Enabling Pipeline Parallelism in Heterogeneous Managed Runtime Environments via Batch Processing

Enabling Pipeline Parallelism in Heterogeneous Managed Runtime Environments via Batch Processing

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

Athanasios Stratikopoulos
athanasios.stratikopoulos@manchester.ac.uk
The University of Manchester
Manchester, United Kingdom

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

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

Abstract

During the last decade, managed runtime systems have been constantly evolving to become capable of exploiting underlying hardware accelerators, such as GPUs and FPGAs. Regardless of the programming language and their corresponding runtime systems, the majority of the work has been focusing on the compiler front trying to tackle the challenging task of how to enable just-in-time compilation and execution of arbitrary code segments on various accelerators. Besides this challenging task, another important aspect that defines both functional correctness and performance of managed runtime systems is that of automatic memory management. Although automatic memory management improves productivity by abstracting away memory allocation and maintenance, it hinders the capability of using specific memory regions, such as pinned memory, in order to perform data transfer times between the CPU and hardware accelerators.

In this paper, we introduce and evaluate a series of memory optimizations specifically tailored for heterogeneous managed runtime systems. In particular, we propose: (i) transparent and automatic "parallel batch processing" for overlapping data transfers and computation between the host and hardware accelerators in order to enable pipeline parallelism, and (ii) "off-heap pinned memory" in combination with parallel batch processing in order to increase the performance of data transfers without posing any on-heap overheads. These two techniques have been implemented in the context of the state-of-the-art open-source TornadoVM and their combination can lead up to 2.5*x* end-to-end performance speedup against sequential batch processing.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; • **Computing methodologies** → **Parallel programming languages**; • **Computer systems organization** → **Single instruction, multiple data**; • **General and reference** → **Performance**.

Keywords: Data Transfers, GPUs, Heterogeneous Architectures, Memory Management, Optimizations, Virtual Machines

ACM Reference Format:
Florin Blanaru, Athanasios Stratikopoulos, Juan Fumero, and Christos Kotselidis. 2022. Enabling Pipeline Parallelism in Heterogeneous Managed Runtime Environments via Batch Processing. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22), March 1, 2022, Virtual, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3516807.3516821

1 Introduction

Following the development of heterogeneous programming models such as OpenCL [17] and CUDA [7], managed programming languages such as Java, Python, etc. have been making steady progress towards integrating into their execution models and managed runtime environments (MREs) the various hardware accelerators that are commonly found today in a wide spectrum of devices spanning from smartphones to cloud servers [2, 4, 13, 21, 22, 34, 42].

The majority of existing frameworks have been primarily focused on the challenging task of generating suitable code for hardware accelerators from general purpose or domain specific languages with only a few focusing on memory optimizations for improving data transfer times between CPUs and hardware accelerators [14, 15]. In this paper, we analyze the various challenges that managed runtimes face in terms of memory management and heterogeneous execution, and propose a series of optimizations that aim to increase

VEE '22, March 1, 2022, Virtual, Switzerland
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9251-8/22/03…$15.00
https://doi.org/10.1145/3516807.3516821

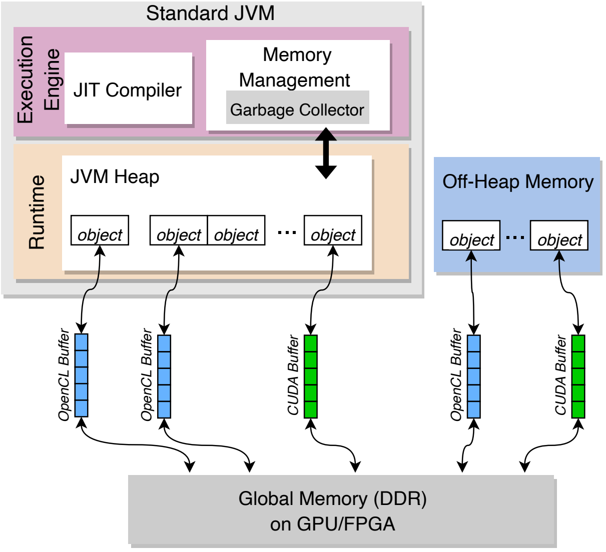


Figure 1. The Overview of a Heterogeneous Managed Runtime System.

performance without posing significant overheads to the underlying runtimes. In detail, this paper makes the following contributions:

- It presents and discusses existing limitations in a state-of-the-art MRE (the Java Runtime Environment) with regards to memory management and data transfers on heterogeneous systems. The analysis is performed based on: (i) the behaviour of asynchronous data transfers, (ii) the utilization of off-heap buffers (data structures that are not managed by the runtime), and (iii) the utilization of pinned memory which is currently not supported by the majority of MREs.
- It introduces the notion of "parallel batch processing" in TornadoVM that enables the overlapping of execution and data transfers between the host device and the accelerators with the aim of increasing performance due to the introduced pipelining.
- It extends the "parallel batch processing" technique with "off-heap pinned" memory that avoids the on-heap memory copies to pre-allocated pinned memory as well as increases performance by avoiding virtual pages from being evicted while execution is performed on the hardware accelerators. The performance analysis indicates that off-heap pinned memory alone can yield up to 50% performance improvement in data transfer times. If combined with parallel batch processing, then results indicate end-to-end execution speedups of up to 2.5x compared to the baseline implementation that does not utilize parallel batch processing and pinned memory.

The remaining of the paper is organized as follows: Section 2 gives the background on managed runtime systems and heterogeneous programming models; and it discusses the posed challenges when combining them. Section 3 presents the implementation of "parallel batch processing" as well as the combination with "off-heap pinned" memory. Section 4 presents the results of the proposed techniques, while Section 5 discusses the related work. Finally, Section 6 concludes the contribution of this paper and pinpoints future work.

2 Background

This section provides the background information on MREs and memory management of heterogeneous programming models. In addition, it analyzes the current challenges for enabling heterogeneous hardware execution from within managed runtimes. In detail, Section 2.1 gives an overview on the internals of a Virtual Machine (VM). Section 2.2 explains the execution model of heterogeneous programming models such as CUDA and OpenCL. Finally, Section 2.3 presents the challenges posed to automatic memory management of MREs by heterogeneous hardware acceleration.

2.1 VM Internal Overview

Although different managed programming languages can execute on various compatible implementations of their corresponding VMs, a number of core VM components can be commonly found across different implementations. In this paper, we focus on the Java programming language, but both the described challenges and the proposed solutions can apply across various VMs that support different managed programming languages. Hence, Figure 1 presents an outline of the core internal components of a standard Java VM (JVM) along with the necessary can be transformed to support heterogeneous hardware acceleration. The standard JVM (top left box) includes an execution engine along with a runtime system. The execution engine encompasses: (i) the JIT Compiler, which generates optimized code based on profiling information retrieved at runtime by the interpreter, thereby enabling profile-guided speculative optimizations which can be mitigated by de-optimization and interpretation if they do not hold true, and (ii) the Memory Management subsystem, which enables automatic memory management - in the form of garbage collection [23] - of the JVM heap. The JVM heap is a memory area allocated in the CPU main memory where all the living objects that belong to the applications reside.

2.2 CUDA and OpenCL execution model

Heterogeneous programming models, such as CUDA [7] and OpenCL [17], employ conceptually similar execution models. Based on the common execution model, a program is composed of two main parts: (i) the host part, which runs on the main CPU and orchestrates the execution of compute kernels as well as the memory management with regards to transferring data from the host to the heterogeneous devices (e.g., a GPU), and backwards, and (ii) the compute kernels, which are functions written in a C-based dialect (either CUDA or OpenCL compute kernels) and they contain the instructions to execute on the heterogeneous device. Discrete GPUs and

FPGAs are usually interconnected with the CPU via PCIe and they are equipped with dedicated memory (i.e., DDR) which is named as "global memory" and is used to store data transferred by the main memory of the CPU. CUDA and OpenCL developers are responsible for allocating the corresponding memory regions (buffers) and keep consistency between the host memory (memory regions allocated on the main CPU) and the device memory (memory regions allocated on the accelerator). In the case of a heterogeneous MRE (shown in Figure 1), the runtime should administer the creation and release of the OpenCL/CUDA buffers, while also assigning them to specific data transfer operations.

