Data Transfer Optimizations for Heterogeneous Managed Runtime Systems

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF MASTER OF PHILOSOPHY IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2021

By
Florin-Gabriel Blanaru
Department of Computer Science
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>5</td>
</tr>
<tr>
<td>Declaration</td>
<td>6</td>
</tr>
<tr>
<td>Copyright</td>
<td>7</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>8</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2 Background</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Heterogeneous computing: a short history, description, benefits, and current usages</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 CUDA</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1.1 CUDA hardware architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1.2 CUDA programming model</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 OpenCL</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Managed runtimes</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Challenges in managed runtimes regarding heterogeneous computing</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Asynchronous data transfers</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Off-heap buffers</td>
<td>19</td>
</tr>
<tr>
<td>2.3.3 Page-locked memory</td>
<td>20</td>
</tr>
<tr>
<td>3 Accelerating data transfers in the context of TornadoVM</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Software Architecture of TornadoVM</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Batch processing in TornadoVM</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1 Proposal: Improving performance of Memory Management for GPUs and FPGAs.</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Off-heap buffers and pinned memory</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Experimental Performance Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>3.4.1 Data transfer times using pinned vs non-pinned memory without batch processing</td>
<td>37</td>
</tr>
<tr>
<td>3.4.1.1 Laptop performance evaluation</td>
<td>37</td>
</tr>
<tr>
<td>3.4.1.2 Server performance evaluation</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 Batch processing times using pinned vs non-pinned memory</td>
<td>47</td>
</tr>
<tr>
<td>3.4.2.1 Laptop batch processing</td>
<td>48</td>
</tr>
<tr>
<td>3.4.2.2 Server batch processing performance evaluation</td>
<td>52</td>
</tr>
<tr>
<td>4 Using CUDA Unified Memory for object allocation in MaxineVM</td>
<td>55</td>
</tr>
<tr>
<td>4.1 The CUDA Unified Memory</td>
<td>55</td>
</tr>
<tr>
<td>4.2 MaxineVM - an overview of the architecture</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Allocating the JVM heap in the Unified Memory address space</td>
<td>58</td>
</tr>
<tr>
<td>4.4 Description of MaxCudaLib</td>
<td>59</td>
</tr>
<tr>
<td>4.5 Experimental performance evaluation</td>
<td>60</td>
</tr>
<tr>
<td>5 Related work</td>
<td>66</td>
</tr>
<tr>
<td>6 Conclusion and future work</td>
<td>68</td>
</tr>
<tr>
<td>Bibliography</td>
<td>69</td>
</tr>
</tbody>
</table>
## List of Tables

| Table 1 | List of TornadoVM bytecodes. The table is taken from [40]. | 25 |
| Table 2 | System configuration used to run the benchmark. | 37 |
| Table 3 | Configuration of sequential/parallel batches and pinned/non-pinned memory used. | 48 |
| Table 4 | The differences between TornadoVM, MaxineVM and related work. | 66 |
List of Figures

1. High level architectural comparison between the CPU on the left side and the GPU
on the right side. The figure is taken from [15]. ........................................... 12
2. The CUDA GPU architecture from a hardware perspective. .............................. 13
3. The relationship between grids, blocks and threads. Grids contain multiple blocks
and each block contains multiple threads. The figure is taken from [15]. ............ 15
4. A single Global Work Group contains four local work-groups with ids from 0 to 3.
Each local work-group incorporates 18 work-items. ........................................ 16
5. Memory hierarchy in OpenCL for GPUs. The figure is taken from [18] ......... 17
6. Non-pinned data transfer (left) compared to pinned data transfer (right). The figure is
taken from [35]. .................................................................................................. 20
7. TornadoVM architecture overview and workflow .............................................. 24
8. Current implementation of batch processing in TornadoVM. The Task Schedule
input array is split into three batches ..................................................................... 27
9. Parallel execution of batch processing in TornadoVM. The Task Schedule input
array is split into three batches. For each batch, COPY_IN, LAUNCH and STREAM_OUT
bytecodes are executed .......................................................................................... 29
10. Intermediate Representation (IR) graph after the Graph Builder phase obtained from
compiling the transformToBlackAndWhite method in Listing 4 ......................... 34
11. Data transfers with pinned and non-pinned memory in PTX over PCIe3 on the Laptop
configuration .......................................................................................................... 38
12. Data transfers with pinned and non-pinned memory in OpenCL over PCIe3 on the
Laptop configuration .............................................................................................. 39
13. Copy-in and copy-out speed up obtained by using pinned memory on the PTX
backend over PCIe3 on the Laptop configuration .................................................. 40
14. Copy-in and copy-out speed up obtained by using pinned memory on the OpenCL
backend over PCIe3 on the Laptop configuration .................................................. 40
15. Total task-schedule time comparison between pinned and non-pinned memory on the
PTX backend over PCIe3 on the Laptop configuration ......................................... 41
16. Total task-schedule time comparison between pinned and non-pinned memory on the
OpenCL backend over PCIe3 on the Laptop configuration .................................... 42
17. Data transfers with pinned and non-pinned memory in PTX over PCIe3 on the Server
configuration ......................................................................................................... 43
18. Data transfers with pinned and non-pinned memory in OpenCL over PCIe3 on the
Server configuration ............................................................................................. 44
19. Copy-in and copy-out speed up obtained by using pinned memory on the PTX
backend over PCIe3 on the Server configuration .................................................. 45
Copy-in and copy-out speed up obtained by using pinned memory on the OpenCL backend over PCIe3 on the Server configuration .................................................. 45
Total task-schedule time comparison between pinned and non-pinned memory on the PTX backend over PCIe3 on the Server configuration .................................................. 46
Total task-schedule time comparison between pinned and non-pinned memory on the OpenCL backend over PCIe3 on the Server configuration .................................................. 47
Total task schedule times over PCIe3 on the Laptop with the combinations described in Table 3 for the PTX backend ................................................................. 49
Total task schedule times over PCIe3 on the Laptop with the combinations described in Table 3 for the OpenCL backend ................................................................. 49
Overlap time of computation and data transfers in OpenCL over PCIe3 on the Laptop ...................................................................................................................... 50
Events running asynchronously on different command queues with a batch size of 128 MB .................................................................................................................. 51
Events running asynchronously on different command queues with a batch size of 150 MB .................................................................................................................. 51
Total task schedule times over PCIe3 on the Server with the combinations described in Table 3 for the PTX backend ................................................................. 52
Total task schedule times over PCIe3 on the Server with the combinations described in Table 3 for the OpenCL backend ................................................................. 53
Overlap time of computation and data transfers in OpenCL over PCIe3 on the Server ...................................................................................................................... 53
Page faulting mechanism in the CUDA Unified Memory ........................................... 56
The structure of MaxineVM. The figure is taken from [50] ........................................ 58
Monte Carlo speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects ....................... 62
Black and White speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects ....................... 63
Write Constant speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects ....................... 64
Garbage Collection slowdown obtained from accelerating the Write Constant, Black and White and Monte Carlo benchmarks on the device in the GC benchmark mode . . . 65
Abstract

Nowadays, most programmable systems contain multiple hardware accelerators with different characteristics. In order to use the available hardware resources and improve the performance of their applications, developers must use a low-level language, such as C/C++. Succeeding the same goal from a high-level managed language (Java, Haskell, C#) poses several challenges such as the inability to perform asynchronous data transfers and declare pinned memory. Therefore, managed languages have not established the path of hardware acceleration yet.

Recently, frameworks that run on top of managed runtime systems have been developed, enabling acceleration of high-level programming languages on heterogeneous hardware. In this project, one particular aspect of hardware acceleration in the context of managed runtimes is analyzed, namely memory transfers between the host and the device. Two different solutions for improvement are proposed.

The first solution enhances TornadoVM, a heterogeneous managed runtime system, to allow for pinned off-heap buffers allocation and batch processing that overlaps computation with data transfers. A performance increase in data transfers of up to 50% is obtained when pinned memory is used. Additionally, up to 2.5x in end to end performance speed up can be achieved over sequential batches, when pinned memory is combined with parallel batching.

The second solution extends MaxineVM to allocate its heap through the CUDA Unified Memory, allowing for Java objects resident in the heap to be accessed by the GPU. A performance increase of up to 134x end to end and a garbage collection slowdown of 2.45x compared against sequential Java execution is obtained.
Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given the University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DoculInfo.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, the University Library’s regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in the University's policy on Presentation of Theses.
Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Dr. Christos Kotselidis for the continuous support during my MPhil research. His guidance and knowledge about heterogeneous managed runtime systems throughout the research were proven to be a key factor for the success of this work.

I would also like to thank Dr. Juan Fumero and Dr. Athanasios Stratikopoulos for their continuous mentorship and guidance, especially on the analysis of data and writing.
1 Introduction

With the advent of general purpose graphics processing unit (GPGPU) computing, in the mid-2000s, different parallel computing platforms and application programming interfaces (APIs) have become more prevalent. One of them is the Open Computing Language (OpenCL) [1], an open standard maintained by the Khronos Group, that can be used to write programs that can execute across heterogeneous platforms such as graphics processing units (GPUs), central processing units (CPUs), integrated GPUs, and field programmable gate arrays (FPGAs). Another heterogeneous programming framework is Compute Unified Device Architecture (CUDA) [2], a parallel computing platform, and API designed to work only with Nvidia hardware.

For both OpenCL and CUDA, the main programming languages for expressing compute kernels to run on the accelerator are enhanced dialects of C and C++ [4]. This highlights the fact that these programming frameworks are not easy to use for non-expert programmers, since they require one to explicitly manage memory allocations on the device, transfer data between the host system and the device, write and optimize low-level operations, and use the different memory types that are available on the device.

On the other hand, the Java language is one of the most popular programming languages in the industry [3]. One of the main reasons for this popularity is due to the benefits of a managed runtime, which includes type safety, backward compatibility, platform portability, and most importantly, automatic memory management in the form of garbage collection (GC).

Recently, there have been a number of efforts to enable the use of heterogeneous devices through Java, allowing software engineers using high-level programming languages to abstract the low-level details required when using CUDA or OpenCL. The most notable projects are:

- TornadoVM [5] is an open-source Java-based parallel programming framework that enables dynamic just-in-time (JIT) compilation and execution of Java applications onto OpenCL or CUDA compatible devices.
- Aparapi [6] exposes specific language constructs for memory allocation (i.e., local memory) and memory synchronization (i.e., barriers) that programmers must explicitly use. Similarly to TornadoVM, Aparapi dynamically compiles Java bytecodes to OpenCL code.
- IBM J9 [4] is a Java Virtual Machine (JVM) that contains a JIT compiler used for GPU offloading but exclusively compiles Java 8 Streams to CUDA assembly (PTX) [7].

There are a number of key aspects that must be fulfilled in order to transparently offload code written in a high-level language to an accelerator in the context of a managed runtime:

- Compiler support must exist in order to successfully generate code that can run on the accelerator from a high-level language.
- The runtime must be able to communicate with the device through a device driver in order to offload computations and perform data transfers between the host and the device. A device driver is a computer program that is used to control and operate on
a particular device by providing a software interface to the hardware that the operating system and other programs can use.

- Memory management support must be enhanced in order to perform data transfers between the host and the device. Additionally, optimizations such as removing redundant data transfers can be applied.

The focus of this thesis is to analyze and optimize memory management of managed runtimes on heterogeneous systems. In detail, the contributions of this thesis are as follows:

- The limitations imposed by a managed runtime in the context of their execution on heterogeneous systems regarding memory management and data transfers are identified. It is shown that currently, asynchronous data transfers between the managed heap and the device are unsafe operations that can break the program. Using off-heap buffers that are not managed by the GC restricts the accessibility and limits the programmer to use a specific API. Declaring pinned memory is currently not possible unless done through off-heap buffers or by applying code changes to the managed runtime.

