu}, which are assigned to ith GPU. It is equal to the sum of $CS_{gpu}(i)$ and 2/3$^{\text{rd}}$ of *computation requirement of* r*th job* (CRJ_r) and is represented mathematically as:

$$Ceiling_{gpu}(i) = CS_{gpu}(i) + \frac{2}{3} CRJ_r \tag{8}$$

where the 2/3 CRJ_r represents a majority computing part of the rth job (similar to the explanation related to *q*th job in Eq. (4)).

Using Eq. (8), the number of jobs that are assigned to *GJQ*(*i*), for ith GPU is given by:

$$GJQ(i) = \begin{cases} J_{gpu} \sum_{i=1}^{r} CRJ_i \leq Ceiling_{gpu}(i) \\ J_{gpu} - J_r \sum_{i=1}^{r} CRJ_i > Ceiling_{gpu}(i) \\ J_{update} i == g \end{cases} \tag{9}$$

After job assignment to *GJQ*(*i*), the set J_{update} will be updated by subtracting *GJQ*(*i*) from J_{update}.

$$J_{update} = J_{update} \backslash GJQ(i) = \left\{ job | job \in J_{update} \, and \, job \notin GJQ(i) \right\} \tag{10}$$

After the completion of jobs assignment to CJQ and GJQ(s), a family of sets $Q = \{CJQ, GJQ_1, GJQ_2, \ldots, GJQ_g\}$ over J is obtained. The output set Q is governed by the constraints given in Eq. (11), Eq. (12), and Eq. (13).

$$\emptyset \notin Q \tag{11}$$

Equation (11) make sure that CPU job-set and GPU(s) job-sets in Q cannot be an empty set ϕ.

$$\cup_{A \in Q} A = J \tag{12}$$

Equation (12) specifies that union of all member set (of Q) is equal to job pool J.

$$(\forall A, B \in Q)\ A \neq B \Rightarrow A \cap B = \emptyset \tag{13}$$

Equation (13) stipulates that intersection for all non-equivalent member sets (A and B) of Q would be an empty set ϕ.

3.3 OSched algorithm

Algorithm 1 presents the detailed steps for the proposed scheduling mechanism OSched. First of all, J_{CR} and T_{CP} are initialized (Algorithm1, lines 1–2). Next, the values of J_{CR} and T_{CP} are calculated (lines 3–6). The value of α is calculated (line 7) in order to compute CS_{cpu} (line 8). Next, CJA_{cpu} the computation requirement of jobs assigned to CPU is initialized (line 9). Subsequently, the number of jobs which are assigned to CJQ are determined (lines 10–11). The value of J_{update} is calculated by subtracting CJQ from J. For job assignment to ith GPU, first $CS_{gpu}(i)$ is calculated (line 13a) $CJA_{gpu}(i)$, which is the computation requirement of jobs assigned to ith GPU. Next, CJA_{gpu} is initialized (see line 13b). After that, the number of jobs that are assigned to $GJQ(i)$ is determined (lines 13c–13d) and J_{update} is updated by subtracting $GJQ(i)$ from J_{update} (line 13e).

Algorithm-1: OSched job scheduling algorithm

INPUT:

i. A job pool *J* in which jobs are sorted in increasing order of the computation requirements
ii. List of CPU and GPU(s) in the system that is sorted in increasing order of processing speed

OUTPUT:

i. All jobs in job pool are assigned to CPU and GPU(s) for execution

```
1. INITIALIZE J_CR ← 0
2. INITIALIZE T_CP ← 0
3. FOR i = 1 to Total Jobs
        a. ADD J_CR += CRJ_i
4.  END FOR
5.  FOR i = 1 to Total Processors
        a. ADD T_CP += PS_i
6.  END FOR
7.  CALCULATE α = (4.12 / 100) × J_CR
8.  CALCULATE CS_cpu = ((P_c / T_CP) × J_CR) − α
9.  INITIALIZE CJA_cpu ← 0   /* CJA_cpu = Computation Requirements of Jobs that are assigned to CPU */
10. FOR i = 1 to q^th job from job pool
        a. ADD CJA_cpu += CRJ_i
        b. IF CJA_cpu ≤ CS_cpu + 2/3 (CRJ_i)
                i. ASSIGN CJQ ← J_i
11. END FOR
12. CALCULATE J_update = J − CJQ
13. FOR i = 1 to Total GPUs
        a. CALCULATE CS_gpu(i) = ((P_G(i) / T_CP) × J_CR) + (α / Total GPUs)
        b. INITIALIZE CJA_gpu(i) ← 0   /* CJA_gpu = Computation Requirements of Jobs that are assigned to
           i^th GPU */
        c. FOR j = 1 to r^th job from J_update
                i. CALCULATE CJA_gpu += CRJ_i
               ii. IF CJA_gpu ≤ CS_gpu(i) + 2/3 (CRJ_i)
                        1. ASSIGN GJQ(i) ← J_i
              iii. IF i = Total GPUs
                        1. ASSIGN GJQ(i) ← J_update
        d. END FOR
        e. SUBTRACT J_update −= GJQ(i)
14. END FOR
```

For the execution of an OpenCL application, the required data to be computed is first stored in main memory buffers. After that, these data buffers are transferred to the device memory (CPU or GPU memory), where all the computations to be performed. The multiple large data buffers lead to contention in main memory that slows down the data transfer (from main memory to the device memory). Moreover, the memory contention also slows down the execution of the OpenCL job on CPU device. Let's consider an example, where two OpenCL jobs i.e., *Matrix Multiplication* (MM) and *Discrete Cosine Transform* (DCT) (taken from the AMD benchmark suit ("APP SDK," n.d.)) are concurrently being executed (as shown in Fig. 4). Total available main memory on the machine is 08 GBs represented with memory footprint rectangle, where each sub-part of the rectangle represents 100MBs of capacity.

