




Article

Resource-Aware Device Allocation of Data-Parallel Applications on Heterogeneous Systems

Donghyeon Kim ¹, Seokwon Kang ¹, Junsu Lim ¹, Sunwook Jung ¹ and Woosung Kim ²
and Yongjun Park ^{1,*}

¹ Department of Computer Science, Hanyang University, Seoul 04763, Korea; donghyeon.kim9309@gmail.com (D.K.); kswon0202@gmail.com (S.K.); vfv0121@gmail.com (J.L.); metaljsw2@naver.com (S.J.)

² LG Electronics CTO Software Platform Laboratory, Seoul 06772, Korea; rain.kim@lge.com

* Correspondence: yongjunpark@hanyang.ac.kr

Received: 7 October 2020; Accepted: 30 October 2020; Published: 2 November 2020



Abstract: As recent heterogeneous systems comprise multi-core CPUs and multiple GPUs, efficient allocation of multiple data-parallel applications has become a primary goal to achieve both maximum total performance and efficiency. However, the efficient orchestration of multiple applications is highly challenging because a detailed runtime status such as expected remaining time and available memory size of each computing device is hidden. To solve these problems, we propose a dynamic data-parallel application allocation framework called ADAMS. Evaluations show that our framework improves the average total execution device time by 1.85× over the round-robin policy in the non-shared-memory system with small data set.

Keywords: device abstraction; dynamic resource management; GPGPUs; heterogeneous system architecture; multitasking; OpenCL

1. Introduction

High performance and energy efficiency are critical parameters for emerging applications such as vision and various machine-learning applications [1–5]. Heterogeneous system architectures (HSAs), which generally comprise multi-core central processing units (CPUs), graphic processing units (GPUs), and other accelerators, are the de-facto solutions for the data-parallel applications, and they can be efficiently used through a single-instruction multiple-thread (SIMT) model (CUDA [6], OpenCL [7]). Although HSAs comprise numerous computing devices, they often experience substantial performance degradation when massive data-parallel job requests are received simultaneously. For example, recent cloud servers are often allocated to multiple users; the users launch their own data-parallel tasks, as shown in Figure 1. In this situation, load balancing failures among multiple devices result in resource underuse of some devices, and therefore, the maximum performance of the system cannot be achieved. Even with proper load balancing, the total system performance can be degraded when memory problems such as frequent memory allocation failures and paging processes occur owing to the large total memory requirement from multiple processes concurrently. Therefore, efficient allocation of multiple workloads onto multiple devices is the most important challenge for maximizing performance.

To meet this requirement, several new techniques (SnuCL [8], VirtCL [9]) have been recently introduced by focusing mainly on the efficient execution of multiple kernels of a single application across multiple devices. However, load balancing on HSAs is not limited to the multi-kernel level. As an HSA is currently used as the main system to support multiple applications, several OpenCL applications often exist concurrently. Thus, an improved multi-application management approach,

beyond the simple kernel-level management, is necessary to effectively use all the available OpenCL devices. To support this, runtime status of all the applications and devices should be monitored.

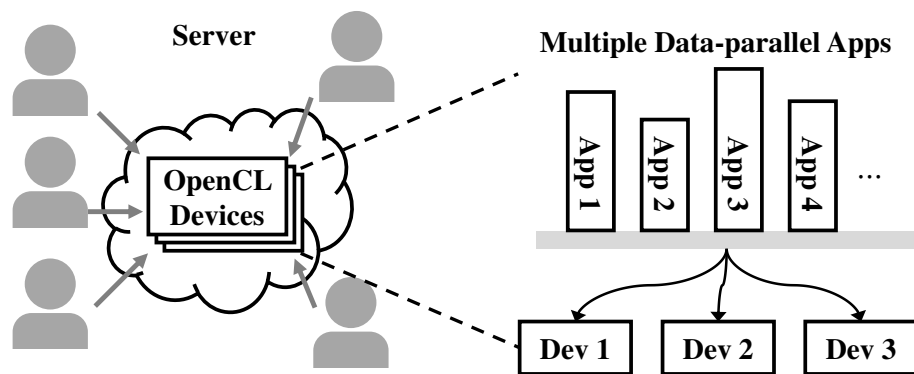


Figure 1. An example system where multiple users execute their own data-parallel tasks.

The first well-known challenge for efficient orchestration of multiple applications on multiple devices is determining the best target device of each running application to maximize total performance during concurrent execution. To achieve this, newly launched application's target device should have a minimum estimated completion time among the multiple available devices by considering both its expected execution time and the remaining execution time of the currently running applications on the device. Although the execution time prediction of data-parallel programs seems impossible, recent studies have shown that execution time of general-purpose GPU (GPGPU) tasks are fairly predictable [9–11] using simple regression models based on offline profiling. Therefore, total performance optimization based on the execution time estimation can be realized with predetermined input applications and prepared profile data.

More critical problem is the potential performance degradation when the device memory is not well used for executing multiple applications on a single device. When an application's memory allocation remains incomplete because other applications are using a substantial amount of the memory, the application cannot proceed until some amount of memory from the other applications is deallocated. In a system where memory is shared between CPUs and GPUs, such as the Intel i-series [12–14] and AMD Fusion [15], although memory allocation failures are rare as memory space is shared between the host CPUs and OpenCL devices, the impact of the memory insufficiency problem on performance is still critical. In such systems, severe performance degradation occurs when significantly large memory spaces are allocated to OpenCL kernels owing to the high memory-swapping overhead and host program slowdown resulting from memory insufficiency.

Recent GPU evolution trends (NVIDIA Pascal [16] and Volta [17]) improve both the throughput and latency by allowing concurrent execution of multiple kernels using preemptive [18]/spatial multitasking [19]. These trends further intensify memory problems. Therefore, efficient memory management has become crucial for achieving better multitasking performance because concurrent kernel execution requires more memory to handle all co-running kernels.

To address this issue, we introduce an automatic device allocation management system (ADAMS). It assigns multiple OpenCL applications onto appropriate devices to maximize device use by avoiding memory allocation failures. For this, an ADAMS selects a device with minimum estimated completion time and without any memory limitation problem. ADAMS comprises multiple APIs that can easily be added to the original OpenCL programs and necessary information in the operating system (OS) shared memory includes the allocated application list and remaining total execution time per device.

2. Background and Motivation

2.1. OpenCL Programming Model

OpenCL kernels consist of multiple thread blocks (TBs), and each TB consists of multiple threads. A thread is the basic unit of execution in a SIMT model. Kernels are executed on accelerators at the TB level. A typical OpenCL program first selects a target device to execute and then, initializes the kernels for the selected device as shown in Figure 2a. Furthermore, the program transfers the input data from the host to the target device prior to kernel execution. Next, it reads the result from the device. Finally, the program releases the allocated resources for the kernel. These procedures are almost identical for all the target devices; therefore, the target device can be dynamically selected with additional application programming interface (API) support.