- A solution that enhances TornadoVM to allow for allocation of off-heap pinned buffers is presented in Section 3.2.1. To optimize the presented solution and avoid device memory limitations, parallel batch processing is implemented, allowing for overlap between computation and data transfers. A speed up of data transfers up to 50% is obtained when pinned memory is used. Additionally, a 2.5x end to end performance speed up can be achieved over sequential batches, when pinned memory is combined with parallel batching.

- Another solution is built and presented in Section 4.3 by extending MaxineVM, a research JVM, to allocate its heap through the CUDA Unified Memory, allowing for Java objects resident in the managed heap to be accessed by the GPU. A performance increase of up to 134x end to end and a garbage collection slowdown of 2.45x are obtained compared against sequential Java execution.

The structure of this dissertation is organized as follows:

- In Chapter 2, the current status of heterogeneous computing is analyzed and the programming models of CUDA and OpenCL are explored. An overview of the hardware characteristics of modern GPUs is provided and then the main components of a managed runtime are described. Finally, a logical analysis of what limitations are imposed by a managed runtime on a heterogeneous system is performed.

- Chapter 3 describes a solution to speed up data transfers between the host and the device using Project Panama [8] and TornadoVM and reviews the profiling results obtained.

- Chapter 4 proposes a different solution, by providing non-excessive changes to the JVM (MaxineVM in this case). It is shown that by changing the underlying managed runtime, it is possible to transparently enable object sharing with the GPU. An analysis of the performance of different GPU computations that access data through this method is then performed.

- In Chapter 5, the state of the art research on the same topic is presented.

- Chapter 6 concludes the research and describes possible future work in this direction.
2 Background

This chapter is separated into three sections, as follows:

- Section 2.1 provides an overview of heterogeneous computing along with the benefits and current usages, with a focus on GPUs as the main heterogeneous device. It also contains the description of two popular parallel computing platforms: OpenCL and CUDA.
- Section 2.2 discusses the insights of managed runtimes. In particular, it presents the structure of managed runtimes and their challenges in general.
- Section 2.3 presents the challenges managed runtime systems face in the context of their execution on heterogeneous systems.

2.1 Heterogeneous computing: a short history, description, benefits, and current usages

Since their emergence, GPUs were fixed-function processors used primarily for 3D graphics rendering. The entire execution model was built around a graphics pipeline that was configurable to some extent, but not programmable. By the mid-1990s, the increasing demand for processing power in the video games industry, with notable titles, such as Doom and Quake, made 3D graphics manufacturers (e.g. 3DFX, Nvidia, and ATI) to start providing reasonably priced graphics cards [9].

Even though the graphics pipeline was drastically increasing the processing of pixels per clock cycle, CPUs were still able to produce more triangles than the GPU. Since rendering 3D graphics is an extremely repetitive task and there are no data dependencies between different computations, the solution was to add multiple pipelines, and therefore obtain parallelism. Nowadays, a modern gaming graphics card, such as the Nvidia GeForce RTX 3080 Ti, contains 98 rendering output units (ROPs) and is able to generate around 186 gigapixels per second.

In recent years, several companies have started using parts of the GPU graphics pipeline, especially the pixel shaders, to perform parallel arithmetic calculations, thereby making the emergence of fully unified GPUs, such as the Nvidia GeForce 8800, unavoidable. A unified architecture in a GPU refers to the fact that the hardware in all of the rendering pipeline stages has the same capabilities and uses the same instruction set architecture. The GeForce 8800 was the first GPU to use scalar thread processors rather than vector processors and allowed for memory operations with byte addressing. Additionally, it provided new instructions to express finer-grained parallelism and communication with the host [10]. These features resulted in higher performance general computing and culminated with the emergence of two popular parallel computing platforms, respectively OpenCL [1] and CUDA [2]. Sections 2.1.1 and 2.1.2 describe the hardware architecture and programming models of OpenCL and CUDA.
Nowadays, GPUs have overcome their initial purpose of being used only for graphics rendering and are fully programmable for general computations that can be massively parallelized. Therefore, the main area where general-purpose graphics processing units (GPGPU) are being used has been expanded from video processing to bioinformatics [11], computer vision [12], machine learning [13], cryptography [14], and many others.

Figure 1 shows a comparison in architectures between the CPU on the left side and the GPU, on the right side. The most significant difference is that more transistors are devoted to data processing (green area) on the GPU. This results in multiple arithmetic logic units mapped to the same control unit (yellow area) and in increased parallelism. Also, the size of the level 1 (L1) caches in GPUs is smaller in comparison to a CPU.

2.1.1 CUDA

In November 2006, together with the GeForce 8800, Nvidia introduced CUDA [2], a general-purpose parallel computing platform and programming model that leverages the parallel compute engine of Nvidia GPUs. Its aim is to solve computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C++, FORTRAN, DirectCompute, and OpenACC as a programming language [15]. Parallel Thread Execution (PTX) [7] is an instruction set architecture for general purpose parallel programming and it is designed to be efficient on Nvidia GPUs. High level language compilers for languages such as CUDA and C/C++ generate PTX instructions, which are optimized for and translated to native target-architecture instructions.
Below, in Section 2.1.1.1, CUDA from a hardware perspective is first presented and in Section 2.1.1.2 an analysis is made on how the programming model maps on top of the hardware.

2.1.1.1: CUDA hardware architecture

Nvidia developed a number of slightly different architectures that support the CUDA programming model. The code names for some of the microarchitectures are listed below:

2. Fermi (2010)
4. Maxwell (2014)
5. Pascal (2016)

Figure 2: The CUDA GPU architecture from a hardware perspective. There are multiple Streaming Multiprocessors (SMs) with each SM containing several Scalar Processors (SPs) used for execution, a special Special Function Unit (SFU) and shared memory.

In general, the CPU is called the host, and the GPU is called the device. Even though the architectures are slightly different in detail, Figure 2 shows a general view that covers all of
them. On the GPU, there are multiple Streaming Multiprocessors (SMs), also called "big cores". Each Streaming Multiprocessor holds several Scalar Processors (SPs) that are used for executing instructions. There are other components such as L1 caches and registers, several Special Function Units (SFUs), and a small on-chip memory that is shared between the SPs belonging to the same SM. All the SMs can access the large off-chip dynamic random access memory (DRAM) of the GPU. Data from the host is also written to the GPU DRAM, usually through a peripheral component interconnect express (PCIe) bus.

Each SM executes a number of threads that are assigned by the CUDA runtime. Then, the total number of threads are split into groups of 32 parallel threads called warps and get scheduled to execute on the SM. All of the threads inside a warp start at the same instruction address and will execute the same instruction in a Single Instruction Multiple Threads (SIMT) manner [15]. From the Volta microarchitecture and onwards, threads in a warp that diverge in control flow regions can continue executing concurrently. Prior to Volta, this was not the case, and each sub-warp had to be executed sequentially up to the point where the control flow merged.

2.1.1.2: CUDA programming model

The thread arrangement is a crucial factor when performing parallel computations on the GPU. In CUDA, threads are organized in blocks that can be 1-dimensional, 2-dimensional, or 3-dimensional. At the same time, thread blocks are grouped in grids that can be 1-dimensional, 2-dimensional, or 3-dimensional. The number of thread blocks in a grid is in general dictated by the size of the data being processed, which typically exceeds the number of SPs in the system.

The CUDA API is used to specify the block dimension and grid dimension when launching a function to execute on the GPU. Figure 3 shows the relation between grids, blocks, and threads.
2.1.2 OpenCL

OpenCL was developed by Apple and refined in collaboration with other industrial corporations, including Intel, Nvidia, AMD, Arm, and IBM. The initial proposal was submitted to the Khronos Group, where the initial specification of OpenCL 1.0 was released in December 2008.

Unlike CUDA, OpenCL is an open standard that aims to enable parallel computing on heterogeneous hardware from different vendors. This includes multiple types of devices, such as FPGAs, GPUs, CPUs, and integrated GPUs. Although OpenCL provides guarantees of portability and correctness of kernels across a variety of hardware, it does not guarantee that a particular kernel will achieve peak performance on multiple architectures. The nature of the hardware may make some programming strategies more appropriate for particular platforms than for others [16].

Similar to CUDA, OpenCL defines threads as work-items and blocks as work-groups. Each work-group consists of a number of work-items and both work-groups and work-items can be 1-dimensional, 2-dimensional, or 3-dimensional. A global work-group incorporates all the other work-groups used to run a kernel [17]. Figure 4 shows the relation between the global work-group, local work-groups and work-items.
OpenCL and CUDA have memory models very similar to each other. Figure 5 shows the memory hierarchy using the OpenCL memory model:

- Global memory and Constant memory are GPU off-chip DRAM. Constant memory is read only. CUDA has the same conventions for these kinds of memories.
- Local memory is on-chip memory that is shared between all the work-items of a work-group. CUDA has the equivalent shared memory that can be accessed by different threads in the same SM.
- Private memory is available only to each work-item. In general, it is mapped to registers of the GPU. If too much private memory is used, then it will spill to slower memories such as shared and global. The equivalent in CUDA is local memory.
Figure 5:
Memory hierarchy in OpenCL for GPUs.
Global memory represents the GPU off-chip DRAM. Local memory is on-chip and shared amongst all the work-items belonging to the same work-group. Private memory is available only to each work-item. This figure is taken from [18].

In order to issue commands to the device, OpenCL uses an object called command-queue. Through calls to the API, commands such as read/write operations or kernel launches are queued to a command-queue. The calls to the driver API can be synchronous or asynchronous. When synchronous, the CPU thread that calls the driver API is blocked until the command is executed. When asynchronous, the command is only queued allowing the CPU thread to return from the API call. Once queued in the command-queue, the commands are executed by the device driver. Depending on the properties of the command-queue, the order in which commands are executed can be the same as the order commands were queued (in-order) or can be in an arbitrary order (out-of-order). When commands are queued to a command queue, OpenCL allows for the retrieval of an event object. Ordering of commands in out-of-order command-queues can be established using OpenCL events. CUDA operates in a very similar way, and the equivalent data structure is stream. CUDA streams are always in-order.

The focus of this work is on optimizing data transfers between global memory and host memory in the context of the JVM. To achieve this multiple in-order OpenCL command-queues or CUDA streams are used and asynchronous calls to the driver APIs are performed.
2.2 Managed runtimes

In the 1970s and 1980s, a different way of executing high-level programming languages emerged, in which a layer of software that intermediates between the high-level program and the machine was added. The developed managed runtimes first gained popularity with implementations for BASIC [19], Lisp [20], and Smalltalk [21]. Even though for a long time managed runtimes have been considered slow when compared to natively compiled languages, nowadays most of the high-level programming languages (e.g. Java, Python, Scala, R) are executed on top of a high-performance managed runtime system.

With the emergence of Java and the JVM in the 1990s, the idea of using a managed runtime system as an abstraction layer gained huge popularity. Below is presented a list of advantages offered by managed runtimes, in the context of Java and the JVM:

- Java code can reach native speeds or sometimes be even faster when compared to C/C++. This is achieved by adding one or two optimizing JIT compilers that exploit profiling code at runtime, revealing optimizations that are missed with ahead of time compilation.
- Automatic heap management is performed by improving memory allocation (most of the time it is only a bump pointer allocation), providing garbage collection, and heap resizing.
- A common API can be used regardless of the underlying operating system to perform various tasks (Input/Output, handling network communication).

