MANCHESTER
1824
The University of Manchester

The University of Manchester Research

Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Papadimitriou, M., Markou, E., Fumero Alfonso, J., Stratikopoulos, A., Blanaru, F-G., & Kotselidis, C-E. (Accepted/In press). *Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes*. 125-138. Paper presented at The 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21).

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:26. Aug. 2022

Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes

Michail Papadimitriou
The University of Manchester
United Kingdom
michail.papadimitriou@manchester.ac.uk

Eleni Markou
BEAT
Greece
e.markou@thebeat.co

Juan Fumero
The University of Manchester
United Kingdom
juan.fumero@manchester.ac.uk

Athanasios Stratikopoulos
The University of Manchester
United Kingdom
{fist}.{last}@manchester.ac.uk

Florin Blanaru
The University of Manchester
United Kingdom
florin.blanaru@manchester.ac.uk

Christos Kotselidis
The University of Manchester
United Kingdom
christos.kotselidis@manchester.ac.uk

Abstract

Modern commodity devices are nowadays equipped with a plethora of heterogeneous devices serving different purposes. Being able to exploit such heterogeneous hardware accelerators to their full potential is of paramount importance in the pursuit of higher performance and energy efficiency. Towards these objectives, the reduction of idle time of each device as well as the concurrent program execution across different accelerators can lead to better scalability within the computing platform.

In this work, we propose a novel approach for enabling a Java-based heterogeneous managed runtime to automatically and efficiently deploy multiple tasks on multiple devices. We extend TornadoVM with parallel execution of bytecode interpreters to dynamically and concurrently manage and execute arbitrary tasks across multiple OpenCL-compatible devices. In addition, in order to achieve an efficient device-task allocation, we employ a machine learning approach with a multiple-classification architecture of Extra-Trees-Classifiers. Our proposed solution has been evaluated over a suite of 12 applications split into three different groups. Our experimental results showcase performance improvements up 83% compared to all tasks running on the single best device, while reaching up to 91% of the oracle performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VEE '21, April 16, 2021, Virtual, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8394-3/21/04...$15.00
https://doi.org/10.1145/3453933.3454019

CCS Concepts: • **Software and its engineering** → **Virtual machines**.

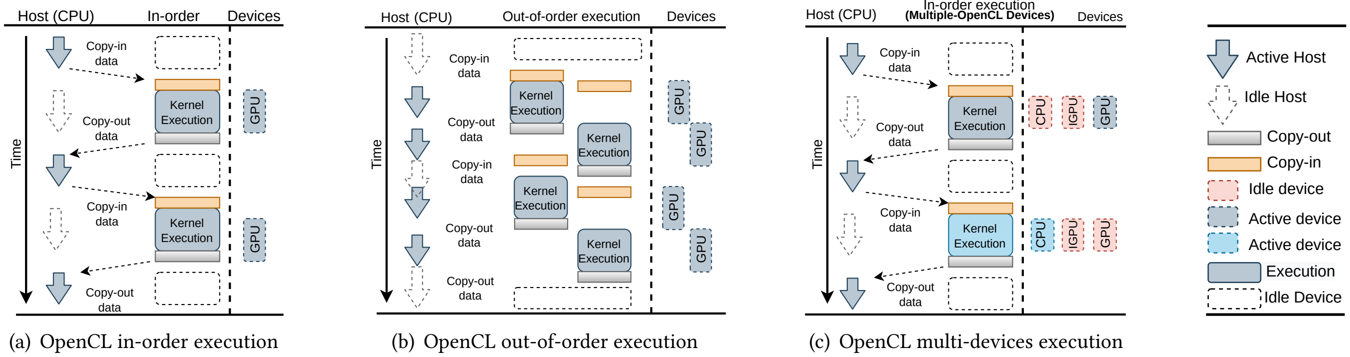
Keywords: JVM, Heterogeneous Hardware, Bytecodes, Multi--threading

ACM Reference Format:
Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. 2021. Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453933.3454019

1 Introduction

High demand for increased computational capabilities and power efficiency has resulted in commodity devices to be equipped with a diverse set of heterogeneous hardware. Desktops, laptops, and smartphones have embraced heterogeneity through multi-core CPUs, energy-efficient integrated GPUs, and powerful discrete GPUs. Consequently, the presence of such hardware has made parallel programming constructs, such as OpenCL [49], OneAPI [30], and CUDA [14] the new norm. Such frameworks support asynchronous data-driven programming models that enable both data parallel and task parallel paradigms of computation for implementing high performance parallel applications.

To ease the transition towards those programming paradigms, a substantial amount of research has focused on making high-level programming abstractions widely available. For instance, TVM [10] is a flexible machine learning compiler framework for CPUs, GPUs and machine learning accelerators, while Halide [1] is a programming language for image processing pipelines on CPUs, GPUs, and FPGAs. In addition, approaches like IBM's J9 [29] with GPU support, StreamIT [28, 50], Aparapi [4] and TornadoVM [17] allow Java programs to execute on heterogeneous hardware. However, although the aforementioned solutions aim at closing



(a) OpenCL in-order execution (b) OpenCL out-of-order execution (c) OpenCL multi-devices execution

Figure 1. Overview of OpenCL execution modes (Out-of-order on Single Device vs In-order on Multiple Devices)

the programmability gap, they tend to focus on single device execution and utilization. Since the availability of multiple devices within a computing platform has become the new norm, heterogeneous managed runtimes [11, 36] and high-level programming frameworks need to also be able to schedule, orchestrate and scale-up the executed programs on a large number of diverse hardware without depending on the user's expertise.

In this work, we introduce a Multiple-Tasks on Multiple-Devices (MTMD) mechanism which allows seamless concurrent heterogeneous execution of Java programs. Our contribution lies in the design, implementation, and evaluation of a new scalable on multiple devices and modular system that employs custom parallel bytecode interpreters that are capable of orchestrating parallel execution on multiple devices, while using intelligent task scheduling across multiple hardware accelerators. The framework is built upon TornadoVM [12, 17] that allows Java programs to leverage heterogeneity by dynamically compiling them to OpenCL and orchestrating execution.

Our proposed system leverages and extends the virtualization layer of TornadoVM by decomposing and executing applications at the task-level granularity into blocks of instructions for scheduling (bytecodes for orchestrating the execution). To perform this decomposition, our system automatically performs data dependency analysis and it generates a set of blocks of bytecodes for enabling concurrent execution on heterogeneous devices. Each individual available device is assigned a system thread that runs an instance of the interpreter that executes the generated bytecodes. Since concurrency does not implicitly guarantee the efficient allocation of tasks to devices, we employ a machine learning (ML) based scheduling approach for dynamically selecting which task will run on which device. To achieve that, program features are extracted through the compiler graph and passed onto a pre-trained multiple classifier system that selects the target device among CPUs, integrated GPUs, and discrete GPUs. The combination of parallel bytecode execution, concurrent deployment of execution contexts at the task-level granularity, and intelligent mapping of tasks onto the available devices, results in the seamless and concurrent execution of multiple-tasks on multiple-devices.

In detail, this work makes the following contributions:

- It introduces a novel mechanism for enabling Multiple-Tasks Multiple-Devices (MTMD) execution for Java programs on heterogeneous devices.
- It presents a static code feature extractor from a compiler Intermediate Representation (IR) for training our ML-based scheduling model.
- It introduces a multiple-classifier system to allocate tasks onto a device selected among CPUs, integrated GPUs, and discrete GPUs.
- It evaluates the proposed approach across twelve applications scheduled in three groups for concurrent execution, with up to 83% performance improvement against the best single device, and up to 91% of the Oracle performance.

2 Background

2.1 OpenCL Execution Modes

OpenCL [49] is one of the first standards for programming heterogeneous platforms by offering a uniform Application Programming Interface (API) and device platform abstraction that allows all different types of devices to be programmed in the same portable way. Commodity devices, like personal computers, can be equipped with a variety of OpenCL-compatible devices, ranging from multi-core CPUs to high-performing discrete GPUs, and FPGAs. By employing OpenCL, developers can harness the computational capabilities of such hardware accelerators to exploit the attributes of their programs, such as task and instruction-level parallelism.

Throughout the years, the OpenCL standard has been extended to better utilize the niche features of modern heterogeneous devices. Part of OpenCL's optimization process was the introduction of different execution modes both for single and multiple device configurations. Figure 1 exemplifies the

three currently supported execution modes of OpenCL: a) in-order single-device execution, b) out-of-order singe-device execution, and c) in-order multiple-devices execution.