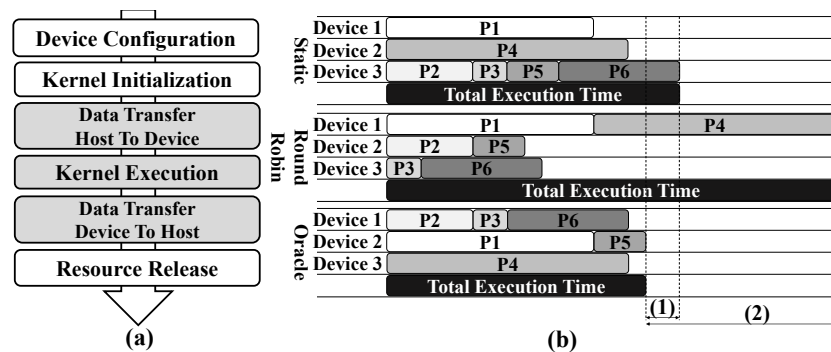


Figure 2. (a) A typical OpenCL program flow and (b) examples of execution scenarios with various allocation methods.

2.2. Target Device-Selection Challenge

The target device of a typical OpenCL program is defined statically. However, in multi-application domains, target device selection must be dynamic because static selection frequently leads to low performance, as it does not consider the dynamic status of the target device. The simplest dynamic target device selection method is the round-robin (RR) policy that functions based on launch order. However, this can also cause frequent performance degradation and high execution time variation owing to load balancing failures. Figure 2b shows the performance variation for different allocation methods. *Static* is a device allocation example when an experienced programmer only knows that P1 and P4 are the longest processes for a total of six processes on three target devices, and *Oracle* is the optimal configuration. In this example, lack of dynamic information leads to lower performance (Figure 2b(1)) despite the programmer having deep understanding of the applications. The worst scenario of the RR policy occurs when P1 and P4 are assigned to the same device, where the performance is worse than that of *Static*. Therefore, improved runtime device allocation policy considering dynamic information is essential.

2.3. Execution Time Prediction of Data-Parallel Applications

To schedule multiple applications on multiple devices, the framework must be aware of each application’s execution time on each device. Though the execution time estimation is nearly impossible for general programs, recent studies have shown that the execution time of GPGPU tasks are fairly predictable [9–11] based on input problem sizes; therefore, we decided to use a problem size-based regression model for execution time prediction, similar to the approach of MKMD [10] based on offline profile data.

In this study, as we admit that execution time prediction for random programs is impossible, the input data range for input applications are assumed to be known with offline profiling, which are social network service (SNS) analysis or machine-learning applications consisting of matrix

multiplication and convolution computations. For fair execution time prediction of the applications, we profiled the execution time of the target applications with various problem sizes from minimum to maximum and constructed their prediction polynomial models considering both data transfers and kernel executions. In addition, conservative misprediction error-handling mechanism was also implemented to tolerate the error situations.

2.4. Memory Limitations

2.4.1. Limitations in Non-Shared-Memory Systems

Memory allocation failure can occur during execution if the total memory requirement for running multiple applications is larger than the GPU memory size. Therefore, to ensure performance improvement while executing multiple applications on a device, the total global memory usage of the device should be managed carefully. Figure 3 shows the average error occurrence of memory allocation for traditional queue-based (QB) and concurrent multi-application executions. As shown in the figure, more than three errors, on an average, are detected in both the cases, and the average error rate for the concurrent case is much higher than that of the QB. Although smart error handling can be implemented, most existing OpenCL programs are terminated when an error is detected, and re-execution is required with high performance overhead. Therefore, accurate memory usage estimation before allocation is essential to avoid memory allocation failures.

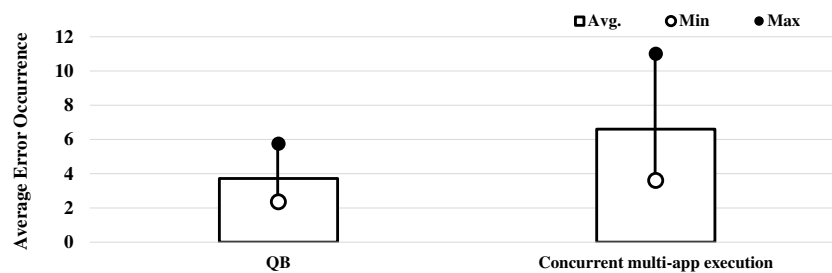


Figure 3. Memory allocation failure occurrence in the Non-Shared-Memory System.

2.4.2. Limitations in Shared-Memory Systems

In a system where multiple computing devices share a single memory, memory allocation to one device affects the rest. In shared-memory systems, an OpenCL program cannot run on the other device when the memory capacity of the originally selected device is insufficient because the devices share the memory space. Figure 4 shows the average total execution time of multiple 2DConvolution benchmarks [20] when the number of processes increases. The X-axis denotes the number of sets used. All the processes execute an OpenCL kernel that needs approximately 1 GB of memory objects on an on-chip Intel HD Graphics GPU, concurrently on a baseline system with a shared 16 GB memory. As shown in the figure, the performance degrades considerably when more than 12 sets of 2DConvolution are concurrently executed because the total memory usage exceeds 16 GB, and it is handled by memory swapping with high paging overhead, without memory allocation errors. To solve this issue, the system needs to control the number of concurrently running processes so that the memory limit is not exceeded. Therefore, memory usage estimation of the application before actual memory allocation is necessary to prevent performance degradation.

However, it is complicated as memory allocation related codes are often distributed throughout the program. Furthermore, the total memory usage cannot always be determined statically because it sometimes depends on the input data. More important problem is that memory usage estimation should be more accurate than the execution time estimation due to the misprediction penalty is pretty high as discussed in this section. Therefore, the programmer needs to analyze the entire code for every new application. To avoid this, smart API support for just-in-time memory usage estimation before the execution is essential.

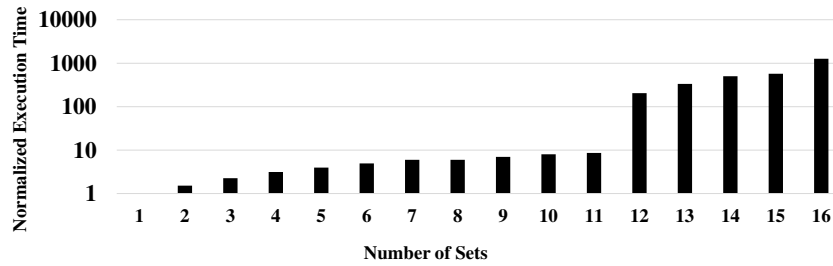


Figure 4. Average total execution time (log scaled) of multiple benchmark executions as the number of processes increases in the Shared-Memory System.

3. ADAMS Framework

3.1. Overview

In this section, we introduce an ADAMS framework that automatically allocates multiple applications to appropriate devices using a periodic device and process information update procedure. ADAMS comprises a simple API set and does not require any additional runtime program. It can be easily applied to the target application, as it does not alter the typical program flow of the OpenCL applications and requires just a simple modification of the original OpenCL APIs into the corresponding ADAMS APIs, as shown in Figure 5. The ADAMS provides six APIs and an OS timer-based software interrupt service routine (ISR), as described in Table 1. In this study, we define the total device execution time for kernel execution and memory transfer between the host and the device, as the device time.