For MM application, each square matrix is of size $12{,}512 \times 12{,}512$ float elements (requiring approximately 600MBs of storage). Therefore, the storage space required for data is 1.8GBs with additional 1.8GBs for data-buffer. The memory requirements for the three matrices are represented as red-color filled area. The yellow-colored area shows the memory requirements for the DCT application (for an image size of $15{,}000 \times 15{,}000$) wherein each image buffer requires 900MBs of memory (total

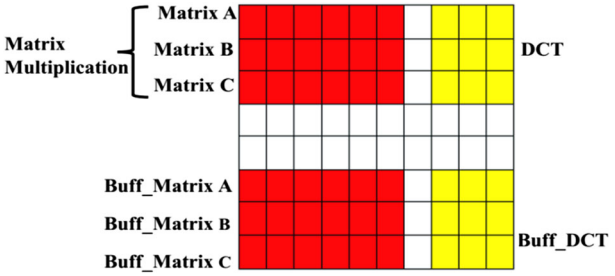

Fig. 4 Memory snapshot for concurrently executing jobs

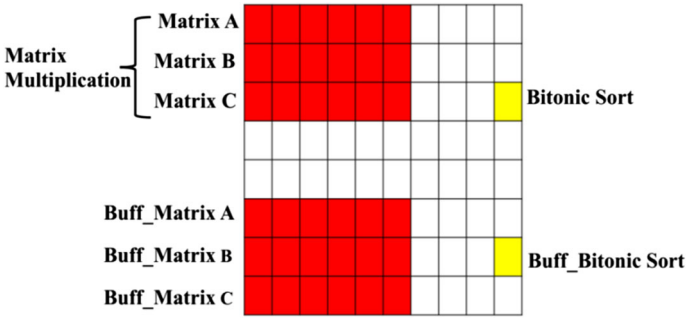

Fig. 5 Memory snapshot after optimization by the E-OSched

1.8GBs of memory). Additional to the execution of a large OpenCL application, the main memory also contains other operating systems services etc., which originate the issue of memory contention and degrade the execution performance of the application. To mitigate the memory contention problem, an extension to the OSched named as *Enhanced-OSched* (E-OSched) is proposed for scheduling data-parallel applications.

3.4 Enhanced-OSched

Enhanced-OSched (E-OSched) further refines the performance of the OSched by overcoming the memory contention problem (highlighted in the previous section). The E-OSched commences the execution of CPU job queue (CJQ) with the smaller job first and the GPU job queue (GJQ) with the larger job first. The induced job selection mechanism results in mitigating the memory contention problem. Figure 5 shows an example schedule (by the proposed E-OSched mechanism) wherein largest size job (i.e., Matrix Multiplication) from GJQ is scheduled along the smallest size job (i.e., Bitonic sort) from the CJQ. The employed scheduling heuristic is depicted in Fig. 3 (as a dotted rectangle labeled *E-OSched*).

Table 3 Experimental setup

Device	CPU	GPU
Model	Intel Core i5-4460	Nvidia GeForce GTX 760
Base Clock	3.2 GHz	0.980 GHz
Boost Clock	3.4 GHz	1.033 GHz
Total Cores	4	1152 (CUDA cores)
Memory	8 GB	2 GB
Processing Speed (Single Precision)	409.6 GFLOPS[a]	2257.9 GFLOPS
Operating System	Ubuntu 16.04 LTS	
OpenCL SDK	Intel SDK for OpenCL2016	CUDA 8.0
Compiler	GCC 5.4.0	Nvcc

[a]The processor contains 04 cores. Each core ticks 3.2×10^9 times per second and is capable of performing 32 floating point operations per cycle. Therefore, $4 \times 3.2 \times 10^9 \times 32 = 409.6$ GFLOPS

4 Experiments and results

The employed experimental setup consists of a heterogeneous multicore machine. The specifications of the employed machine are presented in Table 3.

For experimentations, 18 data-parallel applications from five different benchmark suites are used such as *AMD* [3], *Parboil* [18], *Polybench* [14], and *Rodinia* [9]. The benchmark applications (shown in Table 4) are executed using multiple problem sizes resulting in a job pool of total 182 jobs. All jobs in the job pool are independent and non-preemptive. Moreover, we consider that the processing cores within the processors (i.e., CPU and GPU) are homogeneous. All the experiments are conducted 05 times and mean of the results are reported.