When utilizing in-order single-device execution, as shown in Figure 1a, developers can offload parts of their programs for acceleration on a single OpenCL-compatible device. In addition, in this mode, data copying between the host and the device never overlaps with the execution of the code (or kernel) on the device. This results in a strictly sequential in-order execution mode in which the device can remain idle between the intervals of data copying and execution. To mitigate the introduction of idle cycles, OpenCL introduced the out-of-order execution mode (Figure 1b) in which developers can overlap data copying and kernel execution. In this mode, although a single-device is still utilized, the idle cycles are greatly reduced by simultaneously copying data between the host and device, while executing code on the accelerator. Finally, the last execution mode of OpenCL regards the multi-devices execution, as shown in Figure 1c. In this mode, developers can build multiple-contexts (one per device) and utilize more than one accelerator from within their programs. This mode supports only in-order execution that again results in idle cycles between the different devices.

To address the limitations and the idle-cycles introduced by the multi-devices in-order execution mode of OpenCL, a number of frameworks has been proposed. For instance, VirtCL [53], SnuCL [34], PySchedCL [21], FluidiCL [42], MultiCL [2], EngineCL [39] and SOCL [26] focus on single or multi-task level scheduling for standalone or partitioned OpenCL applications. A common denominator of all aforementioned frameworks is the fact that they solely focus on non-managed applications, thereby leaving the area of managed languages unexplored. Exploiting multi-device concurrency and scalability via managed programming languages poses significant challenges due to the multi-level compilation approach of current frameworks, while creating further research opportunities due to the dynamic nature of managed languages and platforms. In this work, we explore multi-device concurrency and intelligent device selection in the context of managed languages by prototyping our proposed solution in the context of TornadoVM [12, 17].

2.2 TornadoVM

TornadoVM [12, 17] is a plug-in to OpenJDK and GraalVM that allows programmers to automatically accelerate Java programs on heterogeneous hardware. TornadoVM can target OpenCL-compatible devices and it runs on multi-core CPUs, dedicated GPUs (NVIDIA, AMD), integrated GPUs (Intel HD Graphics and ARM Mali), and FPGAs (Intel and Xilinx) [43, 44]. TornadoVM currently allows users to compose groups (called `TaskSchedules`) of multiple-tasks that can execute on hardware accelerators. However, these `TaskSchedules` can only target a single heterogeneous device, without allowing different tasks within a task-schedule to execute concurrently on various accelerators.

Listing 1. Example of the TornadoVM Task Parallel API for TaskSchedule with multiple Tasks

```
1  TaskSchedule filter = new TaskSchedule("blur")
2    .task("red", BlurFilter::compute, redFilter, image)
3    .task("green", BlurFilter::compute, greenFilter, image)
4    .task("blue", BlurFilter::compute, blueFilter, image)
5    .streamOut(redFilter, greenFilter, blueFilter)
6    .execute()
```

As an example, we implemented and evaluated a Blur filter application on TornadoVM. Listing 1 shows that the workload consists of three kernels, each operating independently on an `RGB` pixel of the input image. We evaluated the Blur filter application on commodity hardware equipped with three OpenCL-compatible devices: 1) a multi-core CPU (Intel Core i7-9750H), 2) an integrated GPU (Intel UHD Graphics 630), and 3) a discrete GPU (NVIDIA GeForce GTX 1650).

Since TornadoVM can only schedule all tasks within a `TaskSchedule` to execute on a single device, optimization opportunity is missed due to the lack of concurrency and under-utilization of the available devices in our experimental setup. Figure 2 depicts the evaluation results from running the Blur filter with two data sizes (1K and 4K images) across the three different devices: 1) running all tasks on the CPU, 2) running all tasks on the integrated GPU, and 3) running all tasks on the discrete GPU.

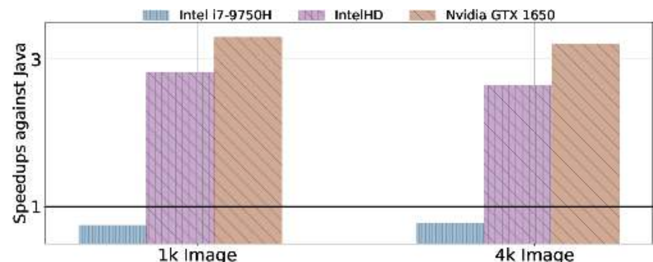
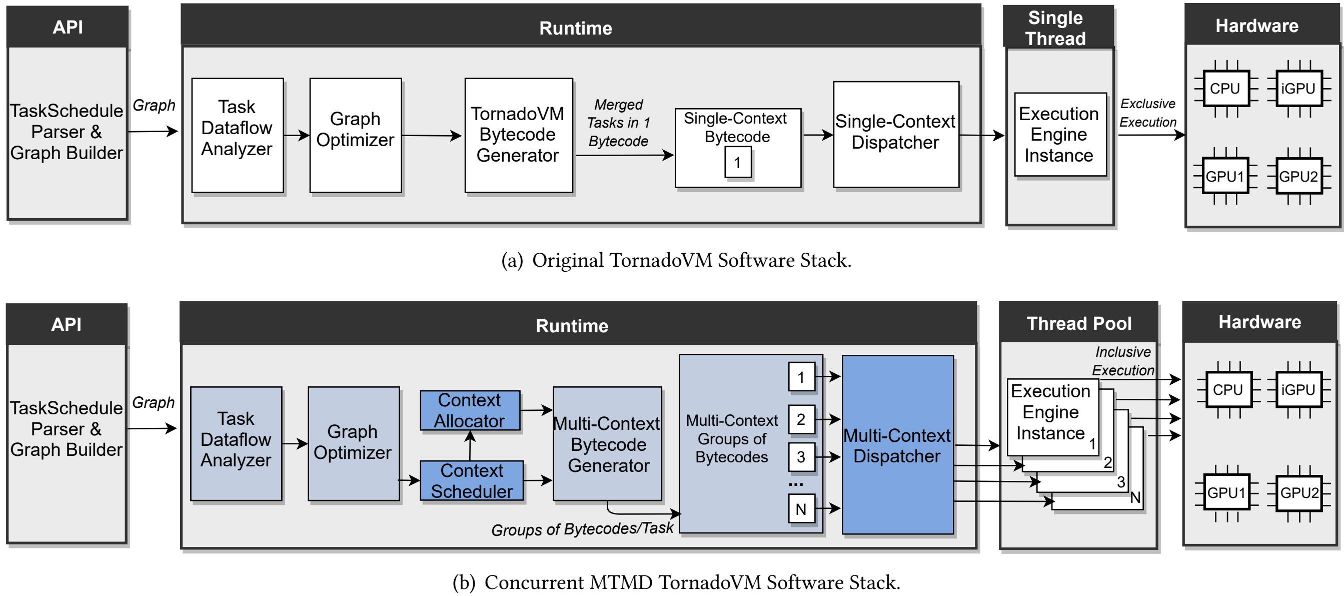


Figure 2. Achieved speedups against sequential Java for a CPU, an integrated GPU and a discrete GPU.

As shown in Figure 2, running all tasks on the discrete GPU yields the best performance for the Blur filter application up to $3.15x$. However, since the tasks are executed in-order, both the integrated GPU and the CPU remain idle without exploiting the potential performance through multi-device execution. To enable concurrent execution by allowing tasks within a `TaskSchedule` to execute on different devices simultaneously, we introduce the Multiple-Task Multiple-Device (MTMD) concurrent interpreter and execution mode in TornadoVM as explained in the following section.



(a) Original TornadoVM Software Stack.

(b) Concurrent MTMD TornadoVM Software Stack.

Figure 3. High-level overview of the components added and modified to the original TornadoVM to enable MTMD execution.

3 Multiple-Tasks on Multiple-Devices

To enable the `Multiple Tasks Multiple Devices(MTMD)` execution mode in TornadoVM, numerous key components have been modified or introduced. Figure 3 outlines both the original TornadoVM software stack (at the top), as well as the proposed modifications for enabling MTMD (bottom). As shown in Figure 3a, TornadoVM utilizes its own API to create `TaskSchedules` which are consequently parsed to create dataflow graphs that contain the various tasks. The graph is then analyzed and optimized during runtime and, in turn, a number of TornadoVM-specific bytecodes are generated. In the original TornadoVM, all the bytecodes that correspond to all the tasks of a particular `TaskSchedule` are enqueued in a single-context buffer, and are consequently dispatched for execution by a single instance of the execution engine. Therefore, all bytecodes, and consequently, all tasks of a `TaskSchedule` can only run on a single device at a time.