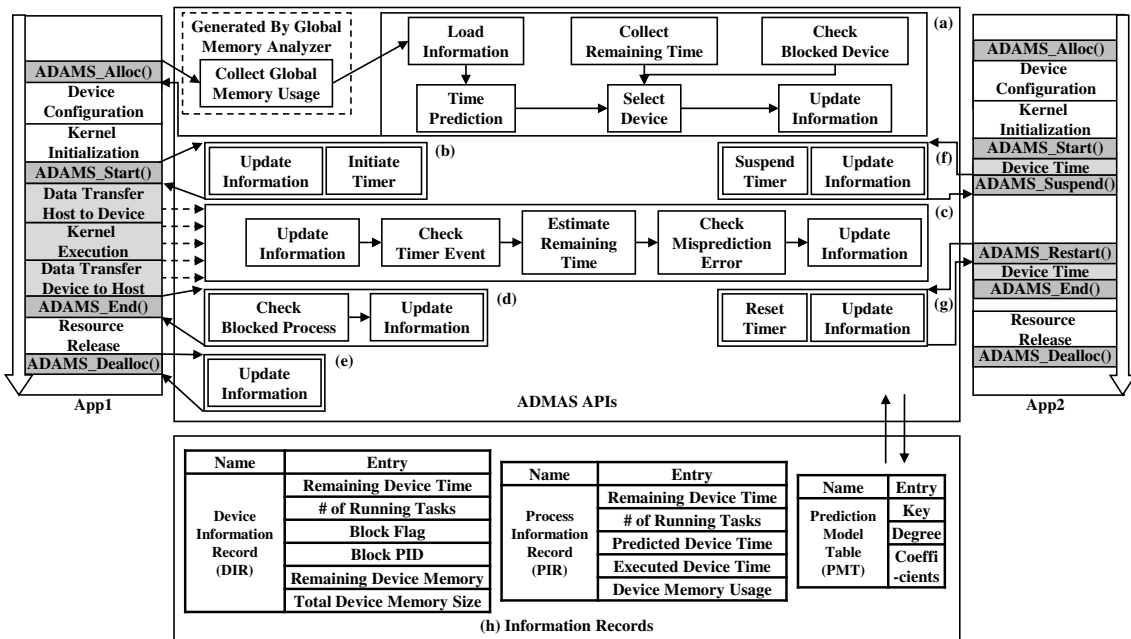


Figure 5. An overview of ADAMS framework, (a–g) API descriptions, (h) information records.

Table 1. ADAMS API Descriptions.

Index	ADAMS API	Description	Input Argument	Return Value
(a)	ADAMS_Alloc	Select the appropriate device to run the OpenCL task considering global memory usage.	platform_id, device_id, problem_size, app_id	error_code
(b)	ADAMS_Start	Start of device time estimation.	timer_object	NULL
(c)	ISR_Call	Time estimation using OS timer. Check time-estimation error.	NULL	NULL
(d)	ADAMS_End	End of device time estimation. Release block status.	timer_object	NULL
(e)	ADAMS_Dealloc	Release all corresponding information in DIR and PIR.	NULL	NULL
(f)	ADAMS_Suspend	Suspend device time estimation without any release of information.	timer_object	NULL
(g)	ADAMS_Restart	Restart device time estimation.	timer_object	NULL

3.2. ADAMS Execution Model

3.2.1. Information Record

To assign a process to an appropriate device, the ADAMS requires information on the running processes and existing devices. As ADAMS does not have a specific control daemon, the required information needs to be shared between the processes. As shown in Figure 5h, the ADAMS uses three data structures: a device list, process lists, and a prediction model table (PMT). These data structures are allocated in the OS shared-memory space, and the processes can update them while execution using the ADAMS APIs. Device information records (DIRs) contain device-specific information and are managed separately for each device. A DIR has six entries, as shown in Figure 5h. The *Remaining Device Time* is the total remaining execution time of the tasks assigned to a device when considering the effects of a preemption, and is calculated by collecting process information records (PIRs) on the processes assigned to the corresponding device. The *Number of Running Tasks* is the total number of tasks that are currently being executed on the device, and is maintained in both a PIR and a DIR separately. It is used to consider the effects of a preemption in the execution time-estimation mechanism. The *Block Flag* and *Block PID* are used for the time-estimation error-handling mechanism. When a time-estimation error occurs, the *Block Flag* and *Block PID* of the relevant device are set to not allocate other new tasks until the error is resolved. In this case, the process in which the error occurred can release *Block Flag* when the error is handled properly. The *Total Device Memory Size* and *Remaining Device Memory* are the memory information of the device. The *Total Device Memory Size* is obtained through the OpenCL API. The *Remaining Device Memory* is the currently available memory size of the devices considering both the total memory size and the memory occupied by currently running tasks. The memory size used can be calculated by applying the *Device Memory Usage* of each PIR during all allocation procedures.

A PIR contains both device- and process-specific information and is managed separately for each process. The *Remaining Device Time* in a PIR represents the remaining execution time of the tasks. To calculate this, PIR entries also have the *Predicted Device Time* and *Executed Device Time*. The *Predicted Device Time* is the predicted execution time of the tasks, which is determined from the device-specific PMT using the performance prediction model and the target problem size. The *Executed Device Time* is the estimated execution time using a concurrent time estimator. The *Device Memory Usage* is the target device global memory size required for the tasks.

Each process has its own process information records (PIRs). The ADAMS updates each DIRs by collecting PIRs of processes running on the device. The PMT contains application-specific device

time prediction models for different problem sizes. Because the information data can be accessed simultaneously from multiple processes, write operations are protected using OS semaphore locks.

Before using the framework, the information records must be initialized through a one-time initialization procedure. This procedure creates information records in the OS shared memory and updates the device information that can be obtained through the OpenCL API. The global memory capacity of each device is automatically recognized by an OpenCL API, but it should be carefully set for shared-memory systems because the OpenCL API returns the total system memory size for every device. Thus, the device memory information of each device should be set manually to the proper size for the shared-memory systems.

3.2.2. Allocation Manager

ADAMS can use dynamic information during process allocation using information records. To achieve maximum system performance for multiple irregularly launched applications, we use a remaining-time-based allocation policy as shown in Algorithm 1. To allocate a new process, the algorithm first calculates the global memory size required for the tasks (M_{Usage}) using a global memory analyzer (GMA) as described in Section 3.2.3. The algorithm then loads the regression model data of the application for all available devices from the PMT (lines 2–4 of Algorithm 1). The remaining total execution time (T_R) for each device ($DIR.T_R$) can be estimated by gathering the remaining time of currently executing processes on the device, from the corresponding PIRs (lines 5–10). Based on this, in lines 11–20, ADAMS determines an optimal device (DIR_{Min}) that has a new minimum remaining device time (T_{Min}) by adding the current remaining device time (T_{Min}) and predicted time (T_P) for the new process. For this process, when some devices are marked as ($DIR.BLOCKED$) owing to the time-estimation error (as discussed in Section 3.2.4), a new process allocation to the devices is not allowed (line 14). In addition, ADAMS can allocate an appropriate device to avoid a memory shortage problem using the memory usage estimation. If the remaining global memory size of a device (M_R) is smaller than the memory requirement ($M_{Required}$) of the new process, the device is not considered a candidate device (line 14). Here, $dList$ and $ProcList$ indicate the “device list” and the “process list”, respectively.