The typical workflow for an OpenCL and CUDA program is as follows: developers (i) allocate both the interim buffers and the physical memory space using the corresponding APIs, (ii) copy data from the host memory to the device memory, (iii) launch the execution of the kernels, and (iv) copy the final results from the device memory to the host memory via the allocated buffers. As described, this workflow encompasses the process of transferring data back and forth from the host to the device which is constrained by the PCI-e bandwidth. In some cases, when the memory bandwidth from the global memory to the device processing cores is significantly higher than the PCIe bandwidth, data transfers between the host and the devices can negatively impact performance.

2.3 Challenges

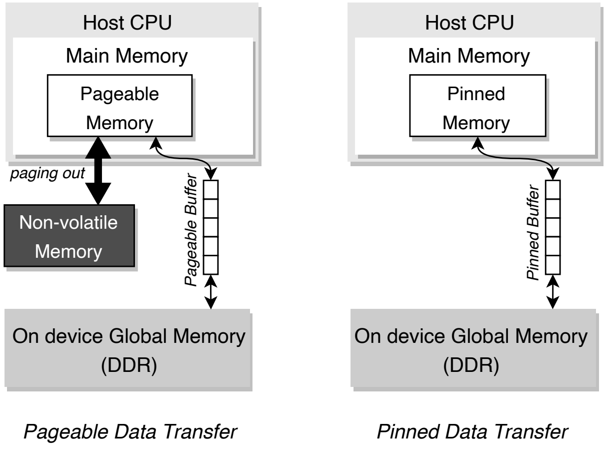
Since managed runtimes have been implemented to execute on CPUs, their adaptation to support heterogeneous hardware acceleration poses several challenges which are explained in the following subsections.

2.3.1 Memory Size of OpenCL/CUDA Buffers. As described earlier in Section 2.2, heterogeneous accelerators have dedicated memory that is used to load data for parallel processing and store the post-processed results. Since dedicated memory on the device is finite and there is no swapping capabilities, applications that want to offload higher volumes of data are restricted. These restrictions are typically reflected to maximum sizes of OpenCL/CUDA buffers that developers can use in order to transfer data between the host and the accelerators. Therefore, developers of such applications are forced to implement data distribution techniques that split the data according to the memory capacity and synchronize the execution of all subsets of data.

A potential solution to this problem could be "batch processing"; a technique that organizes data in groups (called *batches*) of arbitrary sizes. A common application of this technique can be found in distributed Big Data processing frameworks, where data batches are scheduled to execute on distributed nodes of a system [41]. In this paper, we adapt and extend the technique of "batch processing" to heterogeneous processing in the context of MREs, in order to orchestrate data transfer and kernel execution on hardware accelerators with limited memory resources. Section 3 discusses the proposed implementation of "parallel batch processing" which has two advantages: (i) it enables the hardware acceleration of workloads the utilize more memory than physically present on the device, and (ii) it enables the overlapping between data transfers and kernel execution thereby increasing performance via pipeline parallelism.

2.3.2 Blocking Synchronization. Since hardware accelerators are usually attached to the main computing systems via PCI-e, the data used by the compute kernels on the target accelerator must be transferred before the launching of the kernels. CUDA and OpenCL allow data transfers to be performed as asynchronous operations, thereby allowing the CPU threads to continue their execution, while data transfers are being performed. However, in the context of a VM with automatic memory management, all these steps must be in accordance with its memory management system. For instance, if a Java program aims to send an array to a GPU for processing, this will require that the garbage collection will not move the array inside the heap during a GC cycle as this could either crash the execution of the program or lead to uncertain results due to copying of possibly stale data. Therefore, the operations of transferring data from host to devices (and backwards) are typically implemented by heterogeneous MREs (e.g. TornadoVM and Aparapi), as blocking operations. Typically, the runtime system submits data transfer requests to a device driver via a Java Native Interface (JNI) call. The JNI API provides two functions that can be used to lock an object in the JVM heap and prevent it from being moved by the GC: `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` [29]. These functions define a "safe" region as they ensure that the GC will not collect the array object while code is being executed between these two functions. Depending on which GC is being used (e.g., SerialGC, ParallelGC [33], G1GC [10], Shenandoah [11], ZGC [20]), the whole JVM heap might be locked, or a subregion of the heap where the object lives, or only the object itself.

Off-heap buffers. An approach to overcome blocking data transfers is to declare memory off-heap and make it available to the user, as shown in Figure 1. In this case, off-heap memory ensures that garbage collection will not be performed on this memory region and hence enable the usage of non-blocking API calls to the drivers. Thus, the declaration of off-heap memory can allow the CPU threads to continue normal operation without being blocked. The *ByteBuffer* API [31] (introduced in Java 1.4) allows the creation of direct byte buffers, which are allocated off-heap, allowing the user to manipulate off-heap memory directly from Java. One limitation of the *ByteBuffer* API is that the maximum size of a buffer is 2GB. As an alternative to the *ByteBuffer* API, the Foreign Memory Access API [32] (Project Panama)



Pageable Data Transfer *Pinned Data Transfer*

Figure 2. Pageable and Pinned Data Transfers.

has been introduced since Java 14 to address the problem of accessing and managing off-heap memory. The Foreign Memory Access API is implemented as an incubator module (a runtime flag needs to be enabled in order to use it).

2.3.3 Page-locked (Pinned) Memory. The transferring of data that reside in the main memory of the CPU implies that data can be paged-out. Paging out is the process of moving memory pages associated with a particular process from fast-to-access random-access memory (RAM) to slow-to-access non-volatile memory (hard drive). This process uses the non-volatile memory as a storage space, when the RAM is fully occupied. Thus, if the CPU attempts to access any memory area that has been paged out, the contents of this memory area will be first copied from the non-volatile storage to the RAM (paged in). Memory paging can cause significant overheads in the execution of a program. A solution to this problem is Pinned (page-locked) memory which is a term used for pages that are guaranteed that they will not be paged out by the operating system. Furthermore, pinned memory facilitates direct memory access (DMA) [8] compared to pageable memory in which, if a memory area was paged out, the CPU would have to copy data. In general, the performance of data transfers in the case of heterogeneous execution can be influenced by the PCIe interconnection speed and/or the CPU speed in case that pageable memory is being used. Pinned memory can help address the second bottleneck by pinning pages between the host and the hardware accelerator and by using DMA.

Figure 2 illustrates the difference between performing pinned and non-pinned (pageable) memory transfers. In the default scenario (non-pinned) shown in the left part of the figure, data is copied from the pageable host memory to buffers shared with the hardware accelerators. In case of paging-out, pages will be copied between the non-volatile memory of the system and the main memory. When using pinned memory (right part of the figure), data is being shared between the host and the device via pinned buffers. Please note that in reality, even when using pageable memory, generally, the communication between the host and the accelerators take place via interim pinned memory buffers (which requires an extra memory copy operation) [26]. Modern GPUs support pinned memory allocation along with shared virtual address space and unified memory. Therefore, many driver implementations, such as Nvidia [27] and AMD [1], have special guidelines that document the required steps to follow in order to utilize these offerings.

```java
class ProcessImage {
  void apply(int[] image) {
    for (@Parallel int i = 0; i < SIZE; i++)
      filter(image);
  }
  void run(int[] image) {
    TaskSchedule ts = new TaskSchedule("image");
    ts.streamIn(image)
      .task("filter", this::apply, image)
      .streamOut(image)
      .execute();
  }
}
```

Listing 1. Example in TornadoVM to illustrate the use of batch processing.