As part of this thesis, the JVM is the main runtime system that is being used as a technology for the research. The JVM has a complex architecture with details varying between different implementations. However, the main subsystems that are the same between implementations are described below:

- The class loader subsystem loads, links, and initializes classes dynamically at runtime when a class is first referenced by the user code.
- The runtime data area contains the methods area where JIT-compiled code and class level data are being held. The stack area is used to hold Java thread information. Each Java thread has its own runtime stack separately from the native thread stack. Finally, the heap area is part of the runtime data area. Most of the Java objects are allocated in this space (on the heap).
- The execution engine is assigned valid bytecode by the runtime and must execute it either by using the interpreter or the JIT compiler. Another part of the execution engine is the JVM garbage collector that manages the objects on the heap and can move them at different addresses during runtime while preserving the correctness of the program. Different garbage collection algorithms [22] have been implemented across different JVMs but describing them is out of the purpose of this thesis.

Another important feature of the JVM that is used in this thesis is the Java Native Interface (JNI) [23]. The JNI is a native programming interface that allows Java code that runs inside the JVM to interoperate with libraries written in other programming languages such as C or C++. Besides interoperability, JNI also enables programmers to create, inspect and update Java objects, call Java methods from native methods and throw exceptions.
2.3 Challenges in managed runtimes regarding heterogeneous computing

Even though the JVM as a managed runtime system is one of the most popular technologies used nowadays, with the end of Moore's Law [24], the new direction for improvement is to add support for heterogeneity to the platform. Different accelerators can be used to accelerate specific tasks. For example, GPUs perform better by orders of magnitude compared to CPUs when running deep learning algorithms [25]. The following sections present the challenges that managed runtime systems, specifically the JVM, face in the context of heterogeneous systems.

2.3.1 Asynchronous data transfers

In general, asynchronous data transfers between the host and the device are performed, allowing the CPU threads calling the driver API to continue running for the duration of the transfer. The JNI API provides two functions that can be used to lock an object in the JVM heap, `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` [23]. The object is guaranteed to not be moved by the GC in the JVM heap in between the two JNI method calls. Depending on which GC is being used (SerialGC [26], ParallelGC [27], G1GC [28], Shenandoah [29], ZGC [30]) the whole JVM heap might be locked, or a subregion of the heap where the object lives, or only the object itself.

Currently, asynchronous data movements are unsafe in the context of the JVM. Since there is no way to determine when an asynchronous data transfer will execute, it is not possible to know when the object can be unlocked from the heap. If the object is never locked, then the GC is allowed to move it to a different address in the JVM heap, causing the address used by the data transfer to become invalid. If the object is locked but never unlocked, this causes a memory leak in the JVM heap. Both scenarios can lead to the program crashing.

Therefore, in order to transfer objects from the JVM heap, user-level frameworks such as TornadoVM or Aparapi have no choice but to use blocking API calls to OpenCL or CUDA.

2.3.2 Off-heap buffers

An alternative to the use of blocking API calls to perform data transfers is to declare off-heap memory and make it available to the user. Through off-heap memory, it is possible to use non-blocking API calls to the drivers and therefore allow the CPU thread to execute more instructions in the same time frame.

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 it was not designed only for off-heap data.
access, but also for I/O operations and exchange of bulk data. Another issue is that the maximum size of a buffer is limited to 2GB.

As an alternative to the ByteBuffer API, the Foreign Memory Access API [32] has been introduced in Java 14, 15, and 16, as an incubator module (a runtime flag needs to be enabled in order to use it). This API aims to address the problem of accessing and managing off-heap memory in Java, therefore replacing direct byte buffers.

2.3.3 Page-locked memory

Paging out is the process of moving memory areas belonging to a process from fast-to-access random-access memory (RAM) to slow-to-access non-volatile memory. This process is usually used when the RAM memory is fully occupied and the non-volatile memory can be used as a storage space. If the CPU tries to access any memory that was paged out, then the memory is first copied from the non-volatile storage to the RAM (paged in). Memory paging can cause significant overheads in the execution of a program. Pinned (page-locked) memory is guaranteed not to be paged out by the operating system [33]. With pinned memory, it is possible to obtain direct memory access (DMA), without using the CPU [34]. On the other hand, with non-pinned memory, the CPU would have to copy each piece of data to the host buffers. In general, if the CPU speed is the bottleneck causing slow data transfer, pinned memory transfers can be much faster than pageable memory transfers.

Figure 6:
Non-pinned data transfer (left) compared to pinned data transfer (right). Intermediate pinned host buffers are used by the CUDA driver to perform the data transfer to the GPU. This figure is taken from [35].
Figure 6 shows the difference between performing pinned and non-pinned memory transfers in CUDA. In a default data transfer between the host and the device, as shown in the left part of the figure, CUDA copies data from the pageable host memory to intermediate pinned buffers on the host and then performs the data transfer to the device. Even if the copy seems unnecessary, CUDA requires pinned memory to perform the transfer between the host and the device [36]. However, as shown in the right part of the figure, the CUDA API provides options for directly allocating pinned buffers on the host [37], bypassing the extra allocation and copy from non-pinned memory to pinned memory.

In the OpenCL specification, pinned memory is not directly mentioned. However, many driver implementations, such as Nvidia [38] and AMD [39], have special guidelines with the required steps to follow in order to declare and use pinned memory.

Pinned memory in the context of a managed runtime such as the JVM is currently difficult to use since there is no way to instruct the VM to allocate pinned memory. An alternative approach will be presented in Chapter 3, which shows how off-heap buffers can be initialized by using the OpenCL or CUDA APIs. Additionally, Chapter 4 shows how the heap allocation of MaxineVM can be modified to accelerate the transfer of Java arrays from the object heap.
3 Accelerating data transfers in the context of TornadoVM

This chapter presents a proposal to increase the performance of memory management and data transfers of managed runtime systems using heterogeneous hardware. This proposal is built on top of TornadoVM [40], a framework for offloading and accelerating Java methods on various devices such as GPUs, iGPUs, CPUs, and FPGAs. TornadoVM executes on top of the JVM and is able to transparently compile and execute code on heterogeneous hardware. TornadoVM has been developed to provide heterogeneous programming support to the general purpose Java programming language, a language that would not normally be associated with writing either high-performance or heterogeneous code.

Standard JVMs, apart from IBM J9, are only able to generate machine code for various CPU architectures such as X86, ARM, and RISC-V. TornadoVM through the TornadoAPI enhances the JVM, enabling the execution of Java methods on heterogeneous devices such as CPUs, iGPUs, GPUs, and FPGAs.

Specifically, this chapter is split into the following subsections:

- Subsection 3.1 describes the architecture of TornadoVM at a high level view.
- Subsection 3.2 presents the current functionality of batch processing. Besides, it presents how TornadoVM was extended to allow for parallel batch processing.
- Subsection 3.3 demonstrates a solution to the problem of asynchronous data transfers in the context of a managed runtime system (described in Section 2.3.1) using off-heap buffers and pinned memory.
- Subsection 3.4 concludes with a discussion on the evaluation methodology and it shows a detailed view of the performance evaluation.

3.1 Software Architecture of TornadoVM

The software architecture of TornadoVM comprises a combination between a layer architecture and a microkernel architecture. At the top level, TornadoVM exposes an API to program heterogeneous hardware. At the core level, TornadoVM provides an execution engine and a JIT compiler.

The Tornado API provides a task-based parallel API for parallel programming in Java and it enables developers to express parallelism with minimal effort. Each task comprises a Java method handle containing the pure Java code and the data it accesses. The Tornado API provides Java interfaces to create task-schedules. Task-schedules are composed tasks that will be automatically scheduled for execution by the runtime. In addition to enabling definition of tasks and task-schedules, TornadoVM allows developers to indicate that a loop is a candidate for parallel execution through the @Parallel annotation [40].

Listing 1 shows an example written in Java of a parallel map/reduce computation using a TaskSchedule s0 that has two tasks t0 and t1, defined in lines 14 and 15. Map/reduce is a
parallel and distributed computation. It runs a map function on a set of keys K1 and for each key produces an output K2. The reduce function then runs once for each element in K2 to produce a final result. The Java methods to accelerate are specified on each task and are map in lines 2-6 and reduce in lines 7-11. The temp array is the output of task t0 and the input of task t1. The final output of the TaskSchedule is specified in line 16, through the streamOut operation.

```java
public class Compute {
    public void map(float[] in, float[] out) {
        for (@Parallel int i = 0; i < in.length; i++) {
            out[i] = in[i] * in[i];
        }
    }

    public void reduce(float[] in, @Reduce float[] out) {
        for (@Parallel int i = 0; i < in.length; i++) {
            out[0] += in[i];
        }
    }

    public void run(float[] in, float[] out, float[] temp) {
        new TaskSchedule("s0")
            .task("t0", this::map, in, temp)
            .task("t1", this::reduce, temp, out)
            .streamOut(out)
            .execute();
    }
}
```

Listing 1:
Example of using the Tornado API. In lines 13-17 a TaskSchedule is created that has two tasks t0 and t1. The map and reduce methods are passed to the tasks and transparently accelerated. This listing is taken from [40].

Figure 7 is used to describe the workflow of TornadoVM. The components in light blue color are parts of the standard JVM compilation flow. The components in light violet are parts of TornadoVM and are described below:

- The Tornado compiler extends the Graal compiler [41] with extra compilation phases to perform different API replacements for Tornado-specific calls. This produces an Intermediate Representation Graph (IR graph) that is sent to the Dataflow Analyzer.
- The Dataflow Analyzer is an extra compilation phase that performs code reachability and data dependency analysis on the Java methods. It produces information that is passed to the Optimizer.
- The Optimizer uses the data dependency graph and accesses information from the Dataflow Analyzer to produce an internal graph representation and remove unnecessary data transfers between the host and the device. This graph is then passed to the TornadoVM Bytecode Generator.
- The TornadoVM Bytecode Generator traverses the optimized intermediate graph and produces TornadoVM bytecodes in a simple Java Byte Buffer. The TornadoVM bytecodes are shown in Table 1 and explained later.
● The TornadoVM Bytecode Interpreter is a Java switch statement that parses the Java Byte Buffer and executes each TornadoVM bytecode.

● The Device Heap Manager keeps data consistent between the host and the devices. The Optimizer and the Device Heap Manager are used to remove redundant data transfers between the host and the device. For example, In Listing 1, after task t0 executes, the variable temp is held on the device and not copied back to the host memory. Therefore, when task t1 is executed, the data for temp is already present on the device heap.

● The OpenCL (OCL) JIT Compiler and Parallel Thread Execution (PTX) JIT Compiler are two extensions of the Graal compiler with custom compilation phases allowing TornadoVM to generate OpenCL and PTX code. TornadoVM also handles the required calls to the underlying device drivers to perform JIT compilation of the generated OpenCL code and PTX code and load the generated code on the device.

Figure 7:
TornadoVM architecture overview and workflow.

Table 1 enlists the TornadoVM bytecodes generated by the TornadoVM Bytecode Generator and parsed with the TornadoVM Bytecode Interpreter. In total there are 11 bytecodes that allow the virtual machine (VM) to handle data transfers and kernel execution and accept a different number of parameters. All the TornadoVM bytecodes receive at least one argument,
the context identifier, which is a unique number used to identify a task-schedule. TornadoVM generates a new context identifier for each task-schedule in the program. The context identifier is used at run-time to obtain a context object which, among others, contains references to the data accessed by the task-schedule, and information about the device on which the tasks will be executed.