In short, as shown in Figure 6, ADAMS first gathers dynamic information in the beginning, and then selects the most optimal device to a target application based on the minimum remaining time without memory failure. In this figure, P6 is first considered to be allocated to DEV3 owing to the minimal total expected execution time, when five processes are currently running on three devices. However, it is rejected because of the memory shortage problem, and ADAMS then allocates P6 to DEV2.

Once a process is allocated to a device with the minimum expected remaining time, ADAMS re-estimates the remaining global memory size of the device by adding the memory usage of the newly allocated process. If the global memory of all the devices is insufficient, the allocation manager returns a memory failure error, and the framework iteratively retries the allocation until a device with sufficient memory space is available. When the allocation is completed with no errors, ADAMS updates the information records. When all OpenCL-related tasks are completed (all resources are released), ADAMS resets the related information through `ADAMS_Dealloc()`.

Algorithm 1 Process Allocation Algorithm.

Require: *DeviceList dList, ProcessList ProcList, Process P*
Ensure: *process allocation*

{Predict execution time and memory usage of P for all devices}

- 1: $M_{Usage} \leftarrow \text{GetRequiredMemory}(P)$
- 2: **for** DIR in $dList$ **do**
- 3: $T_P[DIR.Index] \leftarrow \text{TimePrediction}(P.AppID, DIR.Index)$
- 4: **end for**

{Update remaining time of each device}

- 5: **for** DIR in $dList$ **do**
- 6: $DIR.T_R \leftarrow 0$
- 7: **for** PIR in $ProcList[DIR.Index]$ **do**
- 8: $DIR.T_R += PIR.T_R$
- 9: **end for**
- 10: **end for**

{Find the target device that has minimum total remaining time}

- 11: initiate T_{Min}
- 12: initiate DIR_{Min}
- 13: **for** DIR in $dList$ **do**
- 14: **if not** $DIR.Blocked$ **and** $DIR.M_R > M_{Required}$ **then**
- 15: **if** $DIR.T_R + T_P[DIR.Index] < T_{Min}$ **then**
- 16: $T_{Min} \leftarrow DIR.T_R + T_P[DIR.Index]$
- 17: $DIR_{Min} \leftarrow DIR$
- 18: **end if**
- 19: **end if**
- 20: **end for**

{Update information in PIR and DIR}

- 21: update PIR_P and DIR_{Min}

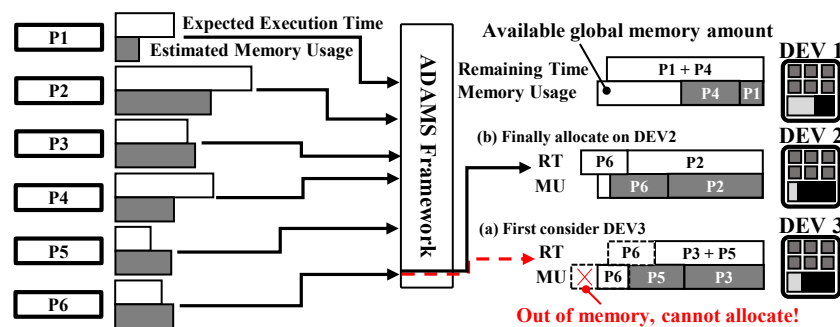


Figure 6. A simplified ADAMS workflow and a process allocation example. (a) a rejected attempt to assign P6 to DEV 3, and (b) a final assignment of P6 on DEV2.

3.2.3. Global Memory Analyzer

An important challenge in avoiding memory errors during execution is the determination of the required memory space prior to device allocation of a target process. The required memory size for an application can vary dynamically depending on the input data. To address this issue, ADAMS extracts minimal instructions to calculate the required memory size using def-use chains during compilation (Figure 7a). By executing these instructions, ADAMS can obtain the required memory size to be secured.

Figure 7b shows an example of the extraction of a minimal instruction from 2DConvolution benchmark [20] to calculate the memory size. ADAMS first identifies the device memory allocation instructions and their predecessor instructions. Subsequently, it removes instructions that do not affect the memory allocation. After all procedures are complete, the overhead for executing the extracted instructions can be minimized.

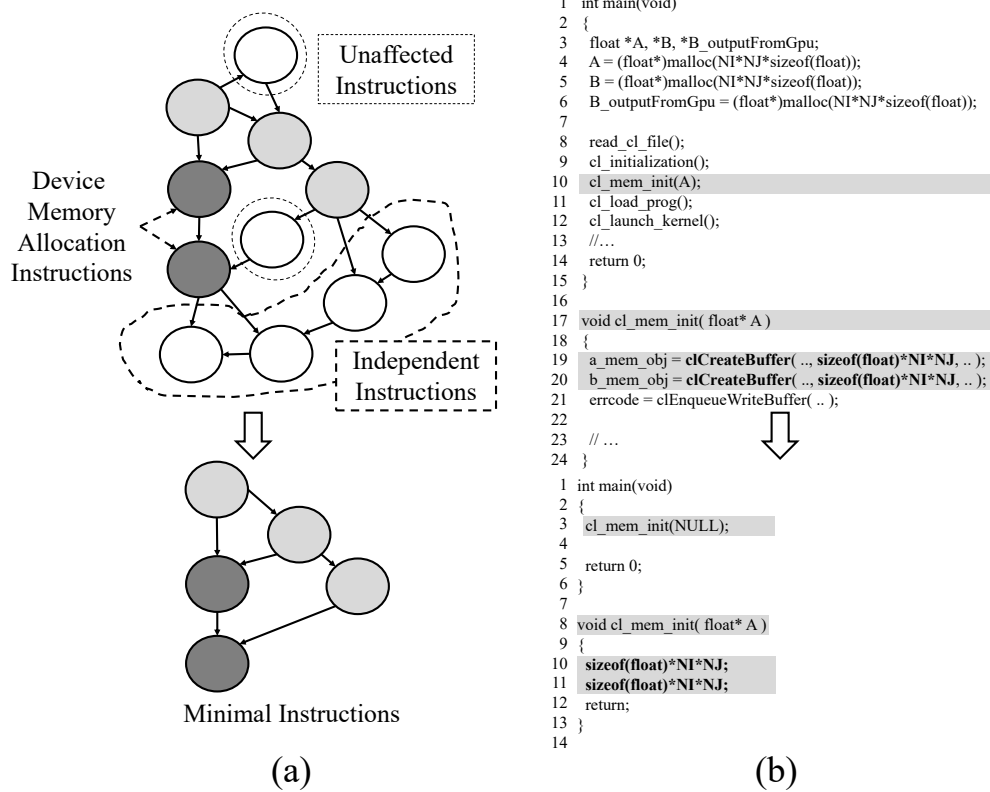


Figure 7. Extracting minimal instructions for memory requirement estimation. (a) Overview and (b) a translation example.

The GMA can be applied to most OpenCL applications (This cannot be applied to applications in which the memory usage is changed by non-reproducible elements such as random numbers). The extraction is automatically performed using code transformation during compilation of the host code. After the minimal instructions are extracted, the analyzer wraps the extracted code into a custom API whose return value is the global memory usage of the application.