Pinned memory in the context of a managed runtime, such as the JVM, is currently challenging to use since there is no way to instruct the memory management of the VM to allocate pinned memory. To mitigate this limitation, Section 3.3 presents a novel approach that exploits OpenCL or CUDA APIs to initialize off-heap pinned buffers to be utilized by Java programs.

3 Optimizing Memory Management on Heterogeneous MREs

This section introduces the memory optimizations we performed in the context of heterogeneous MREs. In detail, Section 3.1 introduces the notion of batch processing for addressing the challenge of limited physical memory on hardware accelerators, Section 3.2 explains the addition of pipeline parallelism to batch processing, and finally Section 3.3 extends both sequential and parallel batching with pinned memory and off-heap memory buffers. Although, all optimizations have been performed in the context of TornadoVM [13] and the Java programming language, they are generally applicable to other heterogeneous MREs and programming languages.

3.1 Batch Processing

This subsection provides a brief overview of the execution model of TornadoVM emphasizing on the orchestration of heterogeneous execution. Furthermore, it introduces batch processing as a mechanism to mitigate the inability to process data sizes that exceed the physical memory of hardware accelerators.

3.1.1 Orchestrating Heterogeneous Execution in TornadoVM.

To motivate the use of batch processing as well as explain its design and implementation details, we use the example shown in Listing 1. The example shows a Java class called `ProcessImage` that contains two Java methods. The first method called `apply` (lines 2-5) contains the kernel function to be offloaded and accelerated on the heterogeneous device (e.g., a GPU). Following the TornadoVM API, this method applies a filter computation over an input buffer (image). The second method, called `run` (lines 6-12), creates a `TaskSchedule` Java object, for defining which Java methods should be accelerated (again following the TornadoVM API). The task-schedule sets an input array (`image`) to be copied in and out between the CPU and the device (lines 8 and 10). Furthermore, it defines a task that points to the method `apply` from the same Java class (line 9). Finally, the execution on the hardware accelerator is triggered in line 11 by the `execute` method.

Upon executing the task-schedule, TornadoVM will perform the following actions: (i) it will build a data-flow graph to optimize the data transfers between the host and the target device, and (ii) it will create a list of bytecodes that represents the orchestration of the execution on the heterogeneous hardware. As defined in [13], TornadoVM runs its generated bytecodes in a bytecode interpreter on the main CPU as a way of orchestrating the execution between the code running on the CPU and the accelerators. When running the bytecode interpreter, TornadoVM allocates the input and output buffers, performs the data transfers (data copy between the CPU and the heterogeneous target device), performs the runtime compilation (from Java bytecode to OpenCL and PTX), and dispatches the generated code on the target device.

Listing 2 presents the TornadoVM bytecodes that correspond to the input application shown Listing 1. As shown, bytecode sections are enclosed in regions marked by the `BEGIN` and `END` bytecodes. These bytecodes have an associated identifier (ID), which indicates the default index of the device on which the application will be executed (e.g., on a GPU with index 0). Line 2 executes the `STREAM_IN` bytecode which performs a copy of the whole input array from the host to the default device. Line 3 executes the parallel kernel on the device with index 0 through the `LAUNCH` bytecode. This bytecode must wait for the bytecode at index 2 (bci-2) to finish before it gets executed. It is important to note that, the first time the `LAUNCH` bytecode for a particular task is executed, TornadoVM will compile the Java bytecode onto OpenCL and PTX and execute the task. Since the majority of the TornadoVM bytecodes are defined as non-blocking operations, dependencies amongst them are satisfied by wait operations on `bci`. Finally, line 4 performs a stream out (array copy from the device to the host) operation.

```
1  BEGIN <0>
2  STREAM_IN <0, image>
3  LAUNCH task <0, bci-2, @ProcessImage::apply, image>
4  STREAM_OUT_BLOCKING <0, bci-3, image>
5  END <0>
```

Listing 2. TornadoVM bytecodes that represent the program define in Listing 1.

As the authors explained in [13], TornadoVM assumes that the input data fits into the memory of the target device. Therefore, the buffer allocation and the thread dispatcher are designed to send the whole data and create a block of threads on the target device that maps the whole iteration space, respectively. Any effort to allocate more memory than physically present on the target hardware accelerator will result in an exception and execution will fall back to traditional CPU-only.

3.1.2 Design and Implementation Details.

We introduce batch processing for optimizing kernels running with a 1D parallelism configuration on heterogeneous devices from managed runtime systems. This type of computation corresponds to expressions that can be executed with a map [9] operator, in which each thread computes an input function with a different item from the input data set. In this way, data can be split in smaller chunks and compute exactly the regions that are needed, allowing developers to process big data applications on devices with limited memory.

Transparent batch processing on heterogeneous devices can be achieved by: a) allowing data partitioning within a task-schedule, and b) tuning the number of threads and thread blocks that can be deployed on a hardware accelerator. These two extensions are coupled together and could potentially be applied to other heterogeneous runtime systems, besides TornadoVM in which they are prototyped. The key parts that a runtime should support are the generation of batches and the native allocation of pinned memory. Additionally, the proposed technique can be exploited by any OpenCL compatible device or PTX device.

API call for enabling batch processing. In order to enable batch processing for specific tasks or task-schedules, a new API call within the TornadoVM API has been introduced. More precisely, we introduced a new method called `batch`, which receives one parameter that expresses the number of bytes to be processed in each batch.

Listing 3 shows an example of the new API call that enables batch processing to the code snippet presented in Listing 1. As shown in Line 3, the batch size is set to 256MB. This call instructs the TornadoVM runtime and the TornadoVM compiler to automatically split the input data in chunks and

```java
TaskSchedule ts = new TaskSchedule("image");
  ts.streamIn(image)
    .batch("256MB") // Enable batch processing
    .task("filter", this::apply, image)
    .streamOut(image)
    .execute();
// image size: 268435456 integers (1073MB)
```

Listing 3. Enabling batch processing on heterogenous devices with TornadoVM.

launch the corresponding kernels with an equivalent set of threads to compute the required batch size. The data partitioning and subsequent execution of the batches are applied to the whole task-schedule expression, meaning that all tasks that belong to the same task-schedule are going to be processed using the same batch size. At a high-level, developers only need to specify their preferred batch size. The runtime will then split the assigned work in batches and coordinate the execution. To illustrate our examples, we assume that the input image occupies 1073MB (268435456 of Java integers).

Extending the bytecode interpreter for batch processing. Besides the aforementioned API call, in order to enable batch processing in TornadoVM, extensions were made to its data-flow graph builder, bytecode generator, and compiler. Regarding the data-flow graph builder, it has been augmented so that each node in the graph has the defined batch size attached to it. Then, from the modified data-flow graph, the runtime system generates the TornadoVM bytecodes that orchestrate the execution on the heterogeneous devices. The bytecode generation process has been altered so that bytecodes can identify which batch (data partition) to process. This is achieved by attaching offset information and batch thread numbers to the `STREAM_IN`, `COPY_IN`, `LAUNCH`, `STREAM_OUT` and `STREAM_OUT_BLOCKING` TornadoVM bytecodes.

During bytecode generation, TornadoVM computes how many data partitions to compute in order to satisfy the selected batch size. It also computes the offsets for the input and output data sets for each batch. Additionally, it computes the block of threads that is needed to compute the input batch for 1D map operations. This is because, as mentioned in Section 3.1.2, the 1D map-computation that TornadoVM exposes relates the input size with the number of threads to be deployed on the hardware accelerator. Therefore, each batch from the data partitioning relates to a specific block of threads that will be executed.

Data management and compilation of batches. Listing 4 shows the list of bytecodes generated from the modified system when batch processing is enabled; again using the example shown in Listing 3. In this example, TornadoVM applies an image-filter to 1073MB of input data represented as an array of integers (268435456 integers). Additionally, TornadoVM has been configured to utilize 1024MB of memory on the hardware accelerator. Since the batch size is set to 256MB, the runtime generates four batches of 256MB each and a fifth one of 49MB.