As shown in Figure 3b, to enable concurrent execution in TornadoVM, several components has been modified (light blue) or introduced (dark blue):

1. The `Task Dataflow Analyzer` and `Graph Optimizer` components, which are responsible for analyzing the dependencies between tasks and optimizing the graph, before scheduling them onto the devices, have been modified to enable concurrent execution.
2. The `Context Allocator` component that creates groups of dependent tasks has been introduced.
3. The `Context Scheduler` component that schedules dependent task groups onto devices has been also introduced.
4. The `Multi-Context Bytecode Generator`, which is an extension of the TornadoVM bytecode generator [17], is responsible for generating bytecodes for multiple target devices concurrently instead of a single one.
5. The `Multi-Context Dispatcher` has been introduced to assign bytecodes that belong to a task group to a particular execution engine instance for execution. The execution instances are implemented as a `thread-pool of execution engines` that run the TornadoVM interpreter with each one being responsible for executing a single context on a single device.

The following subsections describe in detail the aforementioned components.

3.1 Task Dataflow Analyzer and Graph Optimizer

As shown in the example of Listing 1, a `TaskSchedule` in TornadoVM can be composed of multiple tasks that may have data dependencies between them; i.e., the output of one task can be the input to another. Since developers can compose arbitrary `TaskSchedules`, the presence or the absence of dependencies between tasks is not guaranteed. Due to this fact, the original TornadoVM could only use a single device to execute a complete `TaskSchedule`. In order to enable concurrent execution of arbitrary tasks on different devices, we modified the Task Dataflow Analyzer and Graph Optimizer to extract inter-task dependencies.

While analyzing the tasks of a `TaskSchedule`, TornadoVM generates Java bytecodes for each task which are then transformed into a compiler graph based on the Intermediate Representation (IR) of the TornadoVM compiler. The dataflow analysis phase has been implemented as a compiler phase in the JIT Compiler. This phase detects the input and output arguments of the original tasks (Java methods). After the dependencies are identified, the task dependency graph

Listing 2. Example of TaskSchedule with multiple independent tasks.

```
1  TaskSchedule graph = new TaskSchedule("workload")
2   .task("t0",DFT::dft, inReal,inImag,outReal,outImag)
3   .task("t1",Blackscholes::bs,input,callPrice,putPrice)
4   .task("t2",MM::mm, matrixA, matrixB, matrixC, mmSize)
5   .streamOut(outReal,outImag,callPrice,putPrice,matrixC)
6   .execute();
```

is traversed in order to create a map of their accessibility within the different tasks of a `TaskSchedule`. Then, each input/output argument of each task is marked as `READ`, `WRITE` or `READ_WRITE` and stored as task meta-data information. This process is completed when the last task of the input `TaskSchedule` has been analyzed and evaluated correctly.

At the end of the dataflow analysis phase, the captured meta-data are used to create a Direct Acyclic Graph (DAG) of the intra-`TaskSchedule` dependencies. This information is used at a later stage for scheduling dependent tasks on the same device in order to avoid costly data copying of interim variables between devices. In contrast, independent tasks are grouped and scheduled independently for concurrent execution across numerous hardware accelerators.

In order to avoid tasks that are sharing read-only parameters to be grouped together, we implemented an optimization at the Graph Optimizer phase. The proposed optimization tackles `READ`-only dependencies between tasks by duplicating the `READ`-only parameters between tasks. In this way, tasks become independent and can be executed concurrently.

3.2 Context Allocator and Scheduler

Based on the task meta-data derived from the dataflow analysis and optimization phases, tasks can be grouped together or stay independent. Each group of a single task or multiple tasks will then be assigned to a device for execution via a device `context`. The notion of the context is to define an independent computational entity (a single task or a dependent task-group) that can target a device. As soon as contexts are defined, they also lock the allocated devices.

At this point, the scheduling of tasks on devices happens statically without taking into account specific task characteristics, such as memory accesses, parallel dimensions and single or double precision operations. Tasks are assigned onto the available devices in a `First Come First Served` order and they are inferred in the order they are attached on the `Taskschedule`. In addition, devices are ordered based on their characteristics and computational capabilities. In Section 4.4, we discuss in depth how we augment this scheduling approach by introducing predictive modeling based on the method features.

3.3 Multi-Context Bytecode Generator

Previous steps helped to reduce the computational granularity of a `TaskSchedule` to multiple contexts consisting of single or multiple inter-dependent tasks. At this point of execution, TornadoVM creates internal TornadoVM-specific bytecodes [17] that orchestrate the execution, the synchronization, and the data exchanges between the host and devices. The purpose of this extra virtualization layer is to abstract away from developers all the mechanics and details of hardware acceleration and kernel offloading. In the original TornadoVM, since tasks within a `TaskSchedule` could all execute on a single device, the bytecode generator creates single-context bytecodes destined to execute in-order on a particular device.

To exploit concurrent execution on devices, we augmented the existing virtualization layer to embed device selection control at the task-level (rather than in the original `TaskSchedule` level).

Listing 2 showcases three applications using the TornadoVM API, and grouped as independent tasks of the same `TaskSchedule`. These tasks are DFT, BlackScholes and Matrix Multiplication (MM). Initially, the dependency analysis marked them as independent and during context allocation with FCFS scheduling, all tasks have been assigned to the available devices.

As tasks are independent, the introduced multi-context bytecode generator generates three independent sets of bytecodes. Listings 3, 4 and 5 correspond to the generated multi-context bytecodes for tasks `t0`, `t1`, and `t2`, respectively.

The bytecodes of each context are assigned to a separate device (if three are present) awaiting interpretation and execution by TornadoVM.

3.4 Thread-Pool of Execution Engines

In order to execute the multi-context bytecodes introduced in this work in parallel, we introduce a scalable thread-pool of execution engines. Each of the execution engines is responsible for interpreting the bytecodes corresponding to a context assigned to a specific device, as shown in Figure 3b. These bytecodes can contain up to several tasks with or without dependencies among them.

Each of the execution engines deploys an isolated instance of the interpreter per device that executes the multi-context bytecodes assigned to it. At this stage, following the original TornadoVM execution flow, tasks can be dynamically compiled to OpenCL and the execution engines can access binaries from a global code cache. The interpreter itself can be JIT compiled by the underlying JVM (e.g., Oracle HotSpot) to improve performance. Note that the TornadoVM bytecodes only orchestrate the execution between the accelerators and the host machine and do not perform the actual computation. The latter is achieved by executing the generated OpenCL code via the device driver.

Another benefit of reducing the granularity of the execution from a `TaskSchedule` to smaller groups of tasks composing a context, is the ability to increase the resiliency of the

Listing 3. Bytecodes for t0 (DFT)

```
1 BEGIN <0> //New context [device 0]
2 COPY_IN <0, bi1, in> //Copies <in>
3 COPY_IN <0, bi2, in> //Copies <in>
4 COPY_IN <0, bi3, in> //Copies <in>
5 COPY_IN <0, bi4, in> //Copies <in>
6 LAUNCH <0, bi5, @dft, in, temp>
7 COPY_OUT_BLOCK <0, bi6, out> //C-out
8 COPY_OUT_BLOCK <0, bi7, out> //C-out
9 END <0> //Ends context
```

Listing 4. Bytecodes for t1 (BlackScholes)

```
1 BEGIN <1> //New context[device 1]
2 COPY_IN <1, bi1, in> //Copies <in>
3 ALLOC <1, bi2, out> //Allocates <out>
4 ALLOC <1, bi3, out> //Allocates <out>
5 LAUNCH <1, bi4, @bs, temp, out>
6 COPY_OUT_BLOCK <1, bi5, out> //C-out
7 COPY_OUT_BLOCK <1, bi6, out> //C-out
8 END <1> //Ends context
9 ---
```

Listing 5. Bytecodes for t2 (MM)

```
1 BEGIN <2>//New context [device 2]
2 COPY_IN <2, bi1, in> //Copies <in>
3 COPY_IN <2, bi2, in> //Copies <in>
4 COPY_IN <2, bi3, in> //Copies <in>
5 LAUNCH <2, bi4, @mm, in, temp>
6 COPY_OUT_BLOCK <2, bi5, out> //C-out
7 END <2> //Ends context
8 ---
9 ---
```

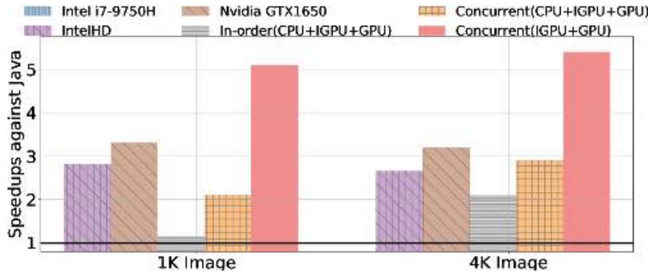


Figure 4. Concurrency limits

execution by enabling fault tolerance which in turn reduces the cost of re-execution.