3.2.4. Concurrent Time Estimator

The time estimator measures the device time by using the OS timer-based software ISR to update the information records. The ISR is periodically executed on the host during the device time (Figure 5b,d). As shown in Figure 8, Equation (1) is used to estimate the remaining execution time (T_R) of each target process, considering the impact of concurrent execution on a device, where N_{T_i} denotes the number of tasks running simultaneously on the device on i th period, and P_i denotes the duration of i th period. During the P_i , the execution of each concurrent task on the same device slows down owing to the preemptive multitasking of co-running kernels. Therefore, the T_R of each task can be estimated roughly by using Equation (1) with the expected execution time (T_P).

$$T_R = T_P - \sum_i^n \frac{P_i}{N_{T_i}} \tag{1}$$

Application Workflow	Application 1					
	Application 2			Application 3		
Remaining time of Application 1	T_P	$T_P - P_A/1$	$T_P - P_A/1 - P_B/2$	$T_P - P_A/1 - P_B/2 - P_C/3$	0	
# of running tasks	0	1	2	3		
Time period of Application 1		A	B	C	D	

Figure 8. Time-estimation example.

In this paper, we define an *event* as a change in the number of running tasks. Every time an event occurs, ADAMS updates the event time as the last event time (T_{Event}) and N_T on both the PIR and DIR. With this information, ADAMS can determine the number of simultaneously running tasks on the device during the unit interval, P_i , and estimate the time spent on each process to effectively perform a task using Algorithm 2. Every time a timer ISR occurs, to check if an event has occurred, the framework compares the N_T of the PIR and DIR (line 1 of Algorithm 2). If an event occurs, the base time (T_{Base}) is updated by adding the difference between the T_{Event} stored in the PIR and DIR to the T_{Base} in the PIR. In addition, T_{Event} and N_T in both the PIR and DIR are updated with the new values (lines 2–4). By doing this, T_{Base} stores the total execution time of a process from the beginning to the last event occurring on the device. The processes can also check whether an event has occurred in another process using the difference between such information stored in its PIR and DIR. Here, T_{Now} is the time stamp when the ISR is called. The difference between T_{Now} and T_{Event} is the period when the task was executed without changing the N_T . Therefore, by dividing the difference between T_{Now} and T_{Event} by N_T , we can estimate the effective execution time of the task during that period. The executed device time ($T_{Executed}$) of the task is also calculated by adding T_{Base} and this (line 6).

Algorithm 2 Remaining Time Update Algorithm.

Require: PIR of process, DIR of device that process allocated

Ensure: update of remaining time in PIR

```

    {If event occurs, calculate base time and updates to value of DIR}
1: if PIR.NT != DIR.NT then
2:   PIR.TBase += (DIR.TEvent - PIR.TEvent) / PIR.NT
3:   PIR.TEvent ← DIR.TEvent
4:   PIR.NT ← DIR.NT
5: end if

    {Calculate remaining time of process}
6: PIR.TExecuted ← PIR.TBase + (TNow - PIR.TEvent) / PIR.NT
7: PIR.TR ← PIR.TPrediction - PIR.TExecuted

    {If error occurs, block device}
8: if PIR.TR < 0 then
9:   DIR.Blocked ← true
10:  DIR.BlockPID ← Process.PID
11: end if

```

Please note that the degree of slowdown on concurrent execution of multiple kernels can vary depending on target device architectures. Therefore, ADAMS has an error-handling mechanism to tolerate the time-estimation errors (lines 8–11). If a task is not finished even when its remaining time becomes zero due to the wrong execution time estimation or severe slowdown on concurrent execution, ADAMS blocks the additional allocation of new processes to the device (running the error process) until the process is finished.

ADAMS_Start() and ADAMS_End() define the device time region. ADAMS also provides ADAMS_Suspend()/Restart() to exclude idle device time for time estimation when multiple kernel

executions exist within an application, as described in Table 1. They temporarily disable/re-enable the OS-timer-based PIR updates.

3.2.5. Time Prediction

Estimating the remaining time using the concurrent time estimator described in Section 3.2.4 requires the estimated execution time (T_p) of all applications on all devices. The execution time of GPGPU tasks can be fairly predicted with high accuracy using machine-learning techniques ([9–11]). Similar to these previous studies, in this study, a regression-based model is used for the execution time prediction. Because the performance of an application varies considerably depending on the specific features of the target device, such as the computing power or memory bandwidth, the time prediction should be applied differently for each device on multi-device systems. Therefore, the prediction model is set separately according to the application and target device. It is trained using profiled data for various problem sizes. Because the ADAMS framework has a reasonable prediction error-handling mechanism, a fair performance gain can be achieved even when some prediction errors occur at runtime, although higher model accuracy is always preferred.

4. Evaluation and Discussion

We tested our framework on both shared and non-shared-memory systems, as shown in Table 2. In the shared-memory system, we restrict the available memory space for each device because the CPUs and the GPU share a single 16 GB DRAM. We use Clang [21] 6.0 and an LLVM [22] 6.0 compilation framework for the global memory analysis. As shown in Table 3, ADAMS uses an OS shared memory to share information between processes similar to Heartbeats [23]. We use a set of applications from the polybench benchmark suite [20] with customized problem sizes for the target applications to have reasonable execution time ranges (Table 4). The prediction models are trained with the offline profiled data from the single execution of the target programs, and different data sets are used for training and evaluation. The framework predicts the device time according to the problem size by using the information stored in the PMT. The theoretical Oracle (Theo.Oracle) is the statically calculated ideal performance in the best device-selection configuration, based on the profiled data (used in Figure 9a,c).

The performance of the RR allocation method significantly depends on the execution order of the application. Among $N!$ number of possible execution order combinations for N applications, 100 randomly selected orders are used to compare the performance of multiple device allocation policies, including RR and ADAMS with and without a memory consideration.

Table 2. System Configuration

Non-Shared-Memory System					
Index	Device	Number of Compute Unit	Global Memory Amount	Global Memory Bandwidth	PCIe (Lane)
Dev1	GTX 1080 Ti	28	11,178 MB	11 Gbps	PCIe 3.0 (x4)
Dev2	GTX 1050	5	2000 MB	7 Gbps	PCIe 3.0 (x8)
Dev3	GTX 1050	5	2000 MB	7 Gbps	PCIe 3.0 (x4)
Shared-Memory System					
Index	Device	Number of Compute Unit	Global Memory Amount	Global Memory Bandwidth	
Dev1	Intel i7-7700K	4	4096 MB	35.76 Gbps	
Dev2	Intel Gen 9 HD Graphics 630 [14]	24	4096 MB	35.76 Gbps	

Table 3. Size of Information Records.

System Type	Device List	Process List	PMT	Total
Non-shared	0.84 KB	175.2 KB	48.8 KB	224.84 KB
Shared	0.56 KB	89.6 KB	36 KB	126.16 KB

Table 4. Problem Sizes of Workloads.