Below is a description of the TornadoVM bytecodes:

- The **BEGIN** and **END** bytecodes are used to signal the beginning and end of a TornadoVM context. The **BEGIN** bytecode is always at the start of Java Byte Buffer and the **END** at the end.
- The **ALLOC** bytecode is used to reserve memory on the heap buffer belonging to the device attached to the context. This bytecode is created when an object passed to a task is write-only and therefore any previous data can be discarded.
- The **STREAM_IN** and **COPY_IN** bytecodes are used to copy data from the host to the device. The difference between **STREAM_IN** and **COPY_IN** is that when a **COPY_IN** is performed, the runtime first checks if the data is marked as present on the device heap. If the data is on the device heap, then no transfer is involved. On the other hand, when the **STREAM_IN** bytecode is executed, a data transfer is guaranteed to be performed.
- The **STREAM_OUT**, **COPY_OUT**, and **COPY_OUT_BLOCK** operations are all analogous to the **STREAM_IN** and **COPY_IN** bytecodes, but perform data transfers from the device heap to the host. The difference between **COPY_OUT** and **COPY_OUT_BLOCK** is that the latter is guaranteed to be a blocking data transfer.
- The **LAUNCH** bytecode is used to launch and asynchronously execute a kernel on the device. During the execution of this bytecode, the runtime also checks a code cache to verify if a kernel has been JIT-compiled already. In the case when a kernel is not available, JIT compilation occurs and the kernel is installed on the device and placed in the code cache.
- The **ADD_DEP** bytecode is used to add execution dependencies between other bytecodes. In TornadoVM, multiple tasks belonging to the same task schedule might execute in parallel. However, when two tasks have a data dependency, they must execute in a certain order that is guaranteed through the **ADD_DEP** bytecode. Parallel execution is disabled by default.
- The **BARRIER** bytecode is used to wait for all the previous operations to finish.

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Operands</th>
<th>Blocking</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>&lt;context&gt;</td>
<td></td>
<td>Creates a new parallel execution context.</td>
</tr>
<tr>
<td>ALLOC</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>No</td>
<td>Allocates a buffer on the target device.</td>
</tr>
<tr>
<td>STREAM_IN</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>No</td>
<td>Performs a copy of an object from host to device.</td>
</tr>
<tr>
<td>COPY_IN</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>No</td>
<td>Allocates and copies an object from host to device.</td>
</tr>
<tr>
<td>STREAM_OUT</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>No</td>
<td>Performs a copy of an object from device to host.</td>
</tr>
<tr>
<td>COPY_OUT</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>No</td>
<td>Allocates and copies an object from device to host.</td>
</tr>
<tr>
<td>COPY_OUT_BLOCK</td>
<td>&lt;context, BytecodeIndex, object&gt;</td>
<td>Yes</td>
<td>A blocking <strong>COPY_OUT</strong> operation.</td>
</tr>
<tr>
<td>LAUNCH</td>
<td>&lt;context, BytecodeIndex, task, Args&gt;</td>
<td>No</td>
<td>Executes a task, compiling it if needed.</td>
</tr>
<tr>
<td>ADD_DEP</td>
<td>&lt;context, BytecodeIndices&gt;</td>
<td>Yes</td>
<td>Adds a dependency between labels.</td>
</tr>
<tr>
<td>BARRIER</td>
<td>&lt;context, BytecodeIndices&gt;</td>
<td>Yes</td>
<td>Waits for all previous bytecodes to be finished.</td>
</tr>
<tr>
<td>END</td>
<td>&lt;context&gt;</td>
<td>Yes</td>
<td>Ends the parallel execution context.</td>
</tr>
</tbody>
</table>

Table 1: List of TornadoVM bytecodes. This table is taken from [40].
3.2 Batch processing in TornadoVM

Another feature of TornadoVM is batch processing. The TornadoVM runtime is able to split the buffers passed to a task into multiple segments of a specified size, called batches, and perform the computation and data transfers for each individual batch. Batch processing can be used if the input and output passed to a task do not fit on the device memory. Choosing a batch size that is less than the device memory size and processing one batch at a time allows for all of the computation to be accelerated on the GPU, even though the data size far exceeds the device memory.

Listing 2 shows how batch processing is used through the Tornado API. A task schedule s0 is created on the first line and the input for the computation of task t0 is split in batches of 32MB, as specified on the second line.

```java
1 TaskSchedule ts = new TaskSchedule("s0")
2 .batch("32MB")
3 .task("t0", Main::transformToBlackAndWhite, arr)
4 .streamOut(arr);
```

Listing 2:
TornadoVM Task Schedule using batch processing

For the computation above, with a batch size of 32MB and an input size of 90MB, the TornadoVM bytecodes generated are shown in Listing 3. The input is split into three batches and for each batch a COPY_IN, LAUNCH and STREAM_OUT bytecode is generated. Each batch uses a different offset to copy from the host to the device and from the device to the host. In this case, the final batch has a smaller size, since the input array size of 90MB can not be split into three equal batches of 32MB. If no batch processing was used, a single COPY_IN, LAUNCH and STREAM_OUT bytecodes would have been generated.

```java
1 COPY_IN size=32000000, offset=0
2 LAUNCH task s0.t0 - transformToBlackAndWhite size=8000000, offset=0
3 STREAM_OUT size=32000000, offset=0
4 COPY_IN size=32000000, offset=32000000
5 LAUNCH task s0.t0 - transformToBlackAndWhite size=8000000, offset=32000000
6 STREAM_OUT size=32000000, offset=32000000
7 COPY_IN size=26000000, offset=64000000
8 LAUNCH task s0.t0 - transformToBlackAndWhite size=6500000, offset=64000000
9 STREAM_OUT_BLOCKING size=26000000, offset=64000000
```

Listing 3:
TornadoVM bytecodes generated when using batch processing.

Figure 8 shows how the current implementation of batch processing works under the example shown in Listing 2 and Listing 3. The initial input array is split into three batches. For each batch a COPY_IN, LAUNCH and STREAM_OUT bytecodes are executed. Each batch is processed in a sequential manner and uses the same memory region on the device heap.
Figure 8: Current implementation of batch processing in TornadoVM. The Task Schedule input array is split into three batches. For each batch COPY_IN, LAUNCH and STREAM_OUT bytecodes are executed. Each batch uses the same memory region in the device heap.
3.2.1 Proposal: Improving performance of Memory Management for GPUs and FPGAs.

As part of this thesis, the current TornadoVM functionality of batch processing is augmented to enable multiple batches to be executed in parallel. This is achieved by splitting the device heap into multiple slots of the size of the batch. For each device heap slot an in-order OpenCL command-queue (or CUDA stream if the TornadoVM PTX backend is used) is created and assigned. Then, each batch (COPY-INs, LAUNCH, COPY-OUTs) is assigned a device heap slot and is enqueued on the command-queue of the device heap slot. All the data transfers associated with the batch and memory accesses performed by the batch kernel operate only on the device heap slot assigned to it. If the size of the input data is larger than the device heap and all the device heap slots are used, then the remaining batches start being assigned from the first indexed command queue. The command-queues used are in-order. CUDA streams are always in-order.

Figure 9 shows how parallel batch processing uses different parts of the device heap for each batch. The first batch occupies the first part of the device heap while the second occupies the second part of the device heap and uses a different command-queue. Finally, because in this example the device heap only fits two batches, the final remaining batch is enqueued using the first command-queue and uses the first heap slot. Additionally, the runtime will trigger a recompilation request for the kernel, to make sure that it accommodates the remaining batch size.
Figure 9: Parallel execution of batch processing in TornadoVM. The Task Schedule input array is split into three batches. For each batch, COPY_IN, LAUNCH and STREAM_OUT bytecodes are executed. The device heap is split into two device heap slots of batch size. To each device heap slot, the runtime assigns a command-queue. The first two batches use different command-queues and batch device heap slots. The final batch is enqueued on the first device heap slot and uses the first command-queue.
Both parallel and sequential batches solve the problem of processing a data set larger than the device memory available. However, parallel batching provides an additional advantage over sequential batches. As analyzed in Section 3.4.2, parallel batches enable the overlap of device computation with data transfers between the host and the device, decreasing the total execution time. This is achieved by having multiple in-order command queues. While a kernel is executed on a command-queue, other command-queues can run data transfers between the host and the device.

Even with the functionality of parallel batch processing implemented, the problem of asynchronous data transfers in the context of a managed runtime system described in Section 2.3.1 is still present. This means that all data transfers performed are blocking, causing the CPU thread running the TornadoVM Bytecode Interpreter to stall and not execute TornadoVM bytecodes in parallel with the data transfers. Therefore no parallelism is achieved only by using this approach. Section 3.3 describes how this challenge is addressed by using off-heap buffers.

### 3.3 Off-heap buffers and pinned memory

As described in Section 2.3.1, performing asynchronous data transfers between the host and the device is an unsafe operation in the context of a managed runtime system. The garbage collector can move the data to a different location during the transfer, causing invalid data to be copied and most probably result in corrupt 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.

Another limitation when using objects allocated in the JVM heap is that they do not live in pinned memory. Pinned memory must be allocated through the driver API (OpenCL or CUDA) and the JVM does not provide a way to allocate objects in different memory pools or even to define such memory pools.

A solution to this is to use off-heap buffers, that are not managed directly by the JVM and that the garbage collector will not move. As part of this work, this feature is implemented in TornadoVM by using Project Panama [8]. Project Panama is an OpenJDK project designed to provide a new API for interconnecting Java with native code and includes multiple components: support for native function calls, native library management APIs, and header file extraction tools. As part of this new approach, Project Panama provides an API to create MemorySegments that are off-heap and to access them at different addresses.

In order to be able to accelerate data transfers, the TornadoVM runtime must be aware of the buffers allocated through pinned memory. Therefore, the Tornado API is enhanced to allow for allocation of off-heap buffers that are backed by a buffer on the target device. In Listing 4 in line 20 it is shown how the Tornado API is used to allocate an off-heap pinned MemorySegment of size arraySizeBytes backed by a device buffer of the same size on the default device. Next, in lines 22-25 a TaskSchedule is created with a single task where a MemorySegment pinnedSegment is used as a parameter. The function to be accelerated,
transformToBlackAndWhite, then accesses the MemorySegment image through the MemoryAccess::getIntAtIndex and MemoryAccess::setIntAtIndex functions in lines 4 and 12.

```
1 private static void transformToBlackAndWhite(MemorySegment image) {
2     int size = (int) image.byteSize() / Integer.BYTES;
3     for (@Parallel int i = 0; i < size; i++) {
4         int rgb = MemoryAccess.getIntAtIndex(image, i);
5         int alpha = (rgb >> 24) & 0xff;
6         int red = (rgb >> 16) & 0xFF;
7         int green = (rgb >> 8) & 0xFF;
8         int blue = (rgb & 0xFF);
9         int grayLevel = (red + green + blue) / 3;
10        int gray = (alpha << 24) | (grayLevel << 16) | (grayLevel << 8) | grayLevel;
11        MemoryAccess.setIntAtIndex(image, i, gray);
12    }
13 }
14 }
15 
16 public static void main(String[] args) {
17     long arraySizeBytes = ((long) SIZE) * Integer.BYTES;
18     
19     TornadoDevice defaultDevice =
20     TornadoCoreRuntime.getTornadoRuntime().getDriver(1).getDefaultDevice();
21     MemorySegment pinnedSegment =
22     TornadoCoreRuntime.getTornadoRuntime().getOffHeapBuffer(defaultDevice, arraySizeBytes);
23     
24     TaskSchedule ts = new TaskSchedule("s0").batch("32MB")
25            .task("t0", Main::transformToBlackAndWhite, pinnedSegment)
26            .streamOut(pinnedSegment);
27     ts.execute();
28 }
```

Listing 4:

Usage of the Tornado API to create off-heap pinned memory backed by a device buffer of the same size. In line 20 the allocation is performed and in lines 22-25 the TaskSchedule is defined and executed. The MemoryAccess API is used in lines 4 and 12 to read and write to the device buffer.