3.5 Discussion

In order to assess the performance benefits of enabling scalable execution across devices within the same compute system, we revisited the Blur filter application of Listing 1. In our revised experiments, we enabled the concurrent execution of the independent tasks of the Blur filter application using the First-Come-First-Serve (FCFS) scheduling scheme. Figure 4 adds three additional data points to Figure 2 which correspond to three additional execution scenarios: a) In order execution of all tasks on the CPU, integrated GPU (IGPU), and discrete GPU (grey bar), b) Concurrent execution of all tasks across all devices (first running on the CPU, second on the IGPU, and third on the discrete GPU - orange bar), and c) Concurrent execution of all tasks across two devices (first two on the discrete GPU, and third on the IGPU - red bar).

As shown in Figure 4, the additional execution scenarios can influence dramatically the performance which can be up to 2x higher compared to running the whole `TaskSchedule` on the same device. However, the problem of statically deciding which policy to employ for scheduling fine-grained tasks across the available accelerators is very challenging, due to the diverse characteristics and performance of each task. To enable efficient scheduling that takes into consideration both device availability, the potential of concurrent execution, and code characteristics, we employ a ML-based scheduling technique described in the next section.

4 Prediction Based Scheduling for MTMD

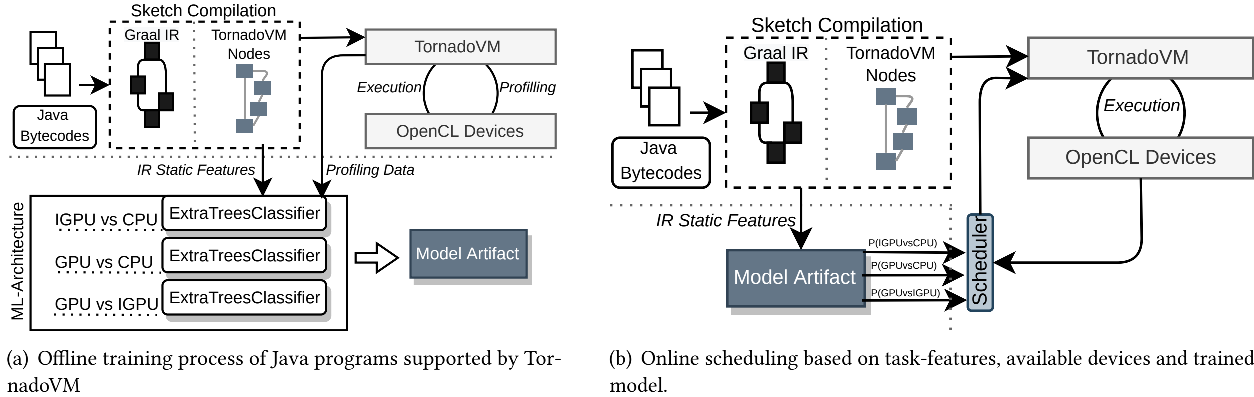
Section 3 outlined the required runtime support for a heterogeneous managed runtime to efficiently handle the orchestration of dispatching multiple tasks on multiple devices concurrently. However, to fully utilize the capabilities of such a system and be able to perform an efficient task/device allocation, in terms of performance, a fast and accurate scheduling policy is required. To that end, we integrated a ML model, trained to perform device-task allocation, that governs our scheduling policy.

A decisive factor in our scheduling strategy is the detection of the best computing device for a given task in terms of performance. Our study focuses on commodity personal computers, due to the wide set of heterogeneous hardware available. This includes a CPU, an Integrated GPU and Discrete GPUs. To train the ML-model, we extract a set of features describing an application from the compiler IR (Graal IR [15]) before generating the OpenCL kernel for a given task. Graal IR is in a graph form, and represents Sea of Nodes [13] (control flow and data flow). Consequently, we use a Multiple-Classifier-System (MCS) to determine the optimal mapping. Each component of this system is a tree-based two-class classifier, trained to compute the probability at which a specific task will exhibit speed-up when executed on one device over another. The final decision is made through the conjugation of the output probabilities of the aforementioned learners. The following subsections describe in detail the components of the proposed ML-based scheduling policy for MTMD.

4.1 Feature Extraction

Being able to extract meaningful application characteristics is a crucial factor for effectively predicting which task will perform better across different devices. Prior work, discussed in detail in Section 6, proposed methodologies for extracting code features directly from OpenCL kernels. Such an approach is not suitable for our work due to the two-stage compilation that TornadoVM employs (from Java to OpenCL C, and from OpenCL C to binary code). Hence, we perform feature extraction from the compiler IR graph during JIT compilation, ensuring that sufficient information is captured for characterizing the behavior of both the Java and the auto-generated OpenCL programs.

When a task is assigned to a `TaskSchedule`, Java bytecodes are transformed to the compiler's IR and that stage we extract the code features. This is achieved by adding a `Feature extraction phase` in the TornadoVM JIT compiler to obtain the number and type of operations based on individual nodes. The design choice of obtaining features directly from the IR, and before code generation, adds modularity to our system since it can cater other backend or pure



(a) Offline training process of Java programs supported by TornadoVM

(b) Online scheduling based on task-features, available devices and trained model.

Figure 5. Offline training process and Online device allocation based on pre-trained model.

x86 execution through Java. The extracted code features are later combined with runtime information regarding the input/output data sizes, number of threads to be deployed, and inter-task dependencies.

4.2 Feature Selection

The initial feature set consists of 26 distinct features which are pre-processed and combined in order to construct new features that have greater predictive ability than the initial ones. During this process the feature set is further expanded to also include interaction features, i.e., features that are computed as the pairwise product of the existing ones. Furthermore, features that are the most relevant to each other (e.g., `float_math_function`, `integer_math_function`) are grouped together.

Upon completion of the feature engineering process, the dimensionality of the data is increased considerably. In such cases, it is beneficial to select only those features that are considered to be the key attributes for the model. This enables the learning algorithm (discussed in Section 4.4) to focus only on the most important variables. Also, this allows us to avoid modelling any underlying noise in the data induced by irrelevant features. The criterion that was used to compute the features' importance is the Gini importance [16]. Based on this criterion, the ten features that influence more the final outcome per classifier are depicted in the Hinton diagram of Figure 6. The sizes of the squares represent the magnitude of the value; i.e., the corresponding Gini importance of each feature.

4.3 Training Dataset

The dataset consists of static code features of various kernels, as well as their execution times on the three available devices, i.e, CPU, IGPU, GPU. Based on these timings the following ratios are computed: $\frac{IGPU execution time}{CPU execution time}$, $\frac{GPU execution time}{CPU execution time}$ and $\frac{GPU execution time}{IGPU execution time}$.

The time ratios are then turned into binary target variables indicating whether the specific task has speedup on a given device. More specifically, ratios lower than 1.0 indicate slowdown and so they are mapped to 0, while ratios above the same threshold correspond to speed up, and consequently are mapped to 1. Each of these binary variables will serve as the target for a classifier in our multiple-classifier-system.

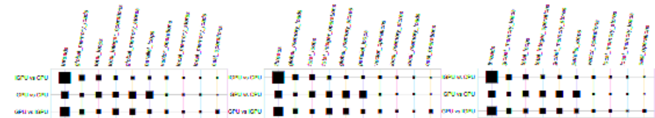


Figure 6. Feature importance for classifiers: 1) IGPU vs GPU, 2) GPU vs CPU and 3) GPU vs IGPU. Squares are representing the impact of the feature in the final decision (large squares have the most influence).

Regarding the specific program selection, we used kernels from the benchmark suite and examples that already exist in the TornadoVM repository. Figure 5a showcases the offline process for collecting the data and training our model. As we opt-in for feature extraction through the IR (generated from the original Java methods), we trained our model purely with Java benchmarks compatible with TornadoVM. Hand-tuned OpenCL programs will result in a different performance pattern compared to the OpenCL automatically generated from Java. Thus, extending the training set with benchmark suites purely written in OpenCL will negatively influence or bias the accuracy of our predictor. We execute these programs for various input configurations, depending on their computational intensity, on an Intel CPU, an Intel HD Graphics and an Nvidia GTX 1650. For each individual data point (i.e., application's features, input size and achieved speedup) we use the existing profiling infrastructure to extract all profiling-events at the OpenCL side, as well as runtime dynamic information and overheads present

Table 1. Scheduling device selection truth table.