Listing 4 displays the meta-information attached to bytecodes corresponding to the processing of each batch. The first block of bytecodes (batch 1) processes 256MB of data starting from offset 0, and it launches 64 million threads on the target device. Similarly, the second block copies the data region of the next 256MB of data, and it also launches the execution with a block size of 64 million threads. This process is repeated until all blocks have been completed.

Most of the bytecodes are non-blocking, which means that the operations of stream-in, launch, and stream-out can be overlapped enabling pipeline parallelism (explained further in Section 3.2).

Regarding JIT compilation (from Java bytecode to OpenCL or PTX), the runtime system compiles the code of the first batch, when the first `LAUNCH` is computed. Since all blocks, except the last one, are processing the same batch size, the code can be reused between them. If all batches were equally-sized, then the runtime system will reuse the code generated from the first batch for all batches. However, if a batch-size differs, then the runtime system will trigger a recompilation just for that one; in our example that would be batch number five. The reason behind the recompilation is that TornadoVM performs partial evaluation [16] for some expressions resulting in constant propagation in the generated code in order to achieve better performance. Hence, if array boundaries change (as it happens in our example), then the code will be re-compiled and re-optimized. The aforementioned on-the-fly re-compilation technique has been added to TornadoVM as part of the batch processing optimization.

Limitations. Although the proposed approach enables developers to execute expressions with data sets that can exceed the memory capacities of accelerators, it has the following limitations: (i) it only supports 1D-range compute kernels, and (ii) all arrays passed to the tasks within a task-schedule must have the same size since the batch generator and the kernel thread dispatch assumes that all input buffers are equally sized. Both aforementioned limitations are part of the future work since they are mostly engineering challenges that can be addressed as follows: for (i) further data dependency analysis phases can be added in order to support 2D and 3D ranges as well as add support for data broadcasting across batches via an API call, for (ii) developers can create separate task-schedules of different data sizes.

3.2 Parallel Batch Processing

As explained in the previous subsection, the introduction of batch processing in combination with the asynchronous nature of the TornadoVM bytecodes creates an opportunity for

```
bci-0: BEGIN <0>
// batch 1
bci-1: STREAM_IN <0, image, offset:0, size:256MB>
bci-2: LAUNCH task <0, bci-2, @apply, image,
                    thread:64M>
bci-3: STREAM_OUT <0, bci-3, image, offset:0,
                    size:1024>
// batch 2
bci-4: STREAM_IN <0, image, offset:256MB, size:256MB>
bci-5: LAUNCH task <0, bci-5, @apply, image,
                    thread:64M>
bci-6: STREAM_OUT <0, bci-6, image, offset:256MB,
                    size:256MB>
// batch 3
bci-7: STREAM_IN <0, image, offset:512MB, size:256MB>
bci-8: LAUNCH task <0, bci-8, @apply, image,
                    thread:64M>
bci-9: STREAM_OUT <0, bci-9, image, offset:512MB,
                    size:256MB>
// batch 4
bci-10: STREAM_IN <0, image, offset:768MB, size:256MB>
bci-11: LAUNCH task <0, bci-12, @apply, image,
                    thread:64M>
bci-12: STREAM_OUT <0, bci-13, image, offset:768MB,
                    size:256MB>
// batch 5
bci-13: STREAM_IN <0, image, offset:1024MB, size:49MB>
bci-14: LAUNCH task <0, bci-15, @apply, image,
                    thread:12435456>
bci-15: STREAM_OUT_BLOCKING <0, bci-16, image,
                     offset:1024MB, size:49MB>
bci-16: END <0>
```

Listing 4. Generated TornadoVM bytecodes when batch processing is enabled.

exploiting pipeline parallelism by overlapping data transfers and computation. In this subsection we describe the changes made to TornadoVM to enable parallel batch processing.

Batching in a single stream. By default, TornadoVM operates with one command queue (or a single stream) in which commands for read, write, and execute to/from the device are enqueued. After the various commands are enqueued, their execution relies on the underlying target heterogeneous programming models that are supported by TornadoVM (OpenCL and CUDA). In this scenario, when batch processing is enabled, multiple instances of the same kernel are submitted in the command queue; one instance per batch. All data transfers associated with a batch, as well as the memory accesses performed by a kernel instance, operate only on a single memory segment that is assigned to this batch. When performing in-order batching, the same memory segment is assigned to all batches and all batches are executed sequentially, meaning that each command within the command queue is executed in-order.

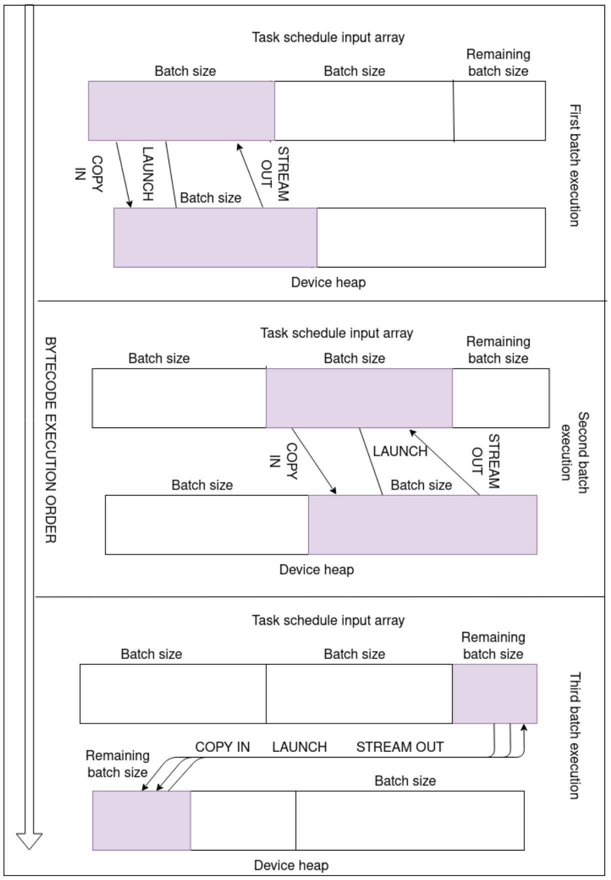


Figure 3. Parallel batch processing in TornadoVM.

Parallel batching with multiple streams. The proposed parallel batch processing technique exploits the concurrent launching of multiple commands via different command queues (or streams). While a kernel is served by a command queue, a second command queue can perform a data transfer between the host and the device memories. Thus, parallel batch processing enables the overlap of kernel execution with data transfers, thereby resulting in lower end-to-end execution time and higher hardware utilization.

Figure 3 illustrates how parallel batch processing is performed. In this example, we consider the execution of three batches, while the device memory space (device heap) can only fit two batches. The first batch occupies the first segment of the device heap, while the second batch occupies the second. Furthermore, each batch of the input array uses a separate command queue. Due to the fact that the device heap in this example fits only two batches, the remaining batch will be enqueued by the first command queue; and therefore, it will use the first memory segment.

3.3 Batching with Pinned Memory

As described in subsection 2.3.2, performing asynchronous data transfers between a host and a hardware accelerator is an unsafe operation in the context of MREs. The garbage collector can move the data to a different location during the transfer, thus causing invalid data to be copied and result in corrupted memory. Since, by design, there is no straightforward way to pin an object in the JVM heap without preventing garbage collection, an alternative is to use off-heap buffers as mentioned in subsection 2.3.2.

To mitigate this limitation, in this subsection we introduce another optimization that complements parallel batch processing for allowing the batches to be allocated outside the JVM heap and utilize page-locked (pinned) memory. To enable this optimization: (i) the batch memory allocator has been integrated with a new Java native API (Project Panama [30]), and (ii) new compiler phases that translate CPU native memory accesses into GPU memory accesses have been added to the TornadoVM JIT compiler.