Support for allocating both pinned and non-pinned off-heap memory was added to TornadoVM. Listing 5 shows the method that contains the main logic for allocating a pinned MemorySegment on the host that is backed by a memory buffer on the device for the PTX backend:

- In the second line a PTXBufferInfo is created that is later populated with meta-data used by the runtime.
- Line 5 allocates the pinned memory on the host and line 10 allocates a buffer on the device.
In line 15 a Project Panama’s `MemoryAddress` object is obtained from the raw `hostBufferAddress` and in line 18 a `MemorySegment` starting at `address` and with size `hostBufferSize` is created.

- In line 21, the `PTXBufferInfo` is registered with the runtime and the memory segment is returned.

```java
private MemorySegment allocatePinnedBuffer(long hostBufferSize, long deviceBufferSize) {
    PTXBufferInfo bufferInfo = new PTXBufferInfo();
    // Allocate the page-locked buffer on the host.
    long hostBufferAddress = getMemoryProvider().allocatePinnedBuffer(hostBufferSize);
    bufferInfo.setHostBufferPointer(hostBufferAddress);
    bufferInfo.setHostBufferSize(hostBufferSize);
    
    // Allocate a mirror buffer on the device.
    long deviceBufferAddress =
        getDeviceContext().getDevice().getPTXContext().allocateMemory(deviceBufferSize, false);
    bufferInfo.setDevicePointer(deviceBufferAddress);
    bufferInfo.setDeviceBufferSize(deviceBufferSize);
    
    // Obtain a memory address object.
    MemoryAddress address = MemoryAddress.ofLong(hostBufferAddress);
    
    // Create an unrestricted memory segment from the address.
    MemorySegment segment =
        address.asSegmentRestricted(hostBufferSize).share();
    
    // Register the buffer, to be able to release it on device.reset().
    getDeviceContext().registerOffHeapBuffer(segment, bufferInfo);
    
    return segment;
}
```

Listing 5:
Allocation of pinned off-heap buffers in TornadoVM. A meta-data `bufferInfo` is created on the second line. Line 5 allocates the pinned buffer on the host and line 10 allocates a buffer on the device. Line 15 obtains a `MemoryAddress` object using the address returned by the CUDA API and line 18 creates a `MemorySegment` of the given size. The meta-data object is registered with the runtime in line 21 and the memory segment is then returned.

The logic to allocate pinned/non-pinned memory in the OpenCL backend is very similar to Listing 5. In case of allocating non-pinned memory, a `MemorySegment` is directly allocated instead of creating one using a raw address from the drivers.

A key feature of the Graal compiler, which is used in TornadoVM, is the Compiler Plugins. These allow for the addition or replacement of method invocations with IR subgraphs during the IR graph building phase [42]. TornadoVM already uses Invocation Plugins for other
functionalities such as Java atomics support. In order to support MemorySegments, Invocation Plugins for read and write functions in the MemoryAccess class were added.

Listing 6 shows how an Invocation Plugin is registered for the function MemoryAccess::set#kindAtIndex, where #kind can be any primitive type. In lines 4 and 5 a memory address is computed based on the index passed to the function. Then, in line 6 a JavaWriteNode is created and in line 7 is added to the graph as a replacement of the method invocation.

```java
1  r.register3("set" + kind.name() + "AtIndex", MemorySegment.class, long.class, kind.toJavaClass(), new InvocationPlugin() {
2         @Override
3             public boolean apply(GraphBuilderContext b, ResolvedJavaMethod targetMethod, Receiver receiver, ValueNode segment, ValueNode index, ValueNode value) {
4                 MulNode mulNode = b.append(new MulNode(index, ConstantNode.forInt(kind.getByteCount())));
5                 AddressNode addressNode = b.append(new OffsetAddressNode(segment, mulNode));
6                 JavaWriteNode writeNode = new JavaWriteNode(kind, addressNode, LocationIdentity.any(), value, OnHeapMemoryAccess.BarrierType.NONE, false);
7                 b.add(writeNode);
8                 return true;
9             }
10 });
```

Listing 6: Graal Invocation Plugin for replacing calls to MemoryAccess::set#kindAtIndex with a JavaWriteNode.

Figure 10 shows the Intermediate Representation (IR) graph obtained after triggering a compilation of the transformToBlackAndWhite method in Listing 4 and right after the graph is built and no optimizations have been applied. Only the loop body is shown. The nodes connected with a red line are fixed and are part of the control flow, meaning that they cannot be reordered in the graph. The nodes connected with a blue line are floating and are part of the data flow, meaning that they have more motion freedom in the graph. The Graal Invoke nodes that should represent the calls made to the MemoryAccess class are replaced by JavaRead and JavaWrite nodes, respectively nodes 28 and 50. The same address node, OffsetAddress indexed with 27, is used for the JavaRead and JavaWrite nodes.
Intermediate Representation (IR) graph after the Graph Builder phase obtained from compiling the `transformToBlackAndWhite` method in Listing 4; only the loop body is shown. The nodes connected with a red line are fixed and are part of the control flow, meaning that they can not be reordered in the graph. The nodes connected with a blue line are floating and are part of the data flow, meaning that they have more motion freedom in the graph. Note that there are no Invoke nodes to represent the calls made to the `MemoryAccess` class. The read and write to memory performed through the `MemoryAccess` class are replaced by `JavaWrite` and `JavaRead` nodes by the Invocation Plugins.

Listing 7 shows the loop of the OpenCL kernel obtained from compiling the `transformToBlackAndWhite` method from Listing 4. Note that the `JavaRead` node from Figure 10 generates the read from global memory in line 23. Similarly, the `JavaWrite` node from Figure 10 generates the write to global memory in line 40. The same functionality is used for the PTX backend.

```
1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
```
#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable
__kernel void transformToBlackAndWhite(__global uchar * _heap_base, ulong _frame_base, __constant uchar * _constant_region, __local uchar * _local_region, __global int * _atomics)
{
  ulong ul_0, ul_5;
  int i_24, i_21, i_20, i_23, i_22, i_1, i_2, i_9, i_8, i_11, i_10, i_7, i_6, i_17, i_16, i_19, i_18, i_13, i_12, i_15, i_14;
  long l_3, l_4;

  __global ulong * _frame = (__global ulong *) & _heap_base[_frame_base];

  // BLOCK 0
  ul_0 = (ulong) _frame[3];
  i_1 = get_global_id(0);

  // BLOCK 1 MERGES [0 2]
  i_2 = i_1;
  for(;i_2 < 8000000;)
  {
    // BLOCK 2
    l_3 = (long) i_2;
    l_4 = l_3 << 2;
    ul_5 = ul_0 + l_4;
    i_6 = *((__global int *) ul_5);
    i_7 = i_6 >> 24;
    i_8 = i_7 & 255;
    i_9 = i_8 << 24;
    i_10 = i_6 & 255;
    i_11 = i_6 >> 16;
    i_12 = i_11 & 255;
    i_13 = i_6 >> 8;
    i_14 = i_13 & 255;
    i_15 = i_12 + i_14;
    i_16 = i_10 + i_15;
    i_17 = i_16 / 3;
    i_18 = i_17 << 16;
    i_19 = i_9 | i_18;
    i_20 = i_17 << 8;
    i_21 = i_19 | i_20;
    i_22 = i_21 | i_17;
    *((__global int *) ul_5) = i_22;
    i_6 = i_22;
    i_23 = get_global_size(0);
    i_24 = i_23 + i_2;
44    i_2 = i_24;
45 }   // B2
46   // BLOCK 3
47 return;
48 }   // kernel

Listing 7:
The OpenCL kernel obtained from compiling the `transformToBlackAndWhite` method shown in Listing 4. The `JavaRead` node from Figure 10 generates a read from global memory, shown in line 23. The `JavaWrite` node from Figure 10 generates a write to global memory, shown in line 40.

By allocating the memory buffers off-heap, it is possible to safely perform asynchronous data transfers between the host and the device. This enables the TornadoVM bytecodes executed by the TornadoVM Bytecode Interpreter to be non-blocking. Additionally, off-heap buffers allow for the allocation of pinned memory, increasing the speed of data transfers.

3.4 Experimental Performance Evaluation

The off-heap buffers implementation in TornadoVM is evaluated under two different scenarios. The first scenario (Section 3.4.1) presents an analysis of the data transfer times for pinned and non-pinned memory without batch processing, while the second scenario (Section 3.4.2) presents the equivalent analysis when sequential and parallel batching is applied - essentially evaluating the combination of the two optimizations described thus far in this thesis.

The experiments that explore the two scenarios are performed on two machines equipped with different GPUs of different classes (low-end and high-end). The configuration of these machines is listed in Table 2. The first machine is a laptop with an Intel i7 @ 4.5 GHz CPU and an Nvidia GeForce GTX 1650 GPU with 4GB of RAM. The second is a server equipped with an Intel i7 @ 4.5 GHz and an Nvidia Quadro GP100 with 16GB of RAM.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Laptop</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Core(TM) i7-9750H</td>
<td>Intel(R) Core(TM) i7-7700K</td>
</tr>
<tr>
<td>Maximum frequency</td>
<td>4.5 GHz</td>
<td>4.5 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia GeForce GTX 1650</td>
<td>Nvidia Quadro GP100</td>
</tr>
<tr>
<td>GPU RAM</td>
<td>4GB</td>
<td>16GB</td>
</tr>
<tr>
<td>PCI express version</td>
<td>3 (16 lanes)</td>
<td>3 (8 lanes)</td>
</tr>
<tr>
<td>PCI interconnect maximum theoretical bandwidth</td>
<td>15760 MB/s</td>
<td>7880 MB/s</td>
</tr>
</tbody>
</table>
Both experiments run the `transformToBlackAndWhite` Java method shown in Listing 4. The `transformToBlackAndWhite` method iterates once over the elements of a `MemorySegment` and for each element performs a read and a write to memory. Additionally, it performs multiple bit manipulation operations. This method is an ideal candidate since it contains an average amount of computation and represents a realistic use case. Before collecting performance numbers, 50 warmup executions are performed to ensure that JVM has JIT compiled all the hot paths of execution in TornadoVM. Then, the average mean of the next 100 iterations is used to obtain the final end-to-end time.

A data transfer from the host to the device is referred to as `copy-in` and a data transfer from the device to the host is referred to as `copy-out`.

### 3.4.1 Data transfer times using pinned vs non-pinned memory without batch processing

Subsection 3.4.1.1 presents the data transfer performance figures obtained on a PCIe3 interconnect with 16 lanes that has a theoretical maximum bandwidth of 16GB/s for both read and write operations.

Subsection 3.4.1.2 presents the data transfer performance figures obtained on a PCIe3 interconnect with 8 lanes that has a theoretical maximum bandwidth of 8GB/s for both read and write operations.

Figures for both the OpenCL and PTX backends are included in the two subsections.

#### 3.4.1.1 Laptop performance evaluation

This subsection presents the performance figures obtained from running the Java method `transformToBlackAndWhite` (Listing 4) on the Laptop as described in Table 2.