Classifier			Target
IGPU vs CPU	GPU vs CPU	GPU vs IGPU	Device
0	0	0/1	CPU
1	0	0/1	IGPU
0	1	0/1	GPU
1	1	0	IGPU
1	1	1	GPU

in the Java side. Overall, we train our ML model with more than 200 data points.

4.4 Machine Learning Architecture

Our ML architecture consists of the training model and three different classifiers running in parallel.

Training Model: Our training model uses three Extremely Randomized Trees (ExtraTrees) [20] classifiers. Each classifier produces a speedup probability for each task between the following pairs: IGPU\CPU (1st classifier), GPU\CPU (2nd classifier) and GPU\IGPU (3d classifier). Among the available tree-based algorithms, such as Decision Trees, Random Forest and Extremely Randomized Trees, the latter was selected due to its ability to better handle overfitting. The hyper-parameters of the model (i.e., estimators, maximum depth) were optimized by searching over a grid of trials and the combination that yielded the best cross validation score (10-fold) was retained. Moreover, by investigating the training dataset for each classifier, it was found that the datasets of the first and the second classifiers, were highly imbalanced, i.e., the target classes were unequally represented and thus the models would ignore, and in turn, underperform on the minority class. To tackle this issue, the SMOTE algorithm [9] was used which upsamples the minority class by synthesizing new examples.

1st Classifier: The chosen ExtraTrees classifier, i.e., the one that yielded the best cross-validation score, fits 100 estimators with maximum depth set to 50. The first level of prediction considers only the IGPU and the CPU and attempts to determine the most suitable device between them for a given task. The output of the model is the probability at which the given task will have speedup when executed on the IGPU instead of the CPU. By selecting an appropriate threshold, the probabilistic output can then be interpreted as class labels, i.e., IGPU or CPU.

For this selection, the Receiver Operating Curve (ROC)[7] and the Precision-Recall Curve[6] were plotted for various candidate thresholds in order to better understand the trade-off in performance at the various levels. Given the imbalanced nature of our dataset, we optimized for F1-score, i.e., the harmonic mean of precision and recall, instead of accuracy, since the former serves as a better measure of the incorrectly classified cases. For the first classifier, the optimal threshold was determined to be around 0.2 resulting in 0.95 F1-score on the held-out dataset.

Table 2. Experimental Testbed.

Hardware	
Processor	Intel Core i7-9750H CPU @ 2.60GHz
Cores	6 (12 HyperThreads)
RAM	32GB
Integrated-GPU	Intel UHD Graphics 630
Discrete GPU	NVIDIA GeForce GTX 1650 (Turing) 4GB GDDR5, 896 CUDA Cores
Software	
Operating System	Ubuntu 20.04 (Kernel 5.4.0-52-generic)
OpenCL (CPU)	2.1 Device Version
OpenCL (IGPU)	2.1 Device Version
OpenCL (GPU)	1.2 Device Version
CUDA Driver	450.80.02
TornadoVM	v0.7
JVM	OpenJDK 1.8.0_262 with JVMCI
Java Heap	-Xmx22G -Xms22G

2nd Classifier: For the second classifier, the optimal performance was achieved by fitting 500 estimators with maximum depth set to 10. In a similar way, the second classifier is trained to distinguish between tasks based on their relative performance on either the discrete GPU or the CPU. Again, the probabilistic output is turned into a class label, i.e., GPU or CPU. The optimal threshold is determined to be approximately 0.6 with 0.96 F1-score on the held-out dataset.

3rd Classifier: Lastly, the third ExtraTrees classifier fits 50 estimators while the maximum depth is set to 50. The third classifier aims to select between IGPU and GPU. With the same process, the best threshold is defined around 0.6 resulting to 0.91 F1-score on the held-out dataset.

4.5 On-Line Scheduling

Figure 5b outlines the on-line scheduling process that performs the inference using the trained model. During runtime, the trained ML model is invoked along with a JSON file that contains the features of a task eligible to run on the system. Note that the time for the model inference does not exceed 60 ms. These features consist of inputs to the multiple-classifier-system which outputs the three aforementioned probabilities. By setting the thresholds discussed in Section 5.3, we convert the probabilities into class labels, i.e., 0 for slowdown and 1 for speedup. The final decision was taken by using the truth table presented in Table 1. Specifically, the following scenarios are considered for each task:

- **Schedule on CPU:** If predicted to have slowdown on both IGPU and GPU compared to CPU.
- **Schedule on IGPU:** a) If predicted to have slowdown on GPU and speedup on IGPU compared to CPU, or b) if predicted to have speedup on IGPU and GPU compared to CPU and on IGPU compared to GPU.
- **Schedule on GPU:** a) If predicted to have slowdown on IGPU and speedup on GPU compared to CPU, or b) if predicted to have speedup on IGPU and GPU compared to CPU and on GPU compared to IGPU.

Table 3. The Applications.

Group	Application	Description
1	DFT [22]	Hierarchical mixed radix FFT algorithms for both power-of-two and non-power-of-two sizes.
	Black-Scholes [23]	Option pricing using the Black-Scholes merton process.
	Matrix Multiplication [51]	Matrix multiplication on square matrices.
2	NBody [46]	Particle simulations.
	MonteCarlo [47]	Monte Carlo simulation for option pricing models.
	RenderTrack [38]	Parallel kernel for image decomposition that contains multiple control flow operations.
	Mandelbrot [27]	Iterative function applied in a large set of points.
	Hilbert Matrix [41]	Dense matrix computation on a square matrix.
3	Matrix Transpose [51]	Matrix transpose operation on a square matrix.
	B&W Filter [31]	A filter that converts an RGB image to Grayscale.
	Convolution [3]	A two dimensional process of adding each element of an image to its local neighbors.
	Euler Method [18]	A first-order numerical procedure for solving ordinary differential equations (ODEs).

Table 4. The input data sizes for each application (task) in three different ranges: small, medium and large.

	DFT	BS	MM	NBody	MC	RT	Mandelbrot	Hilbert	MT	B&W	Conv	Euler
Low	1024	65536	65536	1024	65536	262144	262144	65536	65536	1K img	16384	512
Medium	16384	524288	262144	2048	524288	1048576	1048576	262144	262144	2K img	262144	1024
Large	65536	1048576	1048576	8192	1048576	16777216	4194304	1048576	1048576	4K img	1048576	4096

5 Evaluation

This section presents the experimental evaluation of the proposed MTMD mechanism that enables the seamless and concurrent execution of multiple tasks on multiple hardware accelerators. We first describe the experimental setup and the methodology, as well as the applications used to assess the performance. Finally, we present and discuss the results on concurrent device execution and scheduling.

5.1 Experimental Setup and Methodology

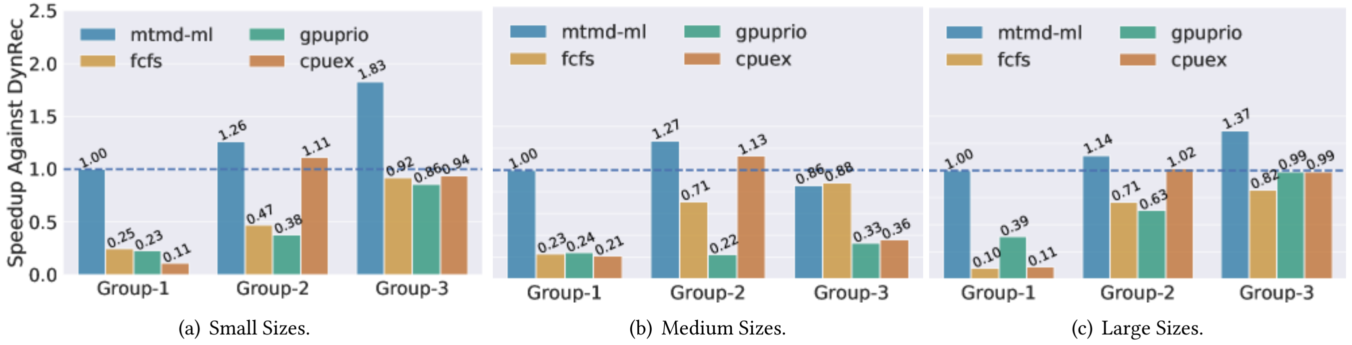
To assess the performance, we used an experimental setup equipped with an Intel CPU, an Intel integrated GPU and a discrete Nvidia GPU. Essentially, this configuration corresponds to a commodity machine with a high compute capacity, which can be seamlessly utilized by a Java application via the MTMD execution mode. Table 2 outlines the hardware and software characteristics of our testbed.