4.1 Scheduling policies and evaluation metrics

For evaluation, we employ five scheduling techniques (listed below):

1. *All_On_CPU* all jobs are assigned to a CPU device for execution. It is a naïve heuristic that is considered as a baseline for performance comparison;
2. *All_On_GPU* all jobs are scheduled on a GPU device. This scheduling scheme indicates that programmers generally prefer GPUs only for execution that often leads to sub-optimal utilization of the other computing devices such as CPU;
3. *Work item guided* this heuristic [39] sorts jobs according to kernel's size of global work items. Then, CPU commences execution from a job with the lowest number of global work items whereas GPU starts execution with the highest number of global work items;
4. *Input size guided* In this scheme [39], jobs are sorted in ascending order in the task queue according to data (number of bytes) required to be transferred among a CPU and a GPUs. CPU starts execution of the task queue from the start of the

Table 4 Benchmarks along with input data sizes

Benchmark suites	Data-parallel applications	Input data size	Number of versions
AMD	Matrix multiplication	786,432–7,077,888	3
	Binomial options	32,768–131,072	3
	Bitonic sort	32,768–16,777,216	9
	Fast walsh transform	8192–204,800	14
	Matrix transpose	32,768–67,108,864	7
	Discrete cosine transformation	2,097,152–943,718,400	17
	Floyd Warshall	262,144–6,553,600	3
Polybench	3MM (3 matrix multiplications)	7,000,000–10,080,000	2
	GEMM(matrix-multiply C = alpha.A.B + beta.C)	750,000–12,000,000	5
	GESUMMV (scalar, vector and matrix multiplication)	8,012,000–1,800,180,000	17
	MVT (matrix vector product and transpose)	4,016,000–900,240,000	17
	ATAX (matrix transpose and vector multiplication)	4,012,000–900,180,000	17
	2MM (2 matrix multiplications)	5,000,000–12,800,000	2
	2DCONV (2D convolution kernel)	2,000,000–1,568,000,000	17
	3DCONV (3D convolution kernel)	1,000,000–1,728,000,000	17
Parboil	3D stencil operations	2,097,152–67,108,864	2
Rodinia	BFS (breadth first search)	43,963–592,428,022	14
Own developed	Matrix–vector multiplication	4,202,496–1,514,299,392	16

sorted queue (smallest job first) while GPU computes tasks from the end of the sorted task queue (largest jobs first);

5. *Machine Learning-based Task* (*MLT*) *scheduling* The heuristic presented by [39], is based on machine learning-based classification that employs the static code features (such as *instructions*, *math functions, barriers* etc.) and dynamic run-time features (such as *local_work_size, global_work_size, input data size* etc.) to determine device suitability of OpenCL kernels.

Table 5 presents the cost and latency analysis of all the scheduling heuristics that have been used for the sake of comparison. Among all the employed scheduling heuristics, the MLT technique [39] incurs the highest overhead. Prior to the scheduling, the

Table 5 Cost and latency analysis of the scheduling heuristics

Scheduling heuristics	Require pre-processing (for N jobs)	Offline training	Time complexity (scheduling only)	Scheduling latency ratio (for $N = 1000$ jobs)
OSched	Computational requirements	–	$O(N^2)$	624 ×
E-OSched	Computational requirements	–	$O(N^2)$	697 ×
All_On_CPU	–	–	$O(N)$	Baseline
All_On_GPU	–	–	$O(N)$	Baseline
Work item guided	Work-item size	–	$O(N^2)$	516 ×
Input size guided	Input data size	–	$O(N^2)$	507 ×
MLT scheduling	(a) Feature extraction (b) Device suitability prediction	Training phase for device suitability predictor	$O(m^2 + n^2)$	939 ×

MLT heuristic requires two major pre-processing steps, i.e., *code features extraction* and *device suitability prediction* (for each job). Moreover, the MLT scheduling technique requires an offline training of the machine learning-based model to predict device suitability. The scheduling complexity of the MLT is $O(m^2 + n^2)$ and the scheduling latency is 939 × as compared to the baseline scheduling heuristics (i.e., *ALL_On_GPU* and *ALL_On_CPU*).

The time complexity of the *OSched*, *E-OSched*, *Work item guided,* and *Input size guided*-based scheduling heuristics is $O(N^2)$, as all of these scheduling heuristics employ a sorting step (for N jobs) in their scheduling mechanism that requires a higher number of computational steps (i.e., $O(N^2)$). The OSched and E-OSched both require a pre-processing step for computing the *jobs computational requirements*. The Work-item and Input size guided heuristics require a pre-processing step of *work-item count* and *size of input data* for each job, respectively. The baseline scheduling schemes (i.e., *All_On_CPU* and *All_On_GPU*) do not require any pre-processing steps. All the other heuristics except the MLT-based scheduling do not require an offline training of the machine learning model. Table 5 presents the latency ratio (of the *scheduling-step* only) for all the employed scheduling heuristics calculated using the baseline heuristics (i.e., *All_On_CPU*, and *All_On_GPU*) which involve a trivial latency overhead. The scheduling latencies of the OSched, E-OSched, Work-item guided, Input size guided, and MLT-based heuristics are observed 624 ×, 697 ×, 516 ×, 507 ×, and 939 × higher as compared to the baseline scheduling heuristics (having a negligible scheduling latency), respectively.

For evaluation, we consider the following performance metrics:

1. *Execution time* depicts the time consumed in the execution of all jobs of the job pool. The smaller value of the execution time is an indication of better results;

2. *Throughput* represents the number of jobs completed per unit time. The higher value of throughput manifest the better results;
3. *Average time* (*of a job*) is defined as an average amount of time taken by a job (of a job pool) to complete its execution. The lower average time exhibits the improved performance;
4. *Load balance* measures the distribution of workload among CPU and GPUs in the form that the employed computing devices accomplish the execution of the assigned workload within the approximately same time duration. We calculate load balance as a percentage of the difference between execution times of all the jobs mapped on a CPU and GPU devices. The lower value of this metric shows a more load-balanced execution.