3.3.1 Off-Heap Buffers. Off-heap memory buffers are not managed directly by the JVM and therefore the garbage collector will not be moving objects residing in them. For implementation purposes we utilize the newly introduced off-heap memory allocation capabilities of OpenJDK in the context of Project Panama [30]. Project Panama is an OpenJDK project designed to provide a new API for interconnecting Java with native code and includes multiple components such as: (i) support for native function calls, (ii) native library management APIs, and (iii) header file extraction tools. The proposed technique has been prototyped as a method in the TornadoVM API which allows the allocation of both non-pinned or pinned off-heap buffers (or `MemorySegments` following the Project Panama naming convention).

```
1  void copy(MemorySegment in, MemorySegment out,
2            int size) {
3    for (int i = 0; i < size; i++) {
4      value = MemoryAccess.getIntAtIndex(in, i);
5      MemoryAccess.setIntAtIndex(out, i, value);
6    }
7  }
8
9  void run(TornadoRuntime rt, Device device, int size)
10 {
11   MemorySegment msA, msB;
12   msA = rt.getOffHeapBuffer(device, size);
13   msB = rt.getOffHeapBuffer(device, size);
14   copy(msA, msB, size);
15 }
```

Listing 5. Off-heap memory allocation in TornadoVM.

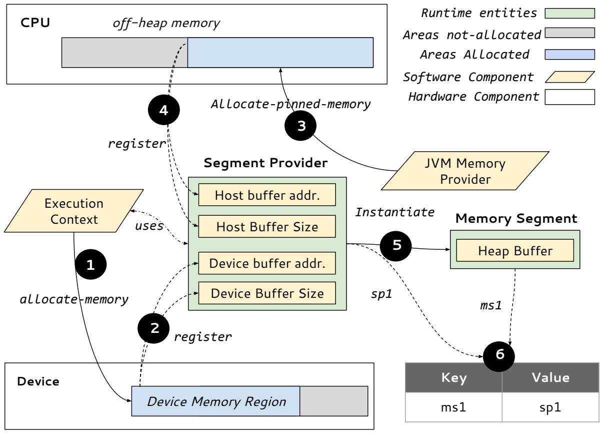


Figure 4. Workflow of our runtime to enable pinned memory for off-heap buffers using `MemoryRegions` from the Panama Project.

As part of this new approach, Project Panama provides an API to create `MemorySegments` that are allocated off-heap. Consequently, the TornadoVM API has been enhanced to allow allocation of off-heap buffers that are backed by a buffer on the target device. Listing 5 presents an example of the enhanced TornadoVM API which is capable of allocating off-heap memory regions. This example copies the contents of a memory segment (`msA`) that is allocated in off-heap memory to a second memory region (`msB`) that is also allocated in off-heap memory. Lines 1-7 present the method that copies the contents of the input array to the output array. Both arrays are declared to be a `MemorySegment` object (named `in` and `out`). Consequently, lines 12 and 13 show how the TornadoVM API has been extended to allocate off-heap memory space pointed by `MemorySegment` of a given `size`. The `getOffHeapBuffer` API method accepts as input the selected device for which memory will be allocated along with the required size for the allocation. In turn, the allocated buffers are forwarded in the `copy` method (line 14). A task can read and write the allocated arrays via the `MemoryAccess::getIntAtIndex` and `MemoryAccess::setIntAtIndex` functions in lines 4 and 5, respectively.

3.3.2 Non-Pinned or Pinned Memory Allocation. The proposed approach for off-heap memory allocation is designed to support both pinned and non-pinned memory since both implementations have advantages and disadvantages. For example, although pinned memory can increase the performance of data transfers the fact that it is page-locked can potentially result in overall system performance degradation if multiple processes are running concurrently [40].

Pinned memory must be allocated through the driver API (either with OpenCL or CUDA driver APIs) and, to the best of our knowledge, current JVMs do not provide a way to allocate objects in different memory pools or even to define such memory pools. Hence, we implemented a method that selected between pinned and non-pinned memory allocation based on a runtime flag that can be also set by developers.

Figure 4 shows the workflow of our runtime system for allocating pinned memory segments that are accessed from both the host and the target hardware accelerator. This workflow is valid for both OpenCL and CUDA backends. The main entity is the segment provider, which is a data type for storing and maintaining pointers between the host and device.

The process for requesting pinned memory segments with pinned memory is as follows: First, ① the device execution context allocates a buffer on the device that is associated with the pinned memory. Then, ② the runtime system registers the device buffer address and the buffer size in the segment provider. Note that the execution context maintains an instance object of a segment provider per buffer being used, and it can be accessed by the runtime system for performing the data management. Besides, the size of the device buffer should be the same as the host buffer. The address of the allocated device buffer is used later by the runtime to perform the actual mapping for data transfers from the host. Then, in step ③, the JVM memory provider allocates off-heap buffers using pinned memory from the main memory of the CPU system (host side). This address, along with the size of the host buffer are also registered in the segment provider (step ④). In step ⑤, the Panama API is utilized to obtain a raw address of the host buffer and instantiate a memory segment object. Finally, in step ⑥, the allocated segment that points to the allocated off-heap pinned memory space is registered along with the runtime information stored in the segment provider. The allocated segment is returned to the application level. Note that the runtime system creates an instance of a JVM memory provider entity per utilized buffer, and the corresponding instance of the memory segment. As an example, step ⑥ takes two object instances, one for the segment provider (`sp1`), and the other for a memory segment instantiation (`ms1`). These objects are stored in a table (implemented as a hash-map), as they are represented at the bottom-right part of Figure 4.

3.3.3 Compiler Support for Off-Heap Memory Regions.

Besides the allocation of off-heap buffers, the TornadoVM compiler has been also extended with custom compiler intrinsincs in the way of compiler snippets [38]. These intrinsincs are responsible for replacing the memory access operations (i.e., read, write) that are performed on a `MemorySegment` object with read/write instructions according to the target backend (OpenCL or PTX). This replacement is performed transparently by the compiler enabling the accessibility of off-heap allocated memory from a kernel running on a heterogeneous device.