<table>
<thead>
<tr>
<th>TornadoVM version</th>
<th>0.11</th>
<th>0.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS version</td>
<td>Ubuntu 20.04</td>
<td>CentOS Linux 7</td>
</tr>
<tr>
<td>Kernel version</td>
<td>5.11.0-27</td>
<td>3.10.0-693.17.1</td>
</tr>
<tr>
<td>OpenJDK version</td>
<td>OpenJDK 16 GraalVM CE 21.1.0</td>
<td>OpenJDK 16 GraalVM CE 21.1.0</td>
</tr>
<tr>
<td>OpenCL driver version</td>
<td>OpenCL 3.0 CUDA 11.4.94</td>
<td>OpenCL C 1.2 384.111</td>
</tr>
<tr>
<td>CUDA driver version</td>
<td>470.57.02</td>
<td>384.111</td>
</tr>
</tbody>
</table>

Table 2: System configuration used to run the benchmark.
Figure 11 shows the times in milliseconds over PCIe3 for copy-in (blue), copy-out (red), copy-in using pinned memory (yellow) and copy-out using pinned memory (green) across different input sizes on the PTX backend. The key observation is that as the size of data transfers increases, the difference in time between pinned and non-pinned copies also increases. For a data transfer of 128MB, the difference between a non-pinned copy-in and a pinned copy-in is 1.62 ms. For a data transfer of 1024MB, the difference is 12.9 ms.

Similarly to Figure 11, Figure 12 shows the data for the same experiment but obtained through calls to OpenCL in order to perform the data transfers. All the data transfer times are similar between the two backends, except for the copy-in time of non-pinned memory. For the OpenCL backend, when performing a data transfer of 1024 MB the value is 103 ms, compared to the 91 ms obtained on the PTX backend. The reason for this discrepancy can be attributed to the actual implementation of the OpenCL and CUDA drivers provided by Nvidia which are closed source.
Figure 12: Data transfers with pinned and non-pinned memory in OpenCL over PCIe3 on the Laptop configuration. With non-pinned memory, in blue is the copy-in time and red is the copy-out time. With pinned memory, yellow is the copy-in time and green is the copy-out time.

Figures 13 and 14 show the speed up obtained from using pinned memory compared to non-pinned memory on the PTX and OpenCL backends. On the PTX backend, pinned memory is between 1.11x and 1.16x faster than non-pinned memory for both copy-ins and copy-outs, regardless of the data size used. On the OpenCL backend, copy-ins using pinned memory are between 1.25x and 1.34x faster than non-pinned memory transfers. Regarding pinned copy-out transfers, they are between 1.12x and 1.14x faster than non-pinned copy-out transfers.
Figure 13:
Copy-in and copy-out speed up obtained by using pinned memory on the PTX backend over PCIe3 on the Laptop configuration. A speed up of 1 means that the copy-in with pinned memory has the same performance as a copy-in with non-pinned memory.

Figure 14:
Copy-in and copy-out speed up obtained by using pinned memory on the OpenCL backend over PCIe3 on the Laptop configuration. A speed up of 1 means that the copy-in with pinned memory has the same performance as a copy-in with non-pinned memory.
Figure 15 shows the overall performance improvement of the total execution time obtained by running the `transformToBlackAndWhite` method in Listing 4 for the PTX backend when non-pinned memory is used (blue) and pinned memory is used (red). Using pinned memory results in a task schedule time that is between 1.11x and 1.14x faster compared to using non-pinned memory, regardless of the data size.
Figure 16 shows the overall performance improvement of the total execution time obtained by running the `transformToBlackAndWhite` method in Listing 4 for the OpenCL backend when non-pinned memory is used (blue) and pinned memory is used (red). Using pinned memory results in a task schedule time that is between 1.17x and 1.21x faster compared to using non-pinned memory, regardless of the data size.

3.4.1.2 Server performance evaluation

This subsection presents the performance figures obtained from running the Java method `transformToBlackAndWhite` (Listing 4) on the Server as described in Table 2.

Figure 17 shows the times in milliseconds over PCIe3 for copy-in (blue), copy-out (red), copy-in using pinned memory (yellow) and copy-out using pinned memory (green) using different input sizes on the PTX backend. The difference between pinned and non-pinned data transfers in this case is not significant. The most considerable difference can be seen with a data size of 1024 MB, when pinned copy-ins are faster by 9.01 ms than non-pinned copy-ins. The difference for copy-outs between pinned and non-pinned transfers with a data size of 1024 MB is only 1.63 ms. The reason for the significant difference between non-pinned copy-ins and non-pinned copy-outs is that the bandwidth reached by the copy-ins is lower than the bandwidth reached by the copy-outs. On the other hand, pinned copy-ins and copy-outs reach similar data transfer bandwidths.
Like Figure 17, Figure 18 shows the data for the same experiment but obtained through calls to OpenCL in order to perform the data transfers. All the data transfer times are similar between the two backends, except for the copy-in time of non-pinned memory. For the OpenCL backend, the time of a data transfer of 1024 MB is 234.36 ms, compared to the 167.42 ms obtained on the PTX backend. A similar discrepancy is also reported in [43] and the reason for it is not known since the OpenCL and CUDA drivers provided by Nvidia are closed source.
Figures 18 and 20 show the speed up obtained from using pinned memory compared to non-pinned memory on the PTX and OpenCL backends. On the PTX backend, pinned memory is between 1.01x and 1.06x faster than non-pinned memory for both copy-ins and copy-outs, regardless of the data size used. On the OpenCL backend, copy-ins using pinned memory are between 1.44x and 1.52x faster than non-pinned memory transfers. Regarding pinned copy-out transfers, they are 1.01x faster than non-pinned copy-out transfers.
Figure 19:
Copy-in and copy-out speed up obtained by using pinned memory on the PTX backend over PCIe3 on the Server configuration. A speed up of 1 means that the copy-in with pinned memory has the same performance as a copy-in with non-pinned memory.

Figure 20:
Copy-in and copy-out speed up obtained by using pinned memory on the OpenCL backend over PCIe3 on the Server configuration. A speed up of 1 means that the copy-in with pinned memory has the same performance as a copy-in with non-pinned memory.
Figure 21: Total task-schedule time comparison between pinned and non-pinned memory on the PTX backend over PCIe3 on the Server configuration.

Figure 21 shows the overall performance improvement of the total execution time obtained by running the `transformToBlackAndWhite` method in Listing 4 for the PTX backend when non-pinned memory is used (blue) and pinned memory is used (red). Using pinned memory results in a task schedule time that is 1.03x faster compared to using non-pinned memory, regardless of the data size.
3.4.2 Batch processing times using pinned vs non-pinned memory

Subsection 3.4.2.1 presents the batch processing figures obtained on a PCIe3 interconnect with 16 lanes that has a theoretical maximum bandwidth of 16GB/s for both read and write operations.

Subsection 3.4.2.2 presents the batch processing figures obtained on a PCIe3 interconnect with 8 lanes that has a theoretical maximum bandwidth of 8GB/s for both read and write operations.

CUDA provides a way to measure the time elapsed in milliseconds between two events. The current implementation of the TornadoVM profiler wraps the driver commands (write, read, kernel launch) with a before and after event. Then, it measures the time difference between the two events in order to get the duration of the driver command. With the current implementation of the TornadoVM profiler in the PTX backend, it is not possible to measure absolute times and order events. In OpenCL, an absolute time is returned and it is possible
to order events based on this value. Therefore, in this section some figures for the PTX backend are not shown due to this limitation.

In total six configurations of pinned/non-pinned and sequential/parallel batches were used and are described in Table 3. The abbreviations are then used in the explanation and map to the figure’s bar colors. N/A in the Parallel batches column means that batches were not used in the configuration (the input/output was copied in a single data transfer). The convention used to derive the shorthands is:

- Prepend N for “Non”
- P for “Parallel batches”
- M for “Pinned Memory”

<table>
<thead>
<tr>
<th>Bar color</th>
<th>Parallel batches</th>
<th>Pinned memory</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>N/A</td>
<td>No</td>
<td>A</td>
</tr>
<tr>
<td>Red</td>
<td>N/A</td>
<td>Yes</td>
<td>B</td>
</tr>
<tr>
<td>Yellow</td>
<td>No</td>
<td>No</td>
<td>NPNM</td>
</tr>
<tr>
<td>Green</td>
<td>Yes</td>
<td>No</td>
<td>PNM</td>
</tr>
<tr>
<td>Orange</td>
<td>No</td>
<td>Yes</td>
<td>NPM</td>
</tr>
<tr>
<td>Teal</td>
<td>Yes</td>
<td>Yes</td>
<td>PM</td>
</tr>
</tbody>
</table>

Table 3: Configuration of sequential/parallel batches and pinned/non-pinned memory used.

3.4.2.1 Laptop batch processing

This subsection presents the performance figures obtained from running the Java method shown in Listing 4 on the Laptop as described in Table 2 with batch processing enabled and with different batch sizes.

Figures 23 and 24 show the task schedule times obtained from running with a MemorySegment of size 1024 MB and with different batch sizes for the PTX and the OpenCL backends, respectively.

As shown in Figure 23, for the PTX backend, PM batches (teal bars) are the fastest and finish in 100 ± 3 ms for batch sizes up to 128MB, providing a speed up of 2x compared to NPNM batches. After the 128MB batch size, the execution time of PM batches (teal bars) slightly increases due to less overlap between computation and data transfers (as later shown in Figure 25). The reason for the time increase in the 1024 MB batch size is that there is a single batch to be computed, and therefore the implementation is equivalent to B (the red bar). Using PNM batches (green bar) provides no speed-up compared to NPNM (yellow bars) and A (blue bar).
Analogous to Figure 23, Figure 24 shows the total task schedule times for the OpenCL backend. Running PM batches (teal bars) is the fastest implementation and finishes in 100 ± 6 ms for batch sizes up to 128MB. PM batches provide a speed up of 2.5x over NPNM batches when the batch size is 8 MB and a speed up of 2.02x when the batch size is 128 MB. After the 128MB batch size, the execution time of PM batches (teal bars) slightly increases due to less overlap between computation and data transfers (as later shown in Figure 25). For a batch size of 1024 MB, there is a single batch to be computed, and therefore the implementation is equivalent to B (the red bar). Using PNM batches (green bar) provides a speed-up of ~1.23x compared to NPNM (yellow bars) for batch sizes up to 32 MB.
Figure 25 is used to explain why PNM batches (green bars) can sometimes perform better than NPNM batches (yellow bars) in the case of OpenCL. The overlap between data transfer and computation is shown when using different batch sizes. With PM batches there is always overlap regardless of the batch size. The most overlap occurs with batch sizes up to 64 MB and is in the range of 55 ± 5 ms. However, when non-pinned memory is used, with PNM batches, overlap only occurs with very small batch sizes, the most overlap being 20 ms with a batch size of 32 MB.

Figure 25 also shows the overlap between data transfer and computation for some irregular sizes, such as 150MB and 384MB. In order to understand the observed values, an analysis of the overlapping between queue indices is presented in Figures 26 and 27. In both figures, on the vertical axis the queue index is used and on the horizontal axis the time in milliseconds is shown. As mentioned in Section 3.2, a batch is composed of three TornadoVM bytecodes: a copy-in (denoted by a red vertical line), a kernel launch (a green vertical line in the figure) and a copy-out (a blue vertical line in the figure). The overlap between computation (green lines) and data transfers (red or blue lines) can be observed in both figures. Figure 26 shows a timeline of running the experiment with a batch size of 128 MB, while Figure 27 uses a batch size of 150 MB. As presented in Section 3.2, when the total input array can not be split into batches of equal size, the TornadoVM runtime will trigger a recompilation for the final batch to accommodate for the latest size. Since the input size is 1024 MB, a batch size of 150 MB will determine the final batch to have a size of 124 MB. The observed behaviour is that when the runtime thread running the TornadoVM Bytecode Interpreter triggers a recompilation, it is blocked until all events in the command-queues have finished; that is until all the previous batches have finished processing. Therefore, for the final batch shown in Figure 27 that uses queue index 1, the copy-in is performed and overlaps computation, but the execution of the kernel and the
copy-out are then blocked due to the requirement to recompile, with no overlap observed for the final kernel execution.