Application	Non-Shared-Memory System		Shared Memory System
	Small Set	Large Set	
2dconv	6400	10,400	22,000
2 mm	1600	2400	1061
3 mm	1600	2400	1061
atax	11,200	16,000	30,000
bicg	11,200	16,000	30,000
gemm	2400	3200	1061
gesummv	5600	8000	20,000
gramschmidt	640	1280	1441
mvt	7200	11,200	20,000

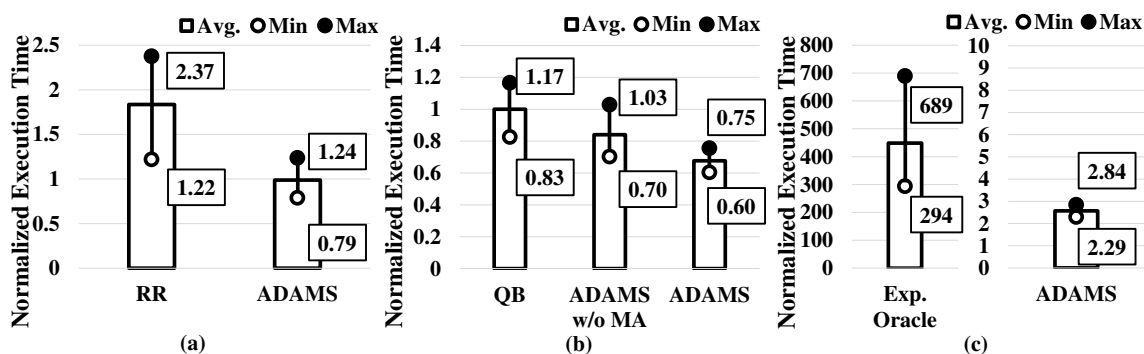


Figure 9. Performance comparisons. (a) Effectiveness of proposed allocation policy and (b) effectiveness of global memory consideration in the Non-Shared-Memory System. (c) Effectiveness of global memory consideration in the Shared-Memory System.

4.1. Allocation Policy on Non-Shared-Memory Systems

To evaluate effectiveness of allocation policy, we compare ADAMS to both RR and Theo.Oracle on the non-shared-memory system. Figure 9a shows execution time of ADAMS and RR, normalized to the that of Theo.Oracle. As shown in Table 4, to avoid performance changes due to memory allocation failures, we use a small set of the application with an approximate 2 GB total memory size (similar to the global memory size of DEV(2/3) in Table 2).

This figure shows that ADAMS improves the performance by 1.85 times on average over the RR. This is because the RR does not consider the dynamic information of the system discussed in Section 2.2; however, ADAMS successfully selects appropriate devices that have the minimum expected execution time. Compared to Theo.Oracle, ADAMS shows almost the same performance. This is because Theo.Oracle does not consider the possibility that the host-device memory transfer and kernel execution might overlap between multiple applications. The overlap cannot be predicted statically and cannot be calculated theoretically, and is determined at runtime. However, this overlap can result in higher performance. Thus, all experiments that use real machines except Theo.Oracle permit this. Therefore, all results may achieve better performance than Theo.Oracle, and the result shows that the allocation policy of ADAMS selects the device that can provide the earliest completion time properly.

Moreover, ADAMS shows more stable performance over the RR because it has a standard deviation (STD) of only 0.1 over an average normalized execution time of 0.99, whereas the RR had

a STD of 0.26 over an average normalized execution time of 1.83. This small performance variance in different execution orders demonstrates that ADAMS can guarantee a stable performance gain on multi-device systems, regardless of input application orders.

4.2. Memory Consideration on Non-Shared-Memory Systems

Figure 9b shows the results of ADAMS, ADAMS without memory analysis ($ADAMS_{w/o_MA}$), and the QB. All results are normalized to an average execution time of QB. We use a large dataset and increase the number of processes to 18, as shown in Table 4, to have several situations in which processes cannot run properly owing to insufficient global memory space. In particular, $ADAMS_{w/o_MA}$ is the ADAMS framework without global memory usage estimation. In case of a memory failure in $ADAMS_{w/o_MA}$, the process with the failure restarts immediately. To evaluate ADAMS with the latest work, we compare the performance with an in-house version of VirtCL [9] (QB), which is a queue-based multi-OpenCL program-management system (Please note that this is an in-house version because the original work is not an open source program).

In Figure 9b, the performance of $ADAMS_{w/o_MA}$ improves of only 1.19 times the QB owing to multiple memory allocation errors. As shown in Figure 3, the average error occurrences of $ADAMS_{w/o_MA}$ and QB are 6.29 and 3.67, respectively. For 100 execution orders, there is a minimum of one error occurrences for both cases, and a maximum of 12 and 10 error occurrences, respectively. Despite more errors, $ADAMS_{w/o_MA}$ shows a slightly better performance than QB because it selects other devices with available memory when a memory allocation failure occurs, whereas QB waits for other processes to release memory on the same device. Furthermore, ADAMS shows a high performance gain of 1.48 times compared to the QB owing to the absence of memory allocation errors.

In terms of performance variance, ADAMS is better than $ADAMS_{w/o_MA}$ and QB because ADAMS shows a STD of only 0.029 for an average normalized execution time of 0.68. However, $ADAMS_{w/o_MA}$ and QB show considerably larger STDs of 0.068 and 0.075 for average normalized execution times of 0.84 and 1, respectively.

4.3. Memory Consideration on Shared-Memory Systems

In shared-memory systems, we limit the maximum memory size of each device to 4 GB as shown in Table 2; the rest of the framework is identical to that used in non-shared-memory systems. Figure 9c shows execution time of ADAMS and experimental Oracle (Exp.Oracle), which are normalized to the Theo.Oracle. The Exp.Oracle is the result of the actual execution of all the applications with the same device mapping as the Theo.Oracle. We use the data set shown in Table 4, for which the total memory usage of the processes exceeds the available memory capacity. The total memory size for the OpenCL memory objects of all the processes is approximately 16.4 GB.

Exp.Oracle shows an average performance that is about 449 times lower than that of the Theo.Oracle because a high memory-swapping overhead exists when the total memory usage is larger than the main memory size. However, ADAMS shows an average performance that is lower by only 0.390 times that of the Theo.Oracle. This is because ADAMS does not use swap memory as ADAMS limits the memory usage of each device to 4GB. Therefore, processes cannot be executed until there is at least one device that secures sufficient memory spaces. Consequently, the average performance of ADAMS is about 175 times higher than that of the Exp.Oracle, which does not consider memory management.

In addition to the high swap overhead, some processes may be unexpectedly terminated by the OS or show unintended errors such as segmentation faults due to the memory shortage. As shown in Figure 9c, ADAMS and Exp.Oracle show STDs of 0.15 and 114 for average normalized execution times of 2.57 and 449, respectively. Therefore, smart memory management is important to prevent unexpected performance degradation in shared-memory systems; ADAMS can handle such problems effectively.

4.4. Case Study: Multiple IRIS Recognition Applications

To evaluate the effectiveness of ADAMS framework, we tried to execute multiple *IRIS* recognition applications concurrently on the shared-memory system. For this, we implemented an in-house version of *IRIS* recognition application [24], based on a convolutional neural network (CNN) consisting of six CNN-layers (2Dconvolution, normalization, and max pooling kernels (OpenCL)) and a fully connected layer (parameter size: 10.29 MB). The *IRIS* is pre-trained and performs inference with a 64 image batch size using a CASIA-Iris-V4 [25] input set. Global memory usage by an *IRIS* is approximately 225 MBs. A small dataset launches 16 *IRIS* detection processes concurrently without memory problems (approximately 3.5 GB of global memory usage), and a large data set launches 75 *IRIS* detection processes concurrently with up to 16.5 GB of global memory usage. Differing from polybench applications, the *IRIS* application has a uniform execution time according to the input image. Therefore, the performance of *IRIS* is not significantly affected by the execution order.