Regarding the experimental methodology, we follow the approach outline in [19]. Initially, we perform a warm-up phase for every application to stabilize the performance of the JVM. The warm-up phase ensures that the Java code of each application is JIT-compiled, and in our case 100 iterations was a sufficient number to achieve this. Once the warm-up phase is complete, we run each application for 10 consequent times and we report the mean of the obtained total execution times, including the time spent for the model inference.

5.1.1 Applications and Input sizes. To evaluate the proposed MTMD mechanism we use twelve applications that can be classified as compute intensive, memory intensive and control-flow intensive. Our goal has been to assess MTMD by running all the applications concurrently. However, the inability of TornadoVM to support data transfers, from the host to the various devices, of sizes over 1 GB, led us to split our total workload of twelve applications into three groups (`Groups 1 to 3`), as shown in Table 3. Each group has a randomly assigned number of applications that can be concurrently executed for different input data sizes (`small`, `medium` and `large`). Table 4 presents the input data sizes for each application.

5.1.2 Scheduling Strategies. For a full coverage of the evaluation of the MTMD mechanism, we employ the following scheduling policies:

1. ***Dynamic Reconfiguration (DynRec) [17]:*** This is the official scheduling policy supported by TornadoVM, in which it examines all the viable configurations exhaustively. Thus, tasks have to be executed serially on all devices to select the highest performing one After the exhaustive execution is performed, TornadoVM stores the winning device and uses it again for further invocations of the same code. However, slight changes to the executed code or input data sizes will trigger again the exhaustive execution.
2. ***First-Come-First-Served (FCFS):*** Tasks are scheduled to run on devices following the order that the TornadoVM system discovers the device drivers. Tasks will be allocated to devices in the order that they arrive with respect in the order that OpenCL device drivers are discovered by the system.
3. ***GPU-Priority (gpuprio):*** Tasks are scheduled to run on devices following a score that ranks the devices based on their compute capabilities, in our system the discrete GPU is the one with the highest compute capabilities.



(a) Small Sizes. (b) Medium Sizes. (c) Large Sizes.

Figure 7. Achieved speedups for each group of applications and size configurations against the baseline Dynamic Reconfiguration (*DynRec*) for consecutive execution. Each bar presents the following policies: ML-based MTMD (*mtmd-ml*), First-Come-First-Served *fcfs*, GPU Priority (*gpuprio*), and CPU Exclusion (*cpuex*).

4. ***CPU-Exclusion (cpuex):*** Tasks are scheduled to run on devices (except CPUs) following the order that the TornadoVM system discovers the OpenCL device drivers.
5. ***ML-based MTMD (mtmd-ml):*** Tasks are scheduled and dispatched to run on devices with respect to our proposed ML-based scheduler (discussed in Section 4).
6. ***Oracle:*** This scheduling strategy presents the device-task allocation that offers the best performance. This strategy is obtained by offline exhaustive exploration of the complete optimization space.

The *Dynamic Reconfiguration* policy is the only policy that requires all the tasks within a TaskSchedule to be executed on a single device due to the *Single-context Dispatcher* in the original TornadoVM system (Figure 3a). On the contrary, the remaining scheduling policies exploit the MTMD mechanism and can operate concurrently on multiple devices. Additionally, note that the *Dynamic Reconfiguration* and the *Oracle* scheduling policies are used mainly to set the peak performance for the consecutive (single-context) and the concurrent (multi-context) executions of the experimental benchmarks, as they introduce a significant cost that makes them unsuitable for real-time execution.

5.2 Performance Evaluation of MTMD

This section is split into two parts. Section 5.2.1 discusses the performance of all scheduling policies that operate with the MTMD execution mode against the best consecutive execution policy which is *Dynamic Reconfiguration*. On the other hand, Section 5.2.2 compares the MTMD scheduling policies against *Oracle*, the best concurrent execution policy.

5.2.1 Relative Performance vs Best Consecutive. Figure 7 compares the performance of the *fcfs*, *gpuprio*, *cpuex* and *mtmd-ml* policies against *DynRec* for different data sizes (small, medium, large). We use the *DynRec* policy as baseline as it results in the best execution plan for consecutive execution. The highest performance increase for each data size is observed for the *mtmd-ml* policy at 1.83*x* (Figure 7a - Group-3), 1.27*x* (Figure 7b - Group-2), and 1.37*x* (Figure 7c - Group-3) for small, medium and large sizes, respectively.

As shown in Figure 7, the *mtmd-ml* policy exhibits the higher performance across all data sizes and all groups of applications. The reason is that this policy leverages the ML trained model to capture a large space of factors that can influence performance. In addition, there are cases that the consecutive execution on a single device (*DynRec* - baseline) results in higher performance than the concurrent execution on multiple devices with *fcfs*, *gpuprio*, or *cpuex*. For instance, Figure 7a shows that the applications in Group-1 can run significantly faster when they are executed consecutively on the Nvidia GPU rather than being concurrently executed across all available devices. The reason is that each application in Group-1 (i.e., DFT, BlackScholes and Matrix Multiplication) is compute intensive and performs an order of magnitude faster on the Nvidia GPU than the other devices. Thus, the *fcfs*, *gpuprio*, or *cpuex* concurrent scheduling policies fail to outperform the baseline for these cases. On the contrary, *mtmd-ml* can achieve the performance of the baseline, as it accounts the single-context scenario during the training of the ML model. The only case that the *mtmd-ml* policy performs lower than the baseline is the medium size for Group-3 (Figure 7b). In this case, the trained ML model mispredicts and schedules the execution of the most compute intensive task (i.e., NBody) in the small GPU (Intel UHD Graphics 630). Section 5.3 discusses the performance and precision analysis of our trained model in more detail.

Additionally, the remaining policies (*gpuprio*, *fcfs* and *cpuex*) show a diverse performance behavior for the three groups of applications when running on the same data sizes. This indicates that the diversity across the applications that belong in the same group is high, and therefore, some of them can perform better in a GPU, while others can perform better in a CPU. For instance, Group-1 shows that the baseline outperforms all the remaining policies (i.e., *gpuprio*, *fcfs* and *cpuex*). The reason is that the applications in this group

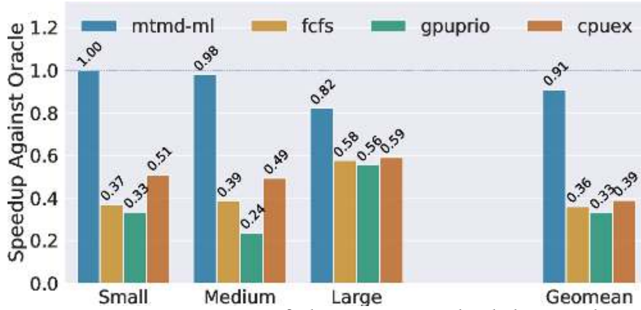


Figure 8. Comparison of the MTMD scheduling policies against the Oracle (peak performance).

are all compute intensive and achieve high speedups when they are executed on the discrete GPU.

Group-2 exhibits higher performance than the baseline when the applications in this group are executed exclusively on the same GPUs (*cpuex* - orange bars), reaching up to $1.13x$ for medium size (Figure 7b). On the other hand, the performance of the *gpuprio*, *fcfs* and *cpuex* policies when running Group-3 is at the same range. In particular, a $0.08x$ performance difference is noted between *gpuprio* and *cpuex* for small size (Figure 7a), while a $0.17x$ difference is observed between *fcfs* and *gpuprio*/*cpuex* for large sizes (Figure 7c). However, for medium sizes, *fcfs* achieves the highest performance among the MTMD policies, indicating that the GPUs are not the most suitable devices to execute for this range.

Finally, it is shown that the MTMD concurrent execution in conjunction with the ML-based scheduling policy (*mtmd-ml*) can increase the performance up to 83% compared to the consecutive execution (*DynRec*).