Figure 26:
Events running asynchronously on different command queues with a batch size of 128 MB.

Figure 27:
Events running asynchronously on different command queues with a batch size of 150 MB.
3.4.2.2 Server batch processing performance evaluation

This subsection presents the performance numbers obtained from running the Java method shown in Listing 4 on the Server (Table 2) with batch processing enabled and with different batch sizes.

Figures 28 and 29 show the task schedule times obtained from running with a MemorySegment of size 1024 MB and with different batch sizes for the PTX and the OpenCL backends, respectively.

As shown in Figure 28, for the PTX backend, running PM batches (teal bars) are the fastest and finish in 200 ± 10 ms for batch sizes up to 128MB, providing a speed up of ~1.67x compared to NPNM batches. After the 128MB batch size, the execution time of PM batches (teal bars) slightly increases due to less overlap between computation and data transfers (as later shown in Figure 30). The reason for the time increase in the 1024 MB batch size is that there is a single batch to be computed, and therefore the implementation is equivalent to B (the red bar). Using PNM batches (green bar) provides no speed-up compared to NPNM (yellow bars) and A (blue bar).

![Nvidia GeForce 1650 - combinations of sequential/parallel and pinned/non-pinned batches](image)

**Figure 28:**
Total task schedule times over PCIe3 on the Server with the combinations described in Table 3 for the PTX backend.

Analogous to Figure 28, Figure 29 shows the total task schedule times for the OpenCL backend. Running PM batches (teal bars) is the fastest implementation and finishes in 190 ± 4 ms for batch sizes up to 128MB. PM batches provide a speed up of 2.45x over NPNM batches when the batch size is 32 MB and a speed up of 2.29x when the batch size is 128 MB. After the 128MB batch size, the execution time of PM batches (teal bars) slightly increases due to less overlap between computation and data transfers (as later shown in Figure 30). Once again, for a batch size of 1024 MB, there is a single batch to be computed, and therefore the implementation is equivalent to B (the red bar). Using PNM batches (green bar) is almost as fast as PM batches (teal bar) and even faster than NPM batches (orange
bar) for batch sizes up to 64 MB. PNM batches (green bar) provide a speed-up of ~2.01x compared to NPNM (yellow bars) for batch sizes up to 64 MB. NPNM batches also perform faster by 40ms when the batch size is 1024MB, and therefore a single batch is used compared to using different batch sizes.

![Graph showing total task schedule times over PCIe3 on the Server with the combinations described in Table 3 for the OpenCL backend.](image)

**Figure 29:**
Total task schedule times over PCIe3 on the Server with the combinations described in Table 3 for the OpenCL backend.

Figure 30 is used to explain why PNM batches (green bars) give a speed up compared to NPNM batches (yellow bars) and are almost as fast as PM batches (teal bars) in the case of OpenCL. The overlap between data transfer and computation is shown when using different batch sizes. With PM batches there is always overlap regardless of the batch size. The most overlap occurs with a batch size of 32 MB and is 11 ms. However, when non-pinned memory is used, with PNM batches, overlap only occurs with batch sizes up to 64MB, the most overlap being 3.9 ms with a batch size of 32 MB.

![Graph showing overlap time of computation and data transfers in OpenCL over PCIe3 on the Server.](image)

**Figure 30:**
Overlap time of computation and data transfers in OpenCL over PCIe3 on the Server.
Similarly to Figure 25, Figure 30 includes the overlap between data transfer and computation for irregular sizes, such 150 MB and 384 MB. The reason the overlap of a batch size of 150 MB is less than that of 256 MB and the overlap of 384 MB is less than that of 512 MB is that the TornadoVM runtime will trigger a recompilation for the final batch to accommodate for the latest size. Since the input size is 1024 MB, a batch size of 150 MB will determine the final batch to have a size of 124 MB. The observed behaviour is that when the runtime thread running the TornadoVM Bytecode Interpreter triggers a recompilation, it is blocked until all events in the command-queues have finished; that is until all the previous batches have finished processing. The blocked CPU thread causes the final kernel to be scheduled late and not perform any overlap with a data transfer.
4 Using CUDA Unified Memory for object allocation in MaxineVM

This chapter presents another approach to increase the performance of memory management and data transfers of managed runtime systems using heterogeneous hardware. This proposal is built on top of MaxineVM [44], a research VM for Java written in Java and uses CUDA Unified Memory (UM) [45]. In this proposal it is shown how non-excessive changes to the managed runtime system can enable allocation of the whole JVM heap through the CUDA Unified Memory API, allowing the programmer to share data from the JVM heap with the GPU without performing any explicit data transfers. Specifically, this chapter is split into the following subsections:

- Subsection 4.1 describes the CUDA Unified Memory.
- Subsection 4.2 presents MaxineVM at a high level.
- Subsection 4.3 presents the changes implemented in MaxineVM in order to allocate the JVM heap through CUDA Unified Memory.
- Subsection 4.4 describes an abstraction layer added to enable acceleration of CUDA kernels using Java objects.
- Subsection 4.5 concludes with a discussion on the evaluation methodology and the performance results.

4.1 The CUDA Unified Memory

One major challenge when programming heterogeneous systems arises from the physically separate memories of the host and the device. Kernel execution on the GPU can only access data that is stored on the device memory. Thus, programmers need to manually manage the data transfers between the host and the device or use various frameworks such as TornadoVM or Aparapi. CUDA Unified Memory [45] addresses this challenge by providing a single and consistent view of the host and device memories. Currently, modern CPUs support 48-bit memory addresses, while UM supports 49-bit memory addresses. This means that through UM, it is possible to access data resident in both the CPU’s RAM and GPU’s RAM using the same virtual address space.

When a page that is not mapped to a physical address on the current device (CPU or GPU) is accessed, a page fault is triggered. The memory system holding the requested page unmaps it and then the page is migrated to the memory of the faulting device [46]. Figure 31 shows the steps of handling a page fault, when the CPU attempts to access a page that is resident on the GPU. The page is unmapped from the GPU, migrated and mapped to the CPU and then the write is performed.

Resolving a page fault has a high overhead [47] and can significantly decrease the system performance when the same memory pages are accessed by the CPU and GPU, resulting in memory thrashing. The massive parallelism on the GPU further exacerbates the page fault overhead because processing stalls while page faults are being resolved, and multiple
threads in different warps accessing the same page can cause multiple duplicated faults [48].

Figure 31:
Page faulting mechanism in the CUDA Unified Memory. The CPU writes to a page that is resident on the GPU. The page is first migrated from GPU memory to CPU memory and then the write is performed.

In general, GPUs have a small memory capacity compared to the system memory on CPUs. One major limitation when porting applications to GPUs is to overcome their memory capacity to enable larger datasets. After the Pascal architecture, it is possible to oversubscribe UM, allowing GPU kernels to use more memory than what is available on the device. The memory oversubscription is handled through the page faulting mechanism. When a page fault occurs and all the device memory is used, an unused page is evicted to the CPU according to the least recently used (LRU) replacement policy to make space for the new pages. Page faulting when the GPU memory is already fully occupied is slower than page faulting when the GPU memory is not fully occupied. The reason for this is that an old memory page must be first moved from the GPU to the CPU, before migrating the new memory page from the CPU to the GPU.

TornadoVM is a framework that executes on top of standard JVMs and is able to execute Java methods on heterogeneous hardware. Using UM in TornadoVM is not reasonable, since TornadoVM already performs automatic data management between the host and the devices. Additionally, UM could only be allocated through off-heap buffers that are not managed by the GC since TornadoVM cannot change the allocation of the JVM heap. MaxineVM is a research JVM that is highly modular and configurable and thus a more suitable candidate. Therefore, Section 4.2 provides an overview of MaxineVM.

4.2 MaxineVM - an overview of the architecture

MaxineVM is a Java virtual machine implementation that is compatible with OpenJDK. It is completely compatible with modern Java integrated development environments (IDEs) and the standard Java Development Kit (JDK) and features a modular architecture that permits alternate implementations of subsystems, such as GC and compilation to be plugged in [49].
MaxineVM is written almost entirely in Java, with a small part, called the substrate, written in C. The substrate implements the native launcher for the Maxine VM and encapsulates in a platform-independent API the native services from the Operating System (OS), e.g., virtual memory operation, native thread support, and signal handling. The Java part of MaxineVM is structured around a set of components that collaborate via public interfaces. Each of these interfaces corresponds to a scheme. Schemes formalize the functional interface between high-level abstractions of a VM implementation. The intent is to limit exposure of many low-level implementation-dependent details across these abstractions to ease the replacement of one implementation with another [50]. Below is a description of the main schemes in MaxineVM, as shown in Figure 32:

- The LayoutScheme configures how objects are represented in memory including the header and fields. By default an object is represented in memory as a continuous block with the first two words used by the VM to store data about the object identity (hashcode), locking and Garbage Collection metadata.
- The ReferenceScheme configures how objects are accessed by the mutator threads and how references are encoded.
- The HeapScheme is used to configure how objects and code are allocated and managed during garbage collection. In the current default configuration, the code is allocated in a heap called the code cache while objects are allocated in a separate heap. The objects are collected using a single threaded stop-the-world semi-space collector [51].
- The MonitorScheme is used to implement thread synchronization across MaxineVM as well as to define the wait and notify methods.
- The RunScheme is invoked during VM startup after the basic services have started. The default “Java” run scheme starts up normal JDK services and then loads and runs a user-specified Java class.
- The CompilationBroker handles requests for compilation and adaptive recompilation of methods. MaxineVM features two compilers, T1X and C1X. T1X is the first line of compilation and is a template-based baseline compiler that favors fast compilation over code quality. Frequently executed methods are then rescheduled for compilation using the optimizing compiler C1X.
Figure 32:
The structure of MaxineVM. Highlighted in red are the different schemes that can be implemented in MaxineVM. This figure is taken from [50].

Research in a managed runtime system involves dealing with complex systems and software. Adding or changing a feature often implies modifying code in numerous files and can unveil intricate dependencies that are hard to debug. MaxineVM solves this issue through its modularity and use of a high level programming language, making it a suitable choice for this research.

4.3 Allocating the JVM heap in the Unified Memory address space

In Subsection 3.3, it was shown how using pinned off-heap memory can result in faster data transfers at the cost of explicit allocation through the TornadoAPI and limiting reads and writes to memory through the Foreign Memory Access API. Another limitation of that implementation is that the lifetime of off-heap objects is not managed by the runtime system. In contrast, in this subsection it is shown how minimal changes to the managed runtime
system can enable allocation of the whole JVM heap through the CUDA Unified Memory API, allowing us to directly use Java objects from the heap in PTX kernels, without performing any explicit data transfers.

Normally, MaxineVM reserves virtual memory for the JVM heap from the Operating System (OS) through calls to the POSIX-compliant function `mmap` [52]. During JVM shutdown, the OS system call `munmap` is used to unmap the memory pages from the process. MaxineVM is enhanced and its heap memory allocator is changed to perform calls to the CUDA runtime and allocate Unified Memory. In order to achieve this, the C substrate of MaxineVM is changed, specifically where virtual memory is reserved, the `virtualMemory.c` file. Two native functions called `allocateCUDA` and `deallocateCUDA` that are called from the `SemiSpaceHeapScheme` class are added. The `SemiSpaceHeapScheme` class defines how objects, code and the JVM heap are allocated in the semi-space collector, the default garbage collector of MaxineVM. The `allocateCUDA` function calls the CUDA `cudaMallocManaged` API [53] function and returns an address to the start of the allocated region. The `deallocateCUDA` function calls into the CUDA `cudaFree` API [54] function and returns the status code of the deallocation.