Figure 10 shows performance of RR, $ADAMS_{w/o_MA}$, and ADAMS. The results are normalized to RR on each data set. For the small set, as shown in Figure 10a, ADAMS shows $1.68\times$ higher performance over RR by efficiently allocating *IRIS* to multiple devices, and shows almost same performance to $ADAMS_{w/o_MA}$ because the set does not incur performance slowdown, owing to the small memory usage. For the large set, as shown in Figure 10b, ADAMS shows $3.24\times$ higher performance over RR by effectively preventing memory-swapping overhead. However, $ADAMS_{w/o_MA}$ shows $0.39\times$ lower performance over RR because memory-swapping occurs more frequently in $ADAMS_{w/o_MA}$ owing to the ISR calls for time estimation. As mentioned in Section 4.2, $ADAMS_{w/o_MA}$ has the same framework as ADAMS, but does not manage memory usage. Thus, $ADAMS_{w/o_MA}$ shows almost the same performance as ADAMS in a small set because the set does not have any memory problems. However, in a large set, the performance is worse than that of ADAMS owing to frequent memory failures. Therefore, both efficient device allocation and memory limitation techniques are important to maximize system performance, and to keep appropriate memory usage precisely is critical to avoid huge performance degradation.

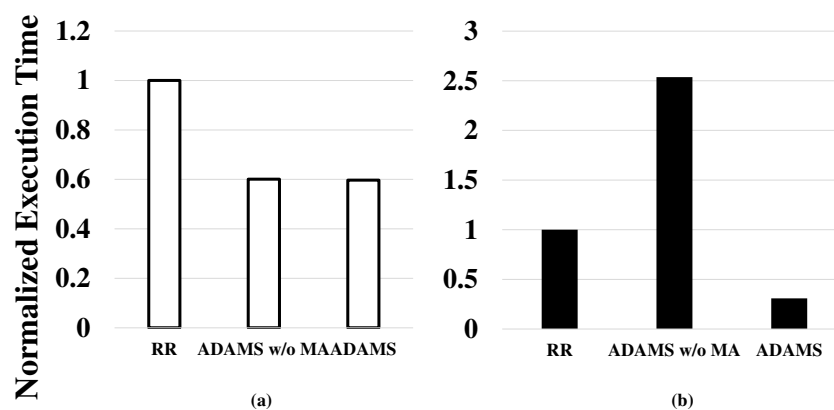


Figure 10. Case study of ADAMS for the *IRIS* application with (a) small set and (b) large set.

4.5. Overhead

We measured the overhead of a single execution of each application on the DEV2 (Table 2) with a small data set in the non-shared memory system. The overall overhead of ADAMS without an ISR call is only 0.025% of whole device time. Global memory analysis shows considerable overhead variance across target applications because it depends on application characteristics. ISR calls use 13% of the total execution time but the overhead is usually hidden because it is handled by host. The results show that only 2.10% performance difference exists between executions with and without ISR calls.

5. Related Work

There are several previous studies on efficiently running OpenCL programs on multi-GPU systems. Several prior studies [26–32] have introduced efficient mechanisms to distribute a single kernel execution to multiple devices properly. FluidiCL [26] dynamically splits kernel tasks into multiple workgroups based on flattened unique numbers and executes them on both the CPU and GPU. Maestro [27] introduces an autotuning technique to use the runtime information for determining the best partitioning ratio of the target task. LBCL [28] implements an OpenCL installable client driver (ICD) layer to execute an OpenCL program on multiple devices and supports a computing-power-based load balancing mechanism. Grewe [29] partitions a single kernel statically into a target CPU-GPU system using a machine-learning-based prediction model. Kaleem [30] presents several profiling-based scheduling techniques to resolve the load imbalance in integrated CPU-GPU processors. Majeti [31] introduces a two-level hierarchical data layout framework for heterogeneous architectures. They build the affinity graph by using a parallel intermediate representation and determine the best data layout for the program automatically. Taylor [32] proposes compiler-based framework to map and schedule tasks on embedded heterogeneous systems using only code features. All these approaches enhance the total performance of the OpenCL programs by using multiple devices; however, these approaches are restricted to only one OpenCL application, in contrast to ADAMS. In addition, there are several works on load balancing of tasks on heterogeneous clusters [33–35].

Singh [36], VirtCL [9] and SnuCL [8] are the most closely related approaches to ADAMS in that they consider a multi-application execution on multi-GPU systems. Singh [36] proposes a runtime management approach that performs mapping and partitioning for embedded CPU-GPU systems considering processor's processing capabilities to improve both performance and energy efficiency. VirtCL [9] provides a device abstraction layer using the CLDaemon program that manages kernel mapping to multiple physical devices. It schedules the kernel using a history-based regression model similar to the ADAMS framework. SnuCL [8] is a framework for the orchestration of multiple OpenCL programs on heterogeneous clusters. It assigns kernels to appropriate devices for minimizing the memory transfer overhead using the SnuCL runtime. Although these studies have demonstrated substantial performance gains in similar domains as ADAMS, they still require a high control overhead owing to the existence of additional management programs. Furthermore, they use a serial kernel launch algorithm with a kernel queue structure and do not consider the global memory usage issue; therefore, they cannot take full advantage of GPU multitasking.

6. Conclusions

We propose a dynamic multiple data-parallel application allocation framework (ADAMS), to efficiently allocating multiple processes to multiple devices. To find the best target device for a new process, it first collects the runtime information of several available devices from shared OS memory and derives the expected execution time of the process on each device. It then selects the optimal device with no memory insufficiency problems.

In this paper, the evaluation shows that ADAMS achieves a $1.85\times$ higher performance than the classic round-robin algorithm in the non-shared-memory system with a memory failure-free condition. The results show the importance of runtime information sharing in assigning GPGPU tasks to optimal target devices. ADAMS effectively shares static/dynamic information of GPGPU applications without any additional daemon process, and can achieve more stable performance improvement. Under conditions where memory failure occurs, ADAMS shows a $1.48\times$ better performance than the queue-based approach in the non-shared-memory system. In the shared-memory system, ADAMS achieves almost the same performance as theoretical oracle, and shows $449\times$ higher performance than the real execution without memory considerations. These results show that smart device memory management can prevent large performance degradation when executing multiple GPGPU applications. Therefore, these experimental results demonstrate that the proposed framework can effectively handle the memory problem to avoid performance degradation.