4 Experimental Evaluation

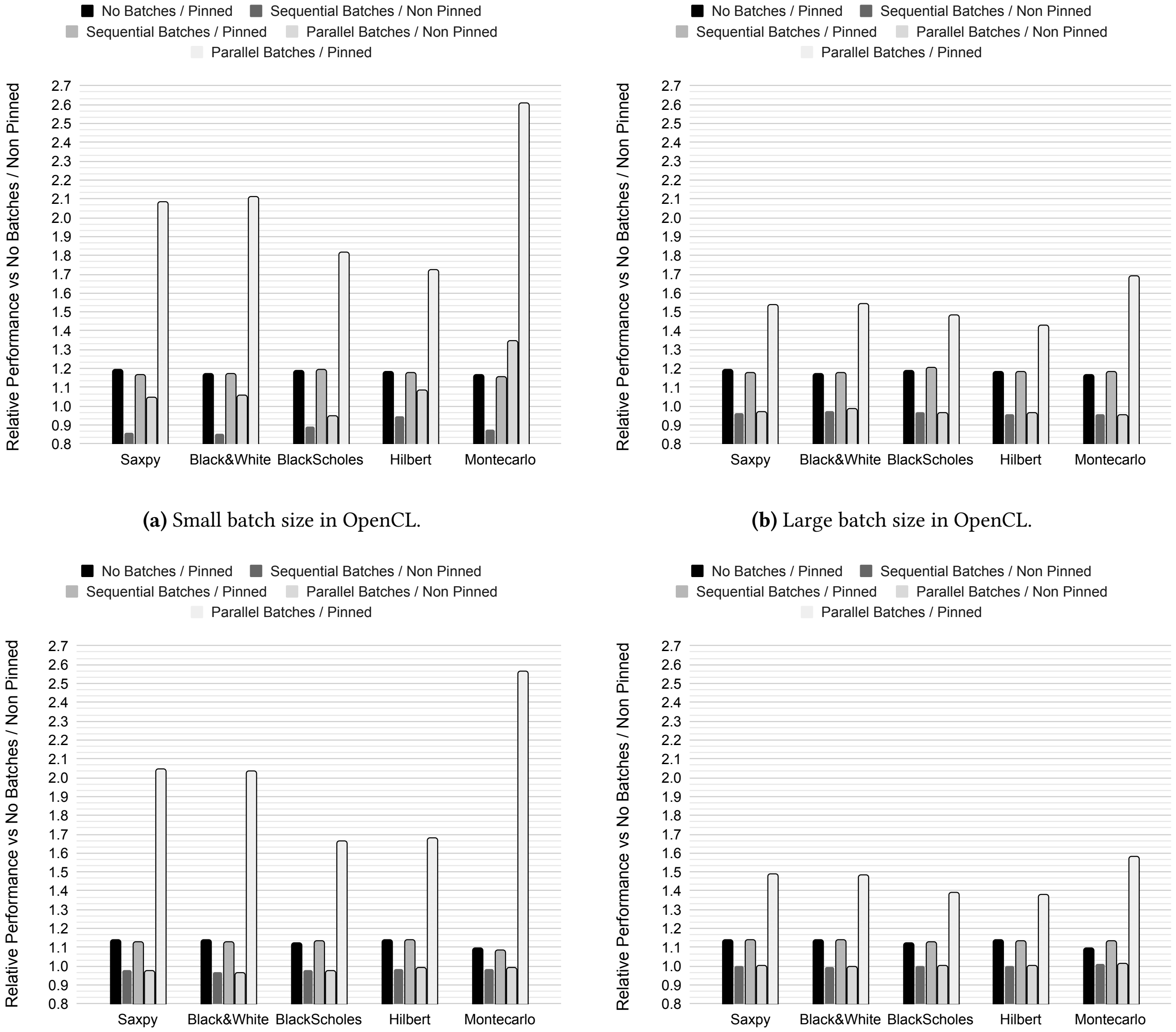
To evaluate the performance of the proposed sequential and parallel batch processing as well as the impact of utilizing pinned memory, we conducted experiments that assess performance on an Nvidia GPU for all TornadoVM backends (OpenCL, PTX). Besides completeness, one of the reasons we evaluate both OpenCL and PTX backends is to discover if there are any differences between them, in terms of performance. Table 1 presents the specifications of our testbed along with the configuration of the JVM that we used to run all the benchmarks (Section 4.1). For each experiment, we performed a warm-up process that includes 10 iterative executions, which are sufficient to ensure that the code is JIT compiled by the TornadoVM compiler for all implementations. The reported results are the average execution times of the next 100 executions. Additionally, to ensure fair comparisons, we performed our experiments for 1 GB of data, as this is the maximum amount of data that the baseline implementation supports. Throughout the evaluation of our experiments, we used two classes of batch sizes (*small* with a size of 32 MB, and *large* with a size of 512 MB) for all benchmarks. Thus, the transaction of 1 GB of data results in moving 32 small batches or 2 large batches, accordingly. Section 4.2 presents the comparative analysis of all implementations. Nonetheless, the proposed techniques of sequential and parallel batch processing along with pinned memory have been evaluated for large data sizes that exceed the memory capacity of the hardware accelerator (e.g., GPU RAM), as discussed in Section 4.3.

Table 1. Hardware/Software testbed characteristics.

CPU	Intel(R) Core(TM) i7-9750H @ 4.5 GHz
Main Memory	32 GB
GPU	Nvidia GeForce GTX 1650
GPU RAM	4 GB
PCIe	Gen3 (16 lanes)
Hard Drive	Toshiba NVMe SSD (1 TB)
JVM	OpenJDK 16 GraalVM CE 21.1.0
JVM Heap Size	16 GB
OS	Ubuntu 20.04
OpenCL Driver	OpenCL 3.0 CUDA 11.4.94
CUDA Driver	470.57.02

4.1 Benchmark Applications

In our experiments, we used five applications that belong to a wide range of domain areas, ranging from mathematical and financial applications to physics and linear algebra. *Saxpy* executes a linear algebra operation that scales a vector by a scalar value and adds the interim result to a second vector, while *Hilbert* is used to perform matrix computations. *Black&White* is an image processing algorithm that transforms the illustration of a colored image in black and white color variants. *Blacksholes* is a mathematical model used in the financial market to calculate theoretical estimates of prices in order to eliminate risks. *Montecarlo* is a simulation method used to solve mathematical and physical problems based on iterative random sampling. This set of benchmarks



(a) Small batch size in OpenCL.

(b) Large batch size in OpenCL.

(c) Small batch size in PTX.

(d) Large batch size in PTX.

Figure 5. Relative performance of sequential and parallel batch processing combined with pinned and non pinned memory, against the system that implements no batch processing and non pinned memory. The higher, the better. The figures are classified based on the TornadoVM backend (OpenCL at the top, PTX at the bottom).

allows us to observe the performance of the proposed techniques in different scenarios as well as highlight the strength and weaknesses of the proposed approaches. For instance, *Saxpy* and *Hilbert* require a large amount of data to be copied while the computation is not too significant. On the other hand, *Black&White* and *BlackScholes* offer higher degree of computation that can be performed in parallel. Finally, *Montecarlo* accepts as input an array that contains seed numbers used for generating the simulation data which are processed in parallel.

4.2 Performance Analysis

Figure 5 presents the relative performance for each configuration of batch processing (*no*/*sequential*/*parallel*) and the utilization of *pinned* or *non pinned* memory; against the baseline which uses *no* batch processing and *non pinned* memory. We conducted the same experiment for two TornadoVM backends (OpenCL at the top, PTX at the bottom) and for small (Figures 5a & 5c) or large (Figures 5b & 5d) batch sizes, respectively. The relative comparison against the different configurations is obtained by timing the end-to-end execution time of each configuration on the same environment.

Pinned Memory. The first bar in Figures 5a & 5b (depicted in black) shows that the utilization of pinned memory can increase the performance of the baseline system (*No Batches/Non Pinned*) in OpenCL from 16% (Montecarlo) to 20% (Saxpy). Similarly, for the PTX implementation pinned memory can yield performance improvements ranging from 10% (Montecarlo) to 14% (Saxpy, Black&White, Hilbert), as shown in Figures 5c & 5d. Note that the performance of the *No Batches/Pinned* implementation is the same for both small and large batch sizes, since it does not split data into batches. However, we report it for both cases to reflect on the improvement of the batch processing technique when pinned memory is also applied.

Sequential Batches and Non Pinned Memory. Another interesting remark is that the *Sequential Batches/Non Pinned* implementation achieves the lowest performance across both backends (OpenCL, PTX) in Figure 5. This indicates that the time consumed in splitting the data as well as the fact that batches are executed sequentially result in performance decrease. In particular, the performance loss in OpenCL against the baseline is up to 15% (Figure 5a - Black&White) and 4% (Figure 5b - Saxpy, Hilbert, Montecarlo) for small and large batch sizes, respectively. However, when pinned memory is applied, the *Sequential Batches/Pinned* configuration showcases up to 1.2x performance speedup (Figures 5a & 5b - BlackScholes) against the baseline. Similarly OpenCL, the *Sequential Batches/Non Pinned* implementation in PTX results to up to 3% (Figure 5c - Black&White) performance loss for small batch sizes, while for the large batch size (Figure 5d) the performance is identical to the baseline performance. Additionally, when pinned memory is combined with sequential batch processing the system achieves performance speedup against the baseline which ranges from 1.08x (Montecarlo) to 1.13x (Hilbert) in small batch sizes (Figure 5c), and up to 1.14x for large batch sizes (Figure 5d - Black&White).