4.2 Execution time analysis

The execution time of each scheduling heuristic is recorded with varying number of CPU cores (i.e., 1, 2, and 4 cores) and GPU to analyze machine size impact on the execution time. Moreover, analysis of adjustment factor α is also presented in this section.

Figure 6 presents execution time (using 4 CPU cores and the GPU device) of the proposed heuristic along with the heuristics found in the literature. These results specify that the OSched and the E-OSched have outperformed all the other scheduling heuristics in terms of the execution time of job pool (mentioned in Sect. 4.1). As compared to the baseline scheduling (i.e., *All_On_CPU*), the *OSched* and the E-OSched consumes 2.03 × and 2.01 × lower execution time, respectively. For the GPU-based execution of the job pool (i.e., *All_On_GPU*), the *OSched,* and *E-OSched* consume 1.70 × and 1.71 × reduced execution time. The *E-OSched* consumes 6.25% reduced execution time as compared to the *input size guided* scheduling heuristic [39] (as shown in Fig. 6). As compared to the *input size guided* heuristic [39], the proposed *E-OSched* consumes 7.35% reduced execution time. As compared to the *work-item guided* scheduling heuristic [39], the *OSched* and *E-OSched* consume 5.16 and 6.25% reduced execution time, respectively. The *OSched* consumes 2.54% less execution time for the job-pool execution as compared to the scheduling heuristic of *MLT* [39]. Moreover, the *E-OSched* further improves the execution time and results in 3.51% reduced execution time as compared to the *MLT* heuristic [39].

Figure 7 presents the execution time (using 02 CPU cores and the GPU device) for the proposed and the other scheduling heuristics. As compared to the All_On_CPU and All_On_GPU scheduling schemes, the *OSched* reduces execution time by 2.05 × and 1.6 ×, respectively. The *E-OSched* consumes 2.1 × and 1.7 × lower execution time as compared to the baseline scheduling heuristics (i.e., All_On_CPU and All_On_GPU), respectively. The *OSched* and the *E-OSched* consume 1.07 × and 1.09 × reduced execution time, respectively, as compared to the *input size guided* scheduling heuristic [39]. As compared to the *work-item guided* scheduling heuristic [39], the *OSched* and *E-OSched* consume 1.07 × and 1.05 × reduced execution time, respectively. As compared to the state-of-the-art scheduling scheme *MLT* [39], the reduction in execution time by the *OSched* and the *E-OSched* is 1.04 × and 1.02 ×, respectively.

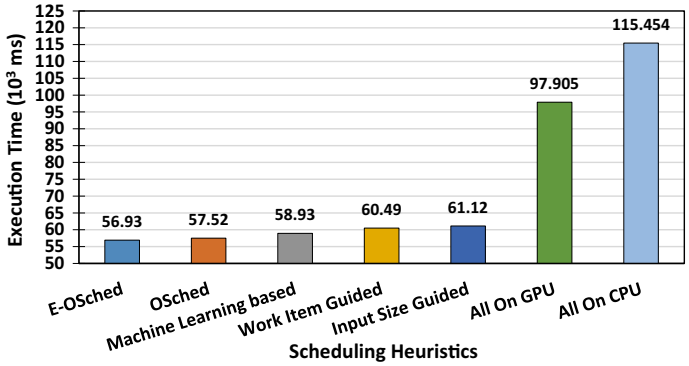

Fig. 6 Execution time—04 CPU cores and a GPU device

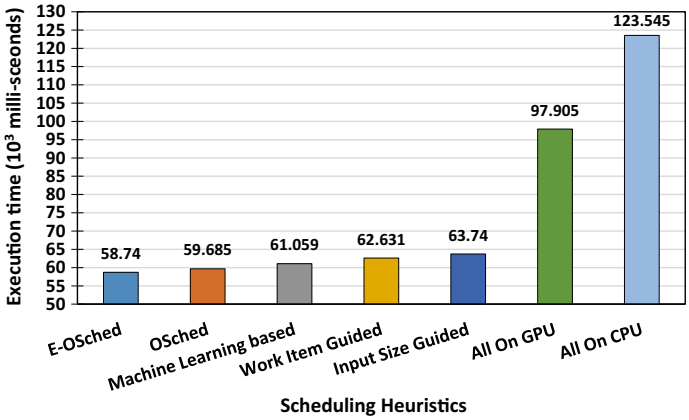

Fig. 7 Execution time—02 CPU cores and a GPU device

Next, the execution time results (using 01 CPU core and the GPU device) for the proposed and the other scheduling heuristics are presented in Fig. 8. The results show that *All_On_CPU* and *All_On_GPU* scheduling schemes are outperformed by the *OSched* by, respectively, consuming 1.6 × and 1.4 × less execution time. Similarly, the *E-OSched* reduces execution time by 1.64 × and 1.17 ×, respectively, when compared to *All_On_CPU* and *All_On_GPU* scheduling schemes. The *OSched* and the *E-OSched* reduce execution time by 1.1 × and 1.13 ×, respectively, as compared to the *input size guided* scheduling heuristic [39]. Reduction in execution time (by the OSched and E-OSched) is observed 1.11 × and 1.13 ×, respectively, when compared to the *work-item guided* scheduling heuristic [39]. The *OSched* and the *E-OSched*, when compared to the *MLT* [39] scheduling scheme, respectively, consumes 1.05 × and 1.08 × less execution time.