Author Contributions: The first author, D.K. and corresponding author, Y.P., are responsible for the overall work in this paper. The second author S.K. has helped implement the framework, and helped writing the related work. Also, he provided useful comments for the manuscript. The third author J.L., the fourth author S.J., and the fifth author W.K. performed part of the experiments and provided effective ideas. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (grant number: 2018R1D1A1B07050609), by Institute for Information & communication Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-00142), and by the LG Electronics CTO Software Platform Lab. grant funded by the LG Electronics Inc.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Redmon, J.; Farhadi, A. Yolov3: An incremental improvement. *arXiv* **2018**, arXiv:1804.02767.
2. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2016**, *39*, 1137–1149. [[CrossRef](#)] [[PubMed](#)]
3. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An automated end-to-end optimizing compiler for deep learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 578–594.
4. Kim, J.; Kwon Lee, J.; Mu Lee, K. Deeply-recursive convolutional network for image super-resolution. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 1637–1645.
5. Garcia, J.; Dumont, R.; Joly, J.; Morales, J.; Garzotti, L.; Bache, T.; Baranov, Y.; Casson, F.; Challis, C.; Kirov, K.; et al. First principles and integrated modelling achievements towards trustful fusion power predictions for JET and ITER. *Nucl. Fusion* **2019**, *59*, 086047. [[CrossRef](#)]
6. Kirk, D. NVIDIA CUDA software and GPU parallel computing architecture. In Proceedings of the ISMM, Montreal, QC, Canada, 7–14 May 2007; Volume 7, pp. 103–104.
7. Stone, J.E.; Gohara, D.; Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* **2010**, *12*, 66–73. [[CrossRef](#)] [[PubMed](#)]
8. Kim, J.; Seo, S.; Lee, J.; Nah, J.; Jo, G.; Lee, J. SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In Proceedings of the 26th ACM International Conference on Supercomputing, San Servolo Island, Venice, Italy, 1–26 June 2012; ACM: New York, NY, USA, 2012; pp. 341–352.
9. You, Y.P.; Wu, H.J.; Tsai, Y.N.; Chao, Y.T. VirtCL: A framework for OpenCL device abstraction and management. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Francisco, CA, USA, 7–11 February 2015; ACM: New York, NY, USA, 2015; pp. 161–172.
10. Lee, J.; Samadi, M.; Mahlke, S. Orchestrating multiple data-parallel kernels on multiple devices. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), San Francisco, CA, USA, 10–21 October 2015; pp. 355–366.
11. Luk, C.K.; Hong, S.; Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Proceedings of the 42nd Annual International Symposium on Microarchitecture, New York, NY, USA, 12–16 December 2009; pp. 45–55.
12. Intel Corporation. In Proceedings of the 8th and 9th Generation Intel Core Processor Families. Available online: <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/8th-gen-core-family-datasheet-vol-1.pdf> (accessed on 1 November 2020).
13. Intel. Intel Core i9-9900K Processor (16M Cache, up to 5.00 GHz). Available online: <https://ark.intel.com/content/www/us/en/ark/products/186605/intel-core-i9-9900k-processor-16m-cache-up-to-5-00-ghz.html> (accessed on 1 November 2020).
14. Intel Corporation. The Compute Architecture of Intel Processor Graphics Gen9. Available online: <https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf> (accessed on 1 November 2020).

15. Branover, A.; Foley, D.; Steinman, M. Amd fusion apu: Llano. *IEEE Micro* **2012**, *32*, 28–37. [[CrossRef](#)]
16. NVIDIA. NVIDIA Tesla P100. Available online: <http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf> (accessed on 1 November 2020).
17. NVIDIA TESLA. V100 GPU Architecture: The World'S Most Advanced Datacenter GPU. Technical Report. Available online: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> (accessed on 1 November 2020).
18. Park, J.J.K.; Park, Y.; Mahlke, S. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*; ACM: New York, NY, USA, 2015; pp. 593–606.
19. Adriaens, J.T.; Compton, K.; Kim, N.S.; Schulte, M.J. The case for GPGPU spatial multitasking. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, New Orleans, LA, USA, 25–29 February 2012.
20. Grauer-Gray, S.; Xu, L.; Searles, R.; Ayalasomayajula, S.; Cavazos, J. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of the 2012 Innovative Parallel Computing (InPar)*, San Jose, CA, USA, 13–14 May 2012; pp. 1–10.
21. Clang: A C Language Family Frontend for LLVM. 2007. Available online: <http://clang.llvm.org> (accessed on 1 November 2020).
22. Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 20–24 March 2004; pp. 75–86.
23. Hoffmann, H.; Eastep, J.; Santambrogio, M.D.; Miller, J.E.; Agarwal, A. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, Washington, DC, USA, 7–11 June 2010; pp. 79–88.
24. Alaslani, M.G.; Elrefaei, L.A. Convolutional neural network-based feature extraction for iris recognition. *Int. J. Comp. Sci. Inf. Tech.* **2018**, *10*, 65–78. [[CrossRef](#)]
25. CASIA Iris-V4.0 Database. Available online: <http://biometrics.idealtest.org/> (accessed on 1 November 2020).
26. Pandit, P.; Govindarajan, R. Fluidic kernels: Cooperative execution of opengl programs on multiple heterogeneous devices. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*; Orlando, FL, USA, 15–19 February 2014; p. 273.
27. Spafford, K.; Meredith, J.; Vetter, J. Maestro: Data orchestration and tuning for OpenCL devices. In *European Conference on Parallel Processing*; Springer: Berlin, Germany, 2010; pp. 275–286.
28. Carlos, S.; Toharia, P.; Bosque, J.L.; Robles, O.D. Static multi-device load balancing for opengl. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, Leganes, Spain, 10–13 July 2012; pp. 675–682.
29. Grewe, D.; O'Boyle, M.F. A static task partitioning approach for heterogeneous systems using OpenCL. In *International Conference on Compiler Construction*; Springer: Berlin, Germany, 2011; pp. 286–305.
30. Kaleem, R.; Barik, R.; Shpeisman, T.; Hu, C.; Lewis, B.T.; Pingali, K. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Edmonton, AB, Canada, 23–27 August 2014; pp. 151–162.
31. Majeti, D.; Meel, K.S.; Barik, R.; Sarkar, V. Automatic data layout generation and kernel mapping for cpu+gpu architectures. In *Proceedings of the 25th International Conference on Compiler Construction*, Barcelona, Spain, 17–18 March 2016; pp. 240–250.
32. Taylor, B.; Marco, V.S.; Wang, Z. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. *ACM Sigplan Not.* **2017**, *52*, 11–20. [[CrossRef](#)]
33. Pfander, D.; Daiß, G.; Pflüger, D. Heterogeneous Distributed Big Data Clustering on Sparse Grids. *Algorithms* **2019**, *12*, 60. [[CrossRef](#)]
34. Tupinambá, A.L.R.; Sztajnberg, A. Transparent and optimized distributed processing on gpus. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 3673–3686. [[CrossRef](#)]

35. Chen, C.; Li, K.; Ouyang, A.; Li, K. Flinkcl: An opencl-based in-memory computing architecture on heterogeneous cpu-gpu clusters for big data. *IEEE Trans. Comput.* **2018**, *67*, 1765–1779. [[CrossRef](#)]
36. Singh, A.K.; Basireddy, K.R.; Prakash, A.; Merrett, G.V.; Al-Hashimi, B.M. Collaborative adaptation for energy-efficient heterogeneous mobile SoCs. *IEEE Trans. Comput.* **2019**, *69*, 185–197. [[CrossRef](#)]

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).