Parallel Batches and Non Pinned Memory. The *Parallel Batches/Non Pinned* evaluation shows that the performance in OpenCL against the baseline ranges from 0.95x (BlackScholes) to 1.35x (Montecarlo) for small batch sizes (Figure 5a). For large batch sizes, the overall performance against the baseline drops up to 5% (Montecarlo). Similarly to OpenCL, the performance of the *Parallel Batches/Non Pinned* implementation in PTX against baseline drops up to 3.5% (Figure 5c - Black&White) for small batch sizes, while performing identically for large batch sizes (Figure 5d). The rationale for the low performance of the *Parallel Batches/Non Pinned* implementation is that non pinned data movements are slower than pinned (will be discussed in Section 4.2.1), thereby resulting in less overlapping and limited parallel execution of batches on the GPU. In our experiment, we observed that by enabling pinned memory, the overlapping for small batch sizes increases up to 72%.

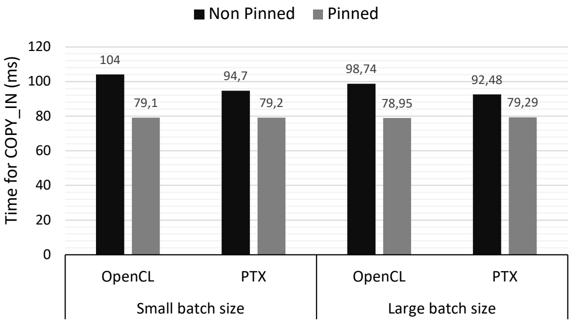


Figure 6. Time for transferring 1GB of data from host to device (COPY_IN). Time is reported in milliseconds.

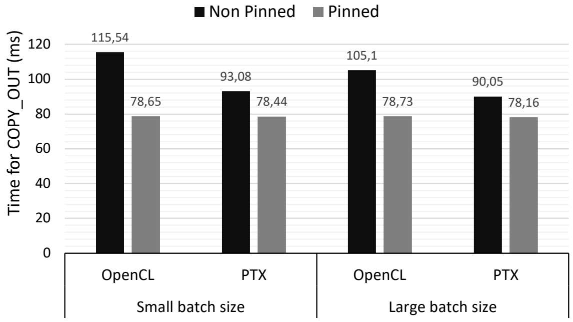


Figure 7. Time for transferring 1GB of data from device to host (COPY_OUT). Time is reported in milliseconds.

Finally, the combination of pinned memory and parallel batch processing outperforms the baseline implementation in OpenCL by 2.6x (Figure 5a - Montecarlo) and 1.68x (Figure 5b - Montecarlo) for small and large batch sizes, respectively. Similarly to OpenCL, this configuration in PTX achieves speedups, against the baseline, of up to 2.56x (Figure 5c - Montecarlo) and 1.58x (Figure 5d - Montecarlo) for small and large batch sizes, accordingly.

4.2.1 Analysis of Pinned Data Transfers. As shown in the previous subsection, pinned memory enables the batch processing techniques to perform better than non pinned memory. The reason is that pinned memory can enable data transfers through DMA. To better understand the impact of pinned memory, we performed a study that compares the performance of pinned data transfers against non pinned. Our analysis focuses on the performance of the data transactions which occurred when we run the Black&White application for 1GB of data, as presented in the previous section.

Performance of copies from host to device. Figure 6 presents the total time for performing the copy of 1GB of data from the host memory to the device memory (`COPY_IN`), when using small and large batch sizes. For non pinned transactions (black bars) the PTX backend performs up to

10% and 7% faster than OpenCL for small and large batch sizes, respectively. For the small batch size, pinned memory results in increasing the performance of data copying from host to device by up to 31% in OpenCL (from 104 ms to 79.1 ms); and up to 19.5% in PTX (from 94.7 ms to 79.2 ms). Furthermore, pinned memory combined with a large batch size yields up to 25% and 16.6% performance improvement for OpenCL and PTX, respectively. A last remark is that pinned data copies (grey bars) take approximately the same time (around 79 ms) for both OpenCL and PTX implementations.

Performance of copies from device to host. Figure 7 presents the time for transferring data from the device memory back to the host (COPY_OUT). The performance trend in this case is similar to the reverse data transfer (Figure 6). For non pinned data transfers, the OpenCL implementation that uses a small batch size is 10 ms slower compared to using a large batch size. On the contrary, the PTX implementations for small and large batch sizes present a deviation of 3 ms, ranging from 93.08 ms to 90.05 ms. For the small batch size, the data transfers performed by the CUDA driver are 24% faster than OpenCL, while for the large batch size they improve by 16.7%. This performance behavior is attributed to the actual implementation of the CUDA and OpenCL drivers. Nonetheless, for pinned data transfers both drivers exhibit similar performance (around 78.5 ms) for both small and large batch sizes.

4.3 Going beyond the GPU Memory Capacity

To study the performance of the proposed batch processing techniques (i.e., sequential and parallel) for large data sizes, we performed an experiment that increases the data size to 8 GB (twice the GPU memory capacity) and compares it to the performance of 1 GB of data (shown previously in Figure 5). In this experiment, we examined the performance for the small batch size, as this size results in higher overlapping between the data transfers and the computation performed on the GPU. Due to the fact that the OpenCL programming model does not allow the allocation of pinned memory for sizes that go beyond to the 25% of the actual memory capacity in the device [17], we were not able to conduct this experiment for OpenCL. Therefore, the rest of this section focuses on PTX, and in particular the involvement of the CUDA driver for performing the pinned and non pinned data transfers.

Figure 8 presents the relative speedup of the sequential and parallel batch processing techniques that utilize pinned memory against their corresponding implementation with non pinned memory (the higher, the better). We observe that the performance achieved with the sequential batch processing technique with pinned memory enabled is at the same level (up to 1.18x for Black&White), while increasing the data size from 1 GB to 8 GB. The parallel batch processing technique also demonstrates the efficiency of the proposed technique for 8 GB of data, as it results in performance improvements ranging from 1.52x (Hilbert) to 2.34x (Montecarlo). The results shown in Figure 8 showcase that the proposed batch processing techniques enable applications to utilize volumes of data that exceed the physical memory capacity of the hardware device; a capability can be extremely beneficial, especially for Java-based Big Data frameworks such as Apache Spark [25, 28] and Flink[3, 5].

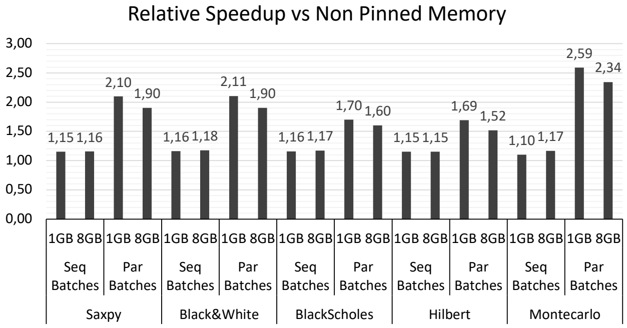


Figure 8. Relative performance of pinned memory against non pinned implementations for different data sizes in PTX.

5 Related Work

This section discusses the related work regarding data transfer optimizations in the context of heterogeneous managed runtime systems. Batch processing and pinned memory are techniques that have been extensively studied during the past years [18, 24, 35, 36, 39] for programming frameworks based on unmanaged programming environments, such as C/C++, CUDA, and OpenCL. However, limited research has been conducted in applying these techniques to managed programming languages, such as Java, .NET, Python, etc. This is, in part, due to the complexity of managed runtime systems regarding memory management and garbage collection. One way to circumvent this challenge is to allow the use of off-heap memory. However, this memory area needs to be handled efficiently and explicitly by developers. To the best of our knowledge, this is the first work that proposes a technique for combining batch processing for overlapping computation and communication along with the use of pinned memory.

There exist solutions for GPU and FPGA acceleration from managed runtime programming languages. Aparapi [2] is a parallel programming framework for GPU compute within Java using on-heap arrays. Similarly, IBM J9 [21] enables expressions written with the Java Stream 8 API to execute on GPUs via CUDA PTX. IBM J9 uses on-heap arrays and precompiled kernels for some of the functions and it does not support pinned memory. Similarly, other Java approaches such as Rootbeer [34] and JaBEE [42] follow the same trend. Finally, Habanero-Java (HB) [19] is a parallel programming language based on Java that enables GPU compute.