5.2.2 Relative Performance vs Best Concurrent. To assess the performance of the MTMD scheduling policies against the maximum performance that can be achieved, we decided to expand our experiments with an Oracle implementation. Therefore, we evaluate the *mtmd-ml*, *fcfs*, *gpuprio* and *cpuex* policies against the *Oracle* policy. *Oracle* represents the peak performance that can be achieved, as it is derived from the exhaustive exploration of all possible concurrent execution plans of each group of benchmarks on the available hardware devices. Note that the diversity across the applications, along with the various data sizes, increases the exploration space significantly, and therefore, the decision of the *Oracle* policy may not be pragmatic for real applications. In fact, the execution of the applications in Group-2 for the large sizes takes 4.5 hours. Nonetheless, *Oracle* is the best baseline to compare the performance of the MTMD policies in terms of the concurrent execution.

The left side of Figure 8 presents the comparative evaluation of the MTMD policies against *Oracle* for small, medium and large data sizes, while the right side depicts their geometric mean. As Figure 8 shows, *mtmd-ml* is the best performing policy reaching up to 91% of the *Oracle*'s performance in average, followed by *cpuex* (39%) and *fcfs* (36%). The lowest average performance is observed for the *gpuprio* policy, due to the low performance of GPUs when running for small and medium data sizes.

5.3 Analysis of the MTMD ML Model

This section presents an analysis of the performance and successful task-device allocation of the trained MTMD machine learning model. In particular, we use the *area under the ROC curve (AUC)* and the *F1-score* as metrics for performance evaluation. The *AUC* is calculated as the integral of the ROC with respect to the false positive rate over [0, 1]. In essence, high *AUC* indicates better prediction of the model.

Figure 9 presents the obtained *AUC* for the three classifiers that we used in our model, as introduced in Section 4.4. In particular, the micro-average ROC that classifies the execution between two different types of devices is 0.94 (Figure 9a), 0.97 (Figure 9b) and 0.82 (Figure 9c) for the first, second and third classifier, respectively. Based on this metric, the second classifier (GPU-CPU) has the best performance, followed by the first (IGPU-CPU) and the third (GPU-IGPU) classifiers. This behavior is also verified by closely investigating the confusion matrices in Table 5, which shows that the third classifier mispredicted the IGPU over the GPU in four out of 31 times. In fact, this is the cause of the misprediction that resulted in the low performance of Group-3 when *mtmd-ml* was used (Figure 7b), as the model decided to use the Intel Integrated GPU instead of the Nvidia GPU.

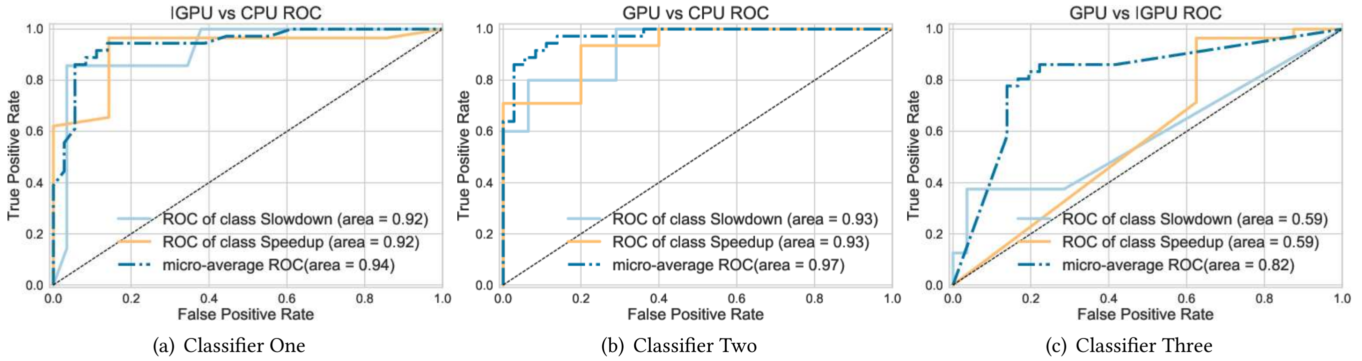
However, the overall decision of the model is not severely influenced as the final outcome on which device to execute is taken based on the combination of all classifiers. Finally, based on the confusion matrices (Table 5), the *F1-score* (i.e., the harmonic mean of precision and recall), was computed for each classifier using the following formula: $g(x) = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$. The final *F1-scores* are 0.95, 0.96 and 0.91 for the first, second and third classifier, respectively.

6 Related Work

We have classified the related work in the following groups. The first group discusses works that apply non-predictive task scheduling, while the second discusses predictive task scheduling. The final group elaborates on works that allow single tasks on multiple devices.

Non-ML Multi-Task Scheduling: Many works focusing on single or multi-task scheduling for standalone or partitioned OpenCL applications, such as VirtCL [53], SnuCL [34], PySchedCL [21], FluidiCL [42], MultiCL [2], EngineCL [39] and SOCL [26]. Our prime difference is that we exploit this opportunity of concurrent execution on heterogeneous hardware for Java, seamlessly.

Parravicini *et al.* [45] use the GrCUDA polyglot API and employ a custom scheduling approach to allow multiple polyglot tasks to be scheduled on a single GPU at runtime.



(a) Classifier One (b) Classifier Two (c) Classifier Three

Figure 9. Offline training process and Online device allocation based on pre-trained model.

Table 5. The confusion matrix of each classifier.

Classifier One

	Actual IGPU (1)	Actual CPU (0)
Predicted IGPU (1)	28	1
Predicted CPU (0)	1	6

Classifier Two

	Actual GPU (1)	Actual CPU (0)
Predicted GPU (1)	31	0
Predicted CPU (0)	2	3

Classifier Three

	Actual GPU (1)	Actual IGPU (0)
Predicted GPU (1)	27	1
Predicted IGPU (0)	4	4

This work exploits pace-sharing and overlaps the time spent in transferring data with the execution, if possible. Our work focuses on scheduling multiple tasks into multiple devices from different vendors, although it can be used to schedule concurrently on a single device.

ML-based Multi-Task Scheduling: Troodon [33] is a load-balancing scheduling heuristic that classifies OpenCL applications as suitable for CPU or GPU execution, based on a speedup predictor. The Qilin [37] compiler uses offline profiling to create a regression model for predicting the execution time of input applications. Ogilvie *et al.* [40] introduce a low-cost predictive model for the automatic construction of heuristics that reduce the training overhead for execution on CPU-GPU equipped platform. Furthermore, Grewe *et al.* [25] leverages predictive modelling to influence the OpenCL code generation from OpenMP programs when speedups are predicted. Additionally, Chen *et al.* [10] combine generic search with learning and benchmarking to find good scheduling methods for execution on heterogeneous hardware, including CPUs, server GPUs, mobile GPUs, and FPGA-based accelerators. However, the supported scheduling mechanism is semi-automated, as the search space must be manually defined by a programmer for each algorithm similar to a template. Wen *et al.* [52] show that the concurrent execution of OpenCL kernels can increase the GPU utilization and improve performance. This is achieved by applying a decision tree based prediction model to determine whether an application kernel should be scheduled individually or along with other kernels. Baldini *et al.* [5] use existing OpenMP applications and supervised learning to predict the potential GPU execution speedup among different vendors. Brown *et al.* [8] present a model that allows to get accurate predictions of speedups using a small set of features, while also being portable portability across Nvidia GPUs with different capabilities. Adams *et al.* [1] propose a novel scheduling algorithm for the Halide programming language that targets image processing pipelines. Their model combines symbolic analysis with machine learning to predict performance.

Single Task Scheduling on Multiple-Devices: Other studies have combined predictive modelling and scheduling for single task/application partitioning onto multiple devices. Kofler *et al.* [35] use an Artificial Neural Network to dynamically partition a given task in two parts, one that operates on a CPU and a second that operates on a GPU. This partition is done through the Insieme [32] that transforms the code from single kernel into multiple kernels. Grewe *et al.* [24] present a system that combines a two-level predictor with supervised learning models (i.e., Support Vector Machines) to partition tasks for hybrid CPU-GPU execution based on their static code features. Also, Singh *et al.* [48] present a runtime system that performs energy efficient mapping and repartitioning of threads of each application between CPU and GPU of an MPSoC, while taking into account the execution time.

The main differentiation point of our work with prior is that we enable the seamless and intelligent mapping of multiple tasks onto multiple devices from Java. Therefore, programmers can remain oblivious of the actual hardware device that their programs will run, while leveraging a predictive machine learning model that can effectively schedule the execution on the most suitable device based on knowledge extracted from the Graal IR.