The execution time-based results (Figs. 6, 7, 8) show that the *OSched* and the *E-OSched* persistently produce better results (in terms of execution time) as compared to the other employed scheduling heuristics. The reduced execution time consumed by the proposed *OSched* and *E-OSched* scheduling heuristics are due to the load-

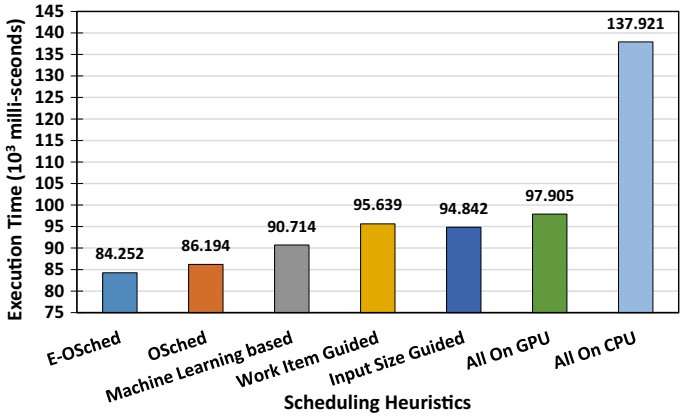

Fig. 8 Execution time—01 CPU core and a GPU device

balanced execution of the job pool (as evident in the further results in this section). An interesting phenomenon observed during the experimentation is that when the number of CPU cores is reduced from 4 to 2, the execution time of all the employed scheduling schemes increases slightly (3–7% higher). However, when CPU cores are further reduced to 1, a significant increase in terms of the execution time is noticed (up to 53%). The increased execution time (for 2 CPU cores) results due to the less number of available CPU cores and the majority of the execution load is mapped on the GPU device. However, when the CPU cores are reduced to only 1, the execution time of the job pool increases significantly as the only CPU core is now responsible to execute the assigned workload (according to its processing capability) in addition to the execution of the host programs (of all the concurrently executing applications). As shown in Figs. 6, 7 and 8, the execution time results for *All_On_GPU* scheduling scheme remain almost similar with varying CPU cores. The reason for the similar attained execution time (by the *All_On_GPU* scheduling scheme) is that it only utilizes GPU device for execution and no job-pool-related load is mapped on CPU device.

4.2.1 Analysis of adjustment factor "α"

The host program of an OpenCL application (mapped either on a CPU or a GPU) is always executed on a CPU device. The execution of the host programs on a CPU device causes a certain execution overhead during the concurrent executions of OpenCL applications. Due to this overhead, the CPU-based execution of a kernel often faces increased execution time. Figure 9 shows an execution profile of different OpenCL programs mapped on a GPU device. It is evident from Fig. 9 that the CPU has to spend a non-negligible time in the execution of the host program. An *adjustment factor* α overcomes this overhead (i.e., the induced load imbalance) by off-loading the computations from CPU to GPU device. The value of α depends on the mean time spent by the CPU for the execution of the host programs of a job pool. The value of α is calculated by considering the mean execution time (of host programs only) of

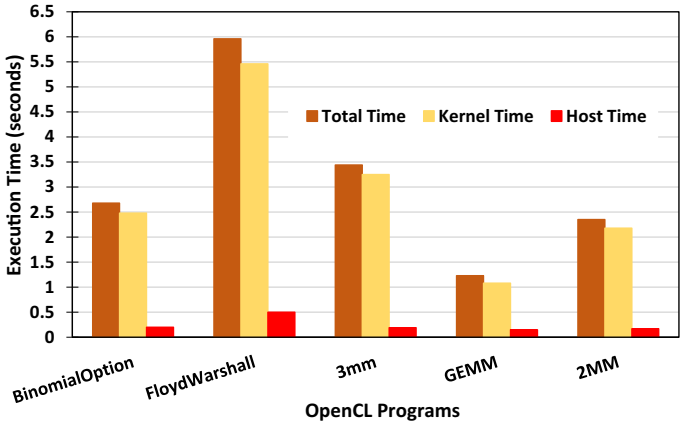

Fig. 9 Application execution profile

different OpenCL applications in the job pool. Therefore, a factor α with 4.12% of J_{CR} is introduced for the proposed scheduling heuristic (as shown in line 7 of Algorithm 1). To adjust the host program execution overhead (on a CPU device), α percentage of the computational load is deducted from the CPU's share and added to the GPU that results in near optima load balance.

4.3 Average execution time analysis

In this section, average execution time of a job (using 04 CPU cores and a GPU) are reported. Figure 10 shows the average execution time of a job for seven scheduling heuristics. As compared to the *All_On_CPU* scheduling, the proposed *OSched* and *E-OSched* result in 101.27 and 103.88% reduced execution time (for a single job) on average, respectively. The *OSched* and *E-OSched* consume on average 70.92 and 73.13% reduced execution time as compared to the *All_On_GPU* scheduling, respectively. As compared to the *input size guided* heuristic [39], the *OSched* and the *E-OSched* consume on average 6.7 and 8.09% reduced execution time, respectively. The *OSched* and the *E-OSched* consume on average 5.75 and 7.11% reduced execution time as compared to the *work-item guided* [39] scheduling, respectively. As compared to the MLT [39] scheduling, the *OSched* and the *E-OSched* consume on average 3.19 and 4.53% reduced execution time, respectively.