Concurnas [6] is a new programming language built on top of the JVM designed to make use of modern hardware. Concurnas uses off-heap and non-pinned arrays that are directly exposed to developers. However, the whole memory management between the accelerator and JVM is handled by developers (memory copies, synchronization and clean-up).

The techniques described in this paper are complementary and applicable to the aforementioned systems.

Batch Processing and Pinned Memory from MRE. The most related work regarding batch processing and pinned memory is Marawacc [12, 15] a runtime system and a JIT compiler for offloading Java bytecodes to OpenCL-compatible hardware. Marawacc exposes an API to developers that allows the transparent use of off-heap and OpenCL pinned memory buffers. Marawacc internally enables batch processing only when the input application does not fit on the device memory. Similarly, R-GPU uses transparent pinned memory and batch processing for large arrays in R [14]. Our proposed approach is more general in the sense that developers can control batch sizes, allowing applications to share the same device. Additionally, Marawacc does not support multiple command queues to overlap data with communications. In contrast to Marawacc and R-GPU, the proposed work in this paper enables batch processing even if the data size does not exceed the physical memory capacity of the GPU; and supports parallel batch processing.

Dandelion [37] is a parallel programming framework for the .NET execution environment that uses different communication channels to perform data communications between CPUs and GPUs via different execution engines. While authors do not explicitly define Dandelion as a system that can overlap communication and computation, the fact that it uses different channels may allow the runtime system to perform such optimization, in a similar manner as proposed in this paper. Regardless, the work proposed in this paper allows the combination of parallel batching with pinned memory.

6 Conclusions

In this paper, we discuss the limitations and challenges of memory management in managed runtime environments (MREs) with regards to heterogeneous hardware acceleration. To address a number of these challenges, this paper introduces two main approaches that optimize the performance of data transfers in heterogeneous MREs. The first approach, called batch processing, enables the utilization of data sizes that exceed the physical memory present on hardware accelerators. The second approach improves batch processing by adding pipeline parallelism which allows the runtime system to overlap communication with computation to increase performance. Finally, it demonstrates a strategy to combine parallel batch processing with pinned memory that enable fast DMA data transfers ensuring that the allocated memory pages will not be paged out. The aforementioned optimizations have been implemented in the context of the state-of-the-art TornadoVM and have been evaluated across a diverse set of benchmarks. The performance evaluation showcases that when all optimizations are combined, end-to-end speedups of up to 2.5x are achieved compared to the baseline implementation.

Acknowledgments

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

References

[1] AMD. Accessed in November 2021. AMD APP SDK OpenCL Optimization Guide. http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf

[2] AMD. Accessed in November 2021. Aparapi. https://aparapi.github.io/

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[4] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language. *SIGPLAN Not.* (feb 2011), 47–56. https://doi.org/10.1145/2038037.1941562

[5] Cen Chen, Kenli Li, Aijia Ouyang, and Keqin Li. 2018. FlinkCL: An OpenCL-Based In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Trans. Comput.* 67 (2018), 1765–1779.

[6] Concurnas. Accessed in December 2021. The Concurnas Programming Language. https://concurnas.com

[7] NVIDIA Corporation. Accessed in 2021. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/

[8] Jakub Jelinek David S. Miller, Richard Henderson. Accessed in November 2021. Dynamic DMA mapping Guide. https://www.kernel.org/doc/html/latest/core-api/dma-api-howto.html

[9] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*. https://doi.org/10.1145/1327452.1327492

[10] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. 37–48. https://doi.org/10.1145/1029873.1029879

[11] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. https://doi.org/10.1145/2972206.2972210

[12] Juan Fumero. 2017. *Accelerating Interpreted Programming Languages on GPUs with Just-In-Time and Runtime Optimisations.* Ph. D. Dissertation. The University of Edinburgh, UK.

[13] 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 2019)*. Association for Computing Machinery. https://doi.org/10.1145/3313808.3313819

[14] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS*

International Conference on Virtual Execution Environments (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/3050748.3050761

[15] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (Melbourne, FL, USA) *(PPPJ '15)*. Association for Computing Machinery, New York, NY, USA, 16–26. https://doi.org/10.1145/2807426.2807428

[16] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 2 (1999), 45–50.

[17] Khronos OpneCL Working Group. Accessed in 2021. The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html

[18] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. *SIGCOMM Comput. Commun. Rev.* (aug 2010), 195–206. https://doi.org/10.1145/1851275.1851207

[19] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2013. Accelerating Habanero-Java Programs with OpenCL Generation. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Stuttgart, Germany) *(PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 124–134. https://doi.org/10.1145/2500828.2500840

[20] Per Liden Iris Clark. Accessed in November 2021. ZGC Main Page. https://wiki.openjdk.java.net/display/zgc/Main

[21] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 419–431. https://doi.org/10.1109/PACT.2015.46

[22] Dejice Jacob, Phil Trinder, and Jeremy Singer. 2019. Python Programmers Have GPUs Too: Automatic Python Loop Parallelization with Staged Dependence Analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/3359619.3359743

[23] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

[24] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1357–1370. https://doi.org/10.1145/3373376.3378529

[25] D. Manzi and David Tompkins. 2016. Exploring GPU Acceleration of Apache Spark. *IEEE International Conference on Cloud Engineering (IC2E)* (2016), 222–223.

[26] Nvidia. Accessed in November 2021. Memory Management CUDA Function Calls. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1gab84100ae1fa1b12eaca660207ef585b

[27] Nvidia. Accessed in November 2021. NVIDIA OpenCL Best Practices Guide. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[28] Yasuhiro Ohno, Shin Morishima, and Hiroki Matsutani. 2016. Accelerating Spark RDD Operations with Local and Remote GPU Devices. *IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)* (2016), 791–799.

[29] Oracle. Accessed in November 2021. Java Native Interface Specification. https://docs.oracle.com/en/java/javase/16/docs/specs/jni/index.html

[30] Oracle. Accessed in November 2021. Project Panama: Interconnecting JVM and native code. https://openjdk.java.net/projects/panama/

[31] Oracle. Accessed in November 2021. The ByteBuffer API Java Documentation. https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/ByteBuffer.html

[32] Oracle. Accessed in November 2021. The Foreign Memory Access API latest incubator JEP. https://openjdk.java.net/jeps/393

[33] Oracle. Accessed in November 2021. The Parallel Collector. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[34] Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. 2012. Rootbeer: Seamlessly Using GPUs from Java. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*. https://doi.org/10.1109/HPCC.2012.57

[35] Carlos Reaño and Federico Silla. 2015. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *IEEE International Conference on Cluster Computing*. 488–489. https://doi.org/10.1109/CLUSTER.2015.76

[36] Chris Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. Symposium on Operating Systems Principles (SOSP). https://www.microsoft.com/en-us/research/publication/ptask-operating-system-abstractions-to-manage-gpus-as-compute-devices/

[37] Chris Rossbach, Yuan Yu, Jon Currey, and Jean-Philippe Martin. 2013. *Dandelion: a Compiler and Runtime for Heterogeneous Systems*. Technical Report MSR-TR-2013-44. https://www.microsoft.com/en-us/research/publication/dandelion-a-compiler-and-runtime-for-heterogeneous-systems/

[38] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, Article 20 (jun 2015), 25 pages. https://doi.org/10.1145/2764907

[39] Mathias Steinbach and Reinhard Hemmerling. 2012. Accelerating batch processing of spatial raster analysis using GPU. *Computers & Geosciences* 45 (2012), 212–220. https://doi.org/10.1016/j.cageo.2011.11.012

[40] N. Wilt. 2013. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education. http://books.google.com/books?id=ynydqKP225EC

[41] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

[42] Wojciech Zaremba, Yuan Lin, and Vinod Grover. 2012. JaBEE: Framework for Object-Oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (London, United Kingdom) *(GPGPU-5)*. Association for Computing Machinery, New York, NY, USA, 74–83. https://doi.org/10.1145/2159430.2159439