7 Conclusions

In this work, we presented a Multiple-Tasks on Multiple-Devices (MTMD) mechanism capable of performing seamless concurrent heterogeneous execution of Java programs. We

implemented this mechanism by extending the virtualization layer of TornadoVM along with additional components for task dependency extraction. Besides, we used code features extracted directly from the compiler's IR as well as a custom ML-architecture to predict the device allocation with the highest projected speedup. To the best of our knowledge, this is the first paper that allows concurrent heterogeneous execution for programs purely written in Java.

Besides, we have presented a scalable and modular system that employs custom parallel bytecode interpreters that can utilize multiple devices, while using intelligent resource allocation. Also, we introduced an online scheduling approach based on a ML-architecture of multiple classifiers, while using code features collected at compile and at run time.

We evaluated our mechanism with ML-based scheduling against the best single device and various concurrent scheduling policies. Our approach exhibits performance improvements of up to 83% compared to the best single device while reaching up to 91% of the oracle performance.

For future work, we plan to extend our ML-architecture to be able to make decisions among different compiler backends (e.g., PTX, SPIR-V, x86) to ensure optimal device and architecture allocation for each application. Therefore, in the future we expect our system to be able to seamlessly offload workloads concurrently on multiple devices, while leveraging the optimal programming construct for each architecture.

Acknowledgments

The work presented in this paper is partially funded by grants from Intel Corporation and the European Union's Horizon 2020 E2Data 780245 and ELEGANT 957286 projects.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4 (2019). https://doi.org/10.1145/3306346.3322967

[2] Ashwin M. Aji, Antonio J. Pe na, Pavan Balaji, and Wu–chun Feng. 2016. MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL. *Parallel Comput.* 58 (2016), 37 – 55. https://doi.org/10.1016/j.parco.2016.05.006

[3] Shams A. H. Al Umairy, Alexander S. van Amesfoort, Irwan D. Setija, Martijn C. van Beurden, and Henk J. Sips. 2012. On the Use of Small 2D Convolutions on GPUs. In *Computer Architecture (ISCA)*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-24322-6_6

[4] AMD. Accessed in 2020. Aparapi project. https://aparapi.github.io/

[5] Ioana Baldini, Stephen J. Fink, and Erik Altman. 2014. Predicting GPU Performance from CPU Runs Using Machine Learning. In *Proceedings of the IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. https://doi.org/10.1109/SBAC-PAD.2014.30

[6] Kendrick Boyd, Kevin H. Eng, and C. David Page. 2013. Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals. In *Proceedings of the 2013th European Conference on Machine Learning and Knowledge Discovery in Databases (ECMLPKDD) - Volume Part III*. Springer-Verlag. https://doi.org/10.1007/978-3-642-40994-3_29

[7] Andrew P. Bradley. 1997. The Use of the Area under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Pattern Recogn.* 30, 7 (1997). https://doi.org/10.1016/S0031-3203(96)00142-2

[8] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. 2021. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2021). https://doi.org/10.1145/3431731

[9] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority over-Sampling Technique. *J. Artif. Int. Res.* 16, 1 (2002).

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] James Clarkson, Juan Fumero, Michail Papadimitriou, Maria Xekalaki, and Christos Kotselidis. 2018. Towards Practical Heterogeneous Virtual Machines. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. https://doi.org/10.1145/3191697.3191730

[12] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Lujan. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang)*. https://doi.org/10.1145/3237009.3237016

[13] Click, Cliff and Paleczny, Michael. 1995. A Simple Graph-Based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR)*. https://doi.org/10.1145/202529.202534

[14] Shane Cook. 2012. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* (1st ed.). Morgan Kaufmann Publishers Inc.

[15] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL)*. https://doi.org/10.1145/2542142.2542143

[16] Frank A. Farris. 2010. The Gini Index and Measures of Inequality. *The American Mathematical Monthly* 117, 10 (2010), 851–864.

[17] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3313808.3313819

[18] V. M. Garcia, A. Liberos, A. M. Climent, A. Vidal, J. Millet, and A. González. 2011. An adaptive step size GPU ODE solver for simulating the electric cardiac activity. In *Computing in Cardiology*. 233–236.

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (2007). https://doi.org/10.1145/1297105.1297033

[20] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely Randomized Trees. *Mach. Learn.* 63, 1 (2006). https://doi.org/10.1007/s10994-006-6226-1

[21] Anirban Ghose, Siddharth Singh, Vivek Kulaharia, Lokesh Dokara, Srijeeta Maity, and Soumyajit Dey. 2020. PySchedCL: Leveraging Concurrency in Heterogeneous Data-Parallel Systems. arXiv:2009.07482 [cs.DC]

[22] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. 2008. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. 1–12. https://doi.org/10.1109/SC.2008.5213922

[23] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating Financial Applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. https://doi.org/10.1145/2458523.2458536

[24] Dominik Grewe and Michael F. P. O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Compiler Construction*, Jens Knoop (Ed.). Springer Berlin Heidelberg.

[25] D. Grewe, Z. Wang, and M. F. P. O'Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2013.6494993

[26] Sylvain Henry, Denis Barthou, Alexandre Denis, Raymond Namyst, and Marie-Christine Counilh. 2013. *SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support.* Research Report RR-8346. INRIA. 18 pages. https://hal.inria.fr/hal-00853423

[27] A. Huseinović and S. Ribić. 2015. Benchmark comparison of computing the Mandelbrot set in OpenCL. In *23rd Telecommunications Forum Telfor (TELFOR)*. https://doi.org/10.1109/TELFOR.2015.7377632

[28] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. 2012. Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2145816.2145818

[29] IBM. [n.d.]. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/java_jvm.html

[30] Intel. [n.d.]. oneAPI Specification. https://spec.oneapi.com/versions/latest/index.html

[31] V. M. Ionescu. 2017. CPU and GPU gray scale image conversion on mobile platforms. In *9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. https://doi.org/10.1109/ECAI.2017.8166501

[32] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1109/PACT.2013.6618799

[33] Yasir Noman Khalid, Muhammad Aleem, Usman Ahmed, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. 2019. Troodon: A machine-learning based load-balancing application scheduler for CPU-GPU system. *J. Parallel and Distrib. Comput.* 132 (2019), 79 – 94. https://doi.org/10.1016/j.jpdc.2019.05.015

[34] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/2304576.2304623

[35] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/2464996.2465007

[36] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3050748.3050764

[37] C. Luk, S. Hong, and H. Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[38] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. 2015. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE International Conference on Robotics and Automation (ICRA)*. arXiv:1410.2167.

[39] Raul Nozal, Jose Luis Bosque, and Ramon Beivide. 2019. Towards Co-execution on Commodity Heterogeneous Systems: Optimizations for Time-Constrained Scenarios. *International Conference on High Performance Computing & Simulation (HPCS)* (2019). https://doi.org/10.1109/hpcs48598.2019.9188188

[40] William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2015. Fast Automatic Heuristic Construction Using Active Learning. In *Languages and Compilers for Parallel Computing*, James Brodman and Peng Tu (Eds.).

[41] Satoshi Ohshima, Ichitaro Yamazaki, Akihiro Ida, and Rio Yokota. 2018. Optimization of Hierarchical Matrix Computation on GPU. In *Supercomputing Frontiers*. Springer International Publishing.

[42] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2581122.2544163

[43] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis. 2019. Towards Prototyping and Acceleration of Java Programs onto Intel FPGAs. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. https://doi.org/10.1109/FCCM.2019.00051

[44] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. 2020. Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. *The Art, Science, and Engineering of Programming* 5, 2 (2020). https://doi.org/10.22152/programming-journal.org/2021/5/8

[45] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D. Santambrogio. 2020. DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime. arXiv:2012.09646 [cs.DC]

[46] DP Playne, MGB Johnson, and KA Hawick. 2009. Benchmarking GPU Devices with N-Body Simulations. In *Proceedings of the International Conference on Computer Design (CDES)*.

[47] Reuven Y. Rubinstein and Dirk P. Kroese. 2016. *Simulation and the Monte Carlo Method* (3rd ed.). Wiley Publishing.

[48] Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2017. Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017). https://doi.org/10.1145/3126548

[49] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[50] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. 2009. Software Pipelined Execution of Stream Programs on GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2009.20

[51] V. Volkov and J. W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC)*. https://doi.org/10.1109/SC.2008.5214359

[52] Y. Wen, Z. Wang, and M. F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *21st International Conference on High Performance Computing (HiPC)*. https://doi.org/10.1109/HiPC.2014.7116910

[53] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: A Framework for OpenCL Device Abstraction and Management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2688500.2688505