4.4 Throughput analysis

For throughput analysis, the scheduling heuristic of *MLT* proposed by [39] is considered as a baseline. Figure 11 presents the attained throughput of the schedulers (using 04 CPU cores and a GPU device) as compared to the baseline heuristic [39]. The *E-OSched* and *OSched* achieve the highest throughput as compared to the baseline (i.e., 2.45 and 3.51% improved throughput, respectively) and the other scheduling heuris-

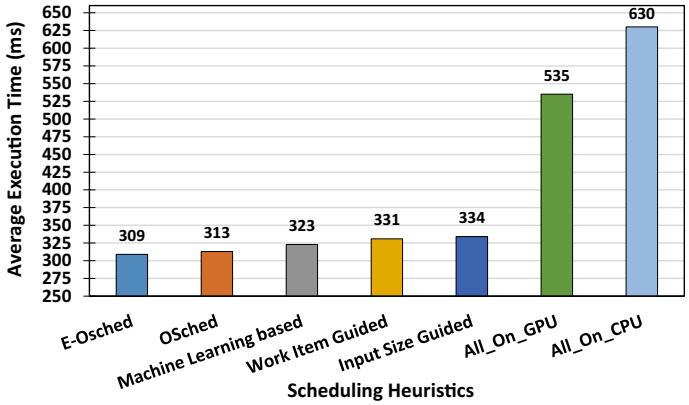

Fig. 10 Average execution time of a job in the job pool

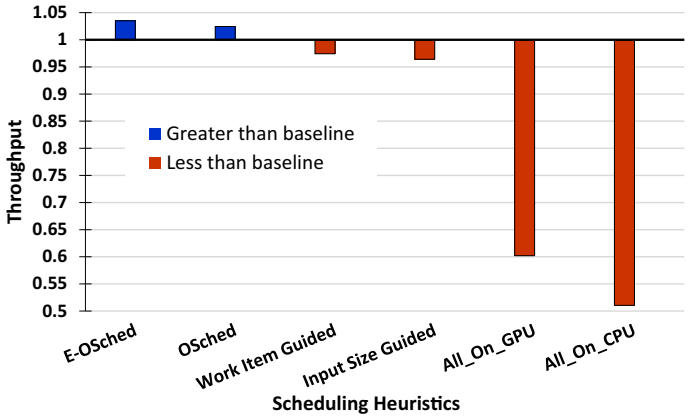

Fig. 11 Throughput analysis

tics. The proposed schedulers *OSched* and *E-OSched* attain 5.02 and 6.08% higher throughput as compared to the *work-item guided* heuristic [39], respectively. As compared to the *input size guided* scheduling [39], the *OSched* and the *E-OSched* attain 6.03 and 7.07% improved throughput, respectively. The *OSched* and the *E-OSched* attain 42.25 and 43.31% higher throughput as compared to the *All On GPU* scheduling, respectively. As compared to the *All On CPU* scheduling, the *OSched* and the *E-OSched* attain 51.4 and 52.46% improved throughput, respectively.

4.5 Load balance analysis

To demonstrate the effectiveness of factor α, we present load balance analysis with and without adjusting α. Figure 12 presents the load balance achieved by the proposed schedulers OSched and E-OSched without adjusting factor α. With the adjusted α, the load balance attained by the OSched and E-OSched is presented in Fig. 13. The

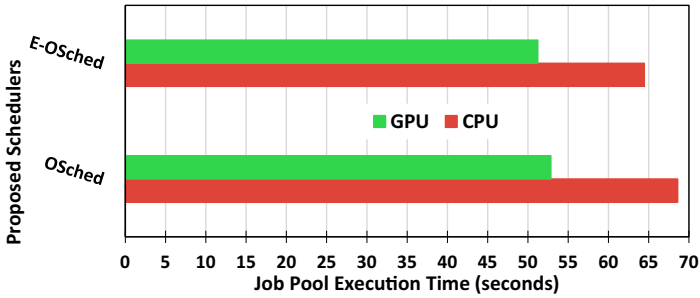

Fig. 12 Load balance without factor α

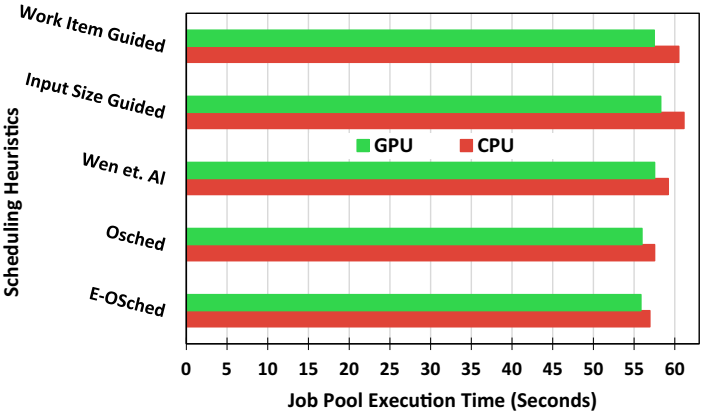

Fig. 13 Load balance after adjusting factor α (in OSched and E-OSched)

results presented in Figs. 12 and 13 are obtained using 04 CPU cores and a GPU device. Figure 12 shows that the load imbalance is 23 and 20.5% for the *OSched* and the *E-OSched,* respectively (without adjusting overhead factor α). After the introduction of α (Fig. 13), load imbalance decreases to 2.7 and 1.9% for the *OSched* and the *E-OSched,* respectively. These results (with the adjusted factor α) show a significant decrease in execution time of job pool. When compared to the load imbalance induced by *work-item guided* [39] scheduling heuristic, the *OSched* and the *E-OSched* reduce load imbalance by 2.3 and 3.1%, respectively. The *OSched* and the *E-OSched* reduce load imbalance by 1.97 and 2.68% as compared to load imbalance induced by *input size guided* [39] scheduling heuristic, respectively. As compared to MLT [39], the load imbalance reduced by the *OSched* and *E-OSched* is observed 0.15 and 0.95%, respectively.

5 Conclusions

This work focuses on two novel scheduling schemes named OSched and E-OSched to ensure the load balancing of compute-intensive applications on heterogeneous multicores. These schedulers perform the resource-aware assignment of jobs while

contemplating the job requirements and processing capabilities of the employed computing devices (i.e., CPUs and GPUs). The performance evaluation experiments have revealed that OSched has significantly improved the load balancing on the employed computing devices, minimized the job-pool make-span, maximized throughput, and maximized the resource utilization as compared to the baseline (i.e., All_On_CPU and All_On_GPU) and the state-of-the-art scheduling heuristics. Moreover, the E-OSched has further enhanced the execution performance by incorporating the memory contention factor. The experimental evaluation shows that the execution time is reduced by 70.92 and 73.13% for OSched and E-OSched, respectively, as compared to the baseline scheduling heuristics. As compared to the state-of-the-art scheduling heuristics, the OSched and E-OSched have reduced the execution time by 6.7 and 8.09%, respectively. The OSched and E-OSched both have improved throughput by 51.4 and 52.46% as compared to the baseline heuristics, respectively. As compared to the state-of-the-art scheduling heuristics, OSched and E-OSched have achieved up to 6.03 and 7.07% higher throughput, respectively. As most of the today's large compute-clusters and supercomputers are based on heterogeneous computing nodes[3] that require a resource-aware load-balanced scheduling mechanism to map jobs across a variety of the computing devices. The OSched and E-OSched will immensely aid to schedule compute-intensive data-parallel jobs in a load-balanced manner to reduce job-pool execution time, to improve system throughput, and to increase device utilization. In future, we intend to extend the E-OSched scheduler for CPU/GPU cluster based on multi-machine configurations.

Acknowledgements The Austrian Promotion Agency (FFG) partially funded this work as part of the project 848448 "Tiroler Cloud".

References

1. Albayrak OE, Akturk I, Ozturk O (2012) Effective kernel mapping for OpenCL applications in heterogeneous platforms. In: Proceedings of International Conference on Parallel Processing Work, pp 81–88. https://doi.org/10.1109/ICPPW.2012.14
2. Aleem M, Prodan R, Fahringer T (2011) Scheduling javasymphony applications on many-core parallel computers. In: Euro-Par 2011 Parallel Processing. Springer, pp 167–179
3. APP SDK [WWW Document], n.d. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/. Accessed 1 May 2017
4. Augonnet C, Thibault S, Namyst R, Wacrenier P-A, Wacrenier StarPU P-A (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput Pract Exp 23:187–198
5. Becchi M, Byna S, Cadambi S, Chakradhar S (2010) Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In: Proceedings of 22nd ACM Symposium Parallelism algorithms Architecture, pp 82–91. https://doi.org/10.1145/1810479.1810498
6. Belviranli ME, Bhuyan LN, Gupta R (2013) A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans Archit Code Optim 9:1–20. https://doi.org/10.1145/2400682.2400716
7. Binotto APD, Pereira CE, Kuijper A, Stork A, Fellner DW (2011) An effective dynamic scheduling runtime and tuning system for heterogeneous multi and many-core desktop platforms. In: 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC). IEEE, pp 78–85

[3] https://www.top500.org/lists/2017/11/.

8. Boyer M, Skadron K, Che S, Jayasena N (2013) Load balancing in a changing world: dealing with heterogeneity and performance variability. In: Proceedings of the ACM International Conference on Computing Frontiers. ACM, p 21
9. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: IISWC 2009. IEEE International Symposium on Workload Characterization, 2009. IEEE, pp 44–54
10. Chen Z, Marculescu D (2017) Task scheduling for heterogeneous multicore systems. arXiv Prepr. arXiv1712.03209
11. Choi HJ, Son DO, Kang SG, Kim JM, Lee H-H, Kim CH (2013) An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. J. Supercomput 65:886–902. https://doi.org/10.1007/s11227-013-0870-6
12. Dolbeau R (2018) Theoretical peak FLOPS per instruction set: a tutorial. J Supercomput 74:1341–1377. https://doi.org/10.1007/s11227-017-2177-5
13. Ghose A, Dey S, Mitra P, Chaudhuri M (2016) Divergence aware automated partitioning of OpenCL workloads. In: Proceedings of the 9th India Software Engineering Conference. ACM, pp 131–135. https://doi.org/10.1145/2856636.2856639
14. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: Innovative Parallel Computing (InPar). IEEE, pp 1–10
15. Gregg C, Boyer M, Hazelwood K, Skadron K (2011) Dynamic heterogeneous scheduling decisions using historical runtime data. In: Proceedings of the 2nd Workshop on Applications for Multi-and Many-Core Processors. San Jose, CA
16. Gregg C, Brantley JS, Hazelwood K (2010) Contention-aware scheduling of parallel code for heterogeneous systems. In: 2nd USENIX Workshop on Hot Topics Parallelism
17. Grewe D, O'Boyle MF (2011) A static task partitioning approach for heterogeneous systems using OpenCL. In: International Conference on Compiler Construction. Springer, pp 286–305
18. IMPACT Research Group and others (2007) IMPACT: parboil benchmarks [WWW Document]. http://impact.crhc.illinois.edu/parboil/parboil.aspx. Accessed 1 May 2017
19. Insieme Compiler Project [WWW Document], n.d. http://www.insieme-compiler.org/. Accessed 9 July 2017
20. Jiménez VJ, Vilanova L, Gelado I, Gil M, Fursin G, Navarro N (2009) Predictive runtime code scheduling for heterogeneous architectures. In: International Conference on High-Performance Embedded Architectures and Compilers. Springer Berlin Heidelberg, pp 19–33
21. Kaleem R, Barik R, Shpeisman T, Lewis BT, Hu C, Pingali K (2014) Adaptive heterogeneous scheduling for integrated GPUs. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. ACM, pp 151–162
22. Kofler K, Grasso I, Cosenza B, Fahringer T (2013) An automatic input-sensitive approach for heterogeneous task partitioning categories and subject descriptors. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing—ICS'13. pp 149–160. https://doi.org/10.1145/2464996.2465007
23. Lee J, Samadi M, Mahlke S (2015a) Orchestrating multiple data-parallel kernels on multiple devices. In: 2015 International Conference on Parallel Architecture and Compilation (PACT). IEEE, pp 355–366
24. Lee J, Samadi M, Park Y, Mahlke S (2015) Skmd: single kernel on multiple devices for transparent cpu-gpu collaboration. ACM Trans Comput Syst 33:1–27. https://doi.org/10.1145/2798725
25. Lee J, Samadi M, Park Y, Mahlke S (2013) Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. IEEE Press, pp 245–256
26. Lösch A, Beisel T, Kenter T, Plessl C, Platzner M (2016) Performance-centric scheduling with task migration for a heterogeneous compute node in the data center. In: Proceedings of the 2016 Conference on Design, Automation and Test in Europe. EDA Consortium, pp 912–917
27. Luk C-K, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 45–55
28. Munshi A (2009) The OpenCL specification. In: 2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, pp 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342
29. OpenCL—The open standard for parallel programming of heterogeneous systems [WWW Document], n.d. https://www.khronos.org/opencl/. Accessed 1 Mar 17

30. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. Proc IEEE 96:879–899. https://doi.org/10.1109/JPROC.2008.917757
31. Pandit P, Govindarajan R (2014) Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM, p 273. https://doi.org/10.1145/2544137.2544163
32. Ravi VT, Agrawal G (2011) A dynamic scheduling framework for emerging heterogeneous systems. In: 18th International Conference on High Performance Computing, HiPC 2011. IEEE, pp 1–10. https://doi.org/10.1109/HiPC.2011.6152724
33. Rohr D, Kalcher S, Bach M, Alaqeeliy AA, Alzaidy HM, Eschweiler D, Lindenstruth V, Alkhereyfy SB, Alharthiy A, Almubaraky A, Alqwaizy I, Suliman RB (2014) An energy-efficient multi-GPU supercomputer. In: 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and Systems (HPCC, CSS, ICESS). IEEE, Paris, pp 42–45. https://doi.org/10.1109/HPCC.2014.14
34. Rul S, Vandierendonck H, D'haene J, De Bosschere K (2010) An experimental study on performance portability of OpenCL kernels. Papers presented at the 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC '10)
35. Samsung Galaxy S8+—Full phone specifications [WWW Document], n.d. http://www.gsmarena.com/samsung_galaxy_s8+-8523.php. Accessed 7 Oct 2017
36. Sun E, Schaa D, Bagley R, Rubin N, Kaeli D (2012) Enabling task-level scheduling on heterogeneous platforms *. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units. ACM, pp 84–93
37. Wang Z, Zheng L, Chen Q, Guo M (2013) CAP: co-scheduling based on asymptotic profiling in CPU + GPU hybrid systems. In: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores—PMAM'13. ACM, pp 107–114. https://doi.org/10.1145/2442992.2443004
38. Wen Y, O'Boyle MF (2017) Merge or separate? Multi-job scheduling for OpenCL kernels on CPU/GPU platforms. In: Proceedings of the General Purpose GPUs. ACM, pp 22–31. https://doi.org/10.1145/3038228.3038235
39. Wen Y, Wang Z, O'boyle MFP (2014) Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In: 2014 21st International Conference on High Performance Computing (HiPC). IEEE, pp 1–10
40. Yan X, Shi X, Wang L, Yang H (2014) An OpenCL micro-benchmark suite for GPUs and CPUs. J Supercomput 69:693–713. https://doi.org/10.1007/s11227-014-1112-2