As mentioned in Section 4.2, MaxineVM uses by default a single threaded stop-the-world semi-space heap collector. This means that the heap is partitioned into two different regions and for each region a call to `cudaMallocManaged` is made. Before garbage collection, all the threads that can allocate and mutate state (called mutator threads) are signalled to stop at a safepoint. A safepoint is a place in the code where the interaction of the current execution thread with other components of the VM is at a defined state. When garbage collection occurs, all the reachable objects are moved from one region to the other. After all the reachable objects have been moved, the previous region is marked as empty. If garbage collection occurs too often, the managed runtime will attempt to resize the heap. In the semi-space collector of MaxineVM this is done during a GC cycle, when a region of heap is deallocated through `cudaFree` and a new larger region is allocated through `cudaMallocManaged`. Then, the reachable objects are moved to the new region and the previous one is replaced with a larger one as well.

### 4.4 Description of MaxCudaLib

In order to enable acceleration of CUDA kernels using Java objects allocated in the JVM heap, a thin layer of abstraction called MaxCudaLib was developed. MaxCudaLib is a Java library that exposes a set of functions allowing the user to JIT compile CUDA kernels to PTX and to launch PTX kernels on the accelerator. If used, this library must be imported in the user project as any other dependency. This subsection describes how MaxCudaLib is built.

Since both T1X and C1X only target CPU architectures, MaxineVM currently does not have the ability to JIT compile Java code to PTX. Therefore, in order to exploit the compute capabilities of the GPU, the user must manually develop CUDA or PTX kernels. MaxCudaLib provides a method to JIT compile CUDA C++ kernels to PTX and also caches the PTX code obtained from previous compilations. The signature of the method is the following:
byte[] PTXProvider::getPTX(String methodName, String CUDAKernel)

The method getPTX takes two parameters and returns PTX code as a Java byte array. The first parameter methodName is the cache key and the CUDAKernel parameter is the CUDA kernel represented as a Java String. If the PTX code is not found in the cache, the getPTX method calls into a Java native function that performs JIT compilation of the CUDA kernel through the NVRTC [55] compilation library. NVRTC is a runtime compilation library that accepts CUDA C++ source code as input and creates the handles that can be used to obtain PTX code. After performing the native call, the getPTX method installs the obtained PTX code into the code cache to avoid future unnecessary compilations.

Another method that MaxCudaLib exposes is to launch PTX kernels on the accelerator. The signature of this method is:

```java
void accelerate(byte[] ptx, Object[] inputs, int[] blockDim, int[] gridDim)
```

The method accelerate receives the following arguments:
- A byte array ptx, which is the accelerated kernel and that is returned by the getPTX method if the compilation of CUDA C++ to PTX is used.
- An inputs array whose elements can be Java arrays allocated in the JVM heap.
- An array blockDim specifying the block dimension that is to be used with the PTX kernel.
- An array gridDim specifying the grid dimension that is to be used with the PTX kernel.

The accelerate method calls into a Java native function that performs the calls below:
- JNI API calls to GetPrimitiveArrayCritical and ReleasePrimitiveArrayCritical are used to guarantee that no GC cycle is performed during the native call. In MaxineVM, the whole JVM heap is locked during a critical section.
- A call to cuLaunchKernel is made to run the kernel provided in the ptx byte array.
- A call to cuCtxSynchronize is made after the kernel launch to wait for it to finish.

MaxineVM together with MaxCudaLib facilitate acceleration of CUDA C++ and PTX kernels, allowing objects from the JVM heap to be accessed directly by the GPU without performing any explicit data transfers.

4.5 Experimental performance evaluation

In order to evaluate the performance of a managed runtime system when Unified Memory is used to allocate the heap, a series of benchmarks are performed:
- Monte Carlo - a simulation algorithm performing approximations using statistical sampling, by generating and analysing a large set of random numbers. This technique is useful in risk management, physics simulation and data analysis.
- Black and White - a computation that transforms a picture with RGB color palette to black and white.
- Write Constant - a very simple computation that writes a constant at each index of an input array.

For all of the three benchmarks presented above, a Java function for running on the CPU and an equivalent CUDA C++ kernel hardcoded in a Java String for offloading to the GPU using MaxCudaLib are used. The baseline of each benchmark is sequential CPU execution. In order to analyse how various parts of the managed runtime system affect the speed-up of the acceleration, the following benchmark modes are defined:

- ALLOC - the input of the benchmark is reallocated before each benchmark iteration, using the new keyword in Java. In this mode, each iteration of the benchmark triggers page faults and depending on the data size and the number of iterations, can result in GPU memory oversubscription.
- GC - an explicit garbage collection is triggered through the System.gc() call after each benchmark iteration. In this mode, even though the same input data is used, the garbage collector moves the data in the heap to a different address. Therefore, on garbage collection, the data is moved from the GPU memory to the CPU memory and then on kernel execution, from the CPU memory to the GPU memory.
- NOGC - no explicit garbage collection is triggered and the same input between benchmark iterations is used. In this mode, unless there is an implicit garbage collection triggered by the managed runtime system, the data is only migrated once to the GPU memory. The first iteration of the benchmark causes page faults that migrate the data from the CPU memory to the GPU memory, and subsequent iterations can provide a significant speed up since no page-faults occur.

The experiments are performed on the Laptop configuration described in Table 2. The machine has an Intel i7 @ 4.5 GHz CPU and an Nvidia GeForce GTX 1650 GPU with 3905 MiB of GPU RAM.

Figures 33, 34 and 35 show the speed up obtained from running the Monte Carlo, Black And White and Write Constant benchmarks with different input sizes on the GPU and by accessing Java objects declared in the JVM heap through the CUDA Unified Memory. The common observations regarding the figures are:

- The NOGC mode always performs the best for sizes up to 2048 MB. The reason for obtaining a lower speed up when a size of 4096 MB is used is that the GPU memory is oversubscribed, resulting in page faults on each benchmark iteration.
- The ALLOC mode shows the least performance improvement. The reason that ALLOC is slower than the GC mode is that the GPU memory is oversubscribed with data from the previous benchmark iterations, resulting in memory pages being swapped out from the device RAM to the host RAM before new pages can be migrated. Occasionally, a garbage collection cycle is triggered by the managed runtime system, causing the GPU memory pages to be migrated before the next kernel launch. Therefore, sometimes, an iteration can be as fast as an iteration in the GC benchmark mode.
- The speed up obtained by the GC mode is relatively stable regardless of the data size. After each benchmark iteration, garbage collection is explicitly triggered, causing the memory pages resident on the GPU memory to be migrated to the CPU memory. For sizes up to 2048 MB, this results in page faults on each benchmark iteration, but no memory oversubscription.
Figure 33 shows the speed up obtained from running the Monte Carlo computation with different data sizes. By far, the NOGC mode reaches the best performance for sizes up to 2048 MB, with speed ups ranging between 125x for a size of 256 MB and 134x for 1024 MB. For a size of 1024 MB, the execution time is reduced from 2707 ms when running on the CPU, to 20 ms when running on the GPU. The total number of GPU page faults for a size of 2048 MB is 5910, compared to 117246 GPU page faults when running with a size of 4096 MB. The GC mode is the second best, showing speed ups of 10-11x compared to sequential CPU. The ALLOC mode provides speed ups between 7.9x (128 MB) and 8.6x (1024 MB) for data sizes up to 1024 MB and 11.6x for 2048 MB and 4096 MB.

Monte Carlo - MaxineVM Unified Memory speed-up

Figure 33:
Monte Carlo speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects.

Figure 34 shows the speed up obtained from running the Black And White computation with different data sizes. The NOGC mode reaches the best performance for sizes up to 2048 MB, with speed ups between 29.2x for a size of 128 MB and 29.9x for 1024 MB. The total number of GPU page faults for a size of 2048 MB is 6013, compared to 117338 GPU page faults when running with a size of 4096 MB. The GC mode is second best, showing speed ups of 3.4-3.6x compared to sequential CPU. The ALLOC mode provides speed ups of 2.5x for data sizes up to 1024 MB and 3.5x for 2048 MB and 4096 MB.
Figure 34:
Black and White speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects.

Figure 35 shows the speed up obtained from running the Write Constant computation with different data sizes. The **NOGC** mode reaches the best performance for sizes up to 2048 MB, with speed ups between 12.5x for a size of 128 MB and 14.27x for 512 MB. The slowdown with a size of 4096 MB is 0.19x. The total number of GPU page faults for a size of 2048 MB is 5910, compared to 117264 GPU page faults when running with a size of 4096 MB. The **GC** mode is the second best, showing speed ups of 1.06-1.1x compared to sequential CPU. The **ALLOC** mode provides slowdowns of 0.8x for data sizes up to 2048 MB and a speed up of 1.12x for 4096 MB.
Figure 35:
Write Constant speed up obtained from running a CUDA C++ kernel instead of the sequential CPU execution and directly using JVM heap objects.

Figure 36, depicts the slowdown of Garbage Collection when Java heap objects are passed through Unified Memory to CUDA C++ kernels. The GC slowdown is measured in the GC benchmark mode, through an explicit call to System.gc() after each benchmark iteration. During GC, all the memory pages resident in the GPU memory are accessed by the CPU, and therefore migrated to the host memory. As expected, the slowdown increases with the input data size (with the number of pages in the GPU memory). The smallest slowdown is 1.6x for an input of 128 MB, while for an input size of 4096 MB, the slowdown is 2.45x.
Figure 36:
Garbage Collection slowdown obtained from accelerating the Write Constant, Black and White and Monte Carlo benchmarks on the device in the GC benchmark mode. Since the benchmark sizes are the same, the same slowdown is observed for all three benchmarks. Therefore, a single figure is shown. The lower the better.
5 Related work

In this chapter, an analysis is made of previous research about data transfer optimizations in the context of a heterogeneous managed runtime system. Over the years, there have been multiple frameworks developed to address the problem of heterogeneous computing using a high level programming language. However, only some solutions look further into optimizing data transfers and memory management when a managed runtime system is used.

Table 4 is used to describe the key differences between TornadoVM, MaxineVM, and previous research. The first two rows show the current versions of TornadoVM and MaxineVM and their status related to the research topics. Then, the TornadoVM + Thesis and MaxineVM + Thesis rows represent the extended versions with the features presented in the current thesis. The remaining paragraphs in this chapter expand the differences between TornadoVM + Thesis, MaxineVM + Thesis, and the related work.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Managed Runtime</th>
<th>OpenCL support</th>
<th>CUDA support</th>
<th>Pinned memory</th>
<th>Batch processing</th>
<th>CUDA Unified memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>TornadoVM v0.12</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>MaxineVM v2.9.0</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TornadoVM + Thesis</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>MaxineVM + Thesis</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Bastem et al.</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dubach et al.</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JOCL</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JCUDA</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Aparapi</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IBM J9</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4: The differences between TornadoVM, MaxineVM and related work.

Bastem et al. [36] expose a programming model that is low level and works only with C++, using OpenACC. It decomposes the data and computation into tiles and treats them as the main data transfer and execution units. Additionally, it solves the issue when the application data does not fit into the device memory. In the analysis, CUDA pinned memory, non-pinned
memory and Unified memory are used. However, this work does not present an analysis of data transfers in OpenCL. In addition to this, TornadoVM, a framework that is built on top of a managed runtime system, was expanded. CUDA Unified Memory is also used, but in the context of a managed runtime system, MaxineVM.

Dubach et al. [56] present a runtime and language implementation, called Lime, based on Java that allows offloading of tasks to execute on GPUs and FPGAs. The Lime compiler statically generates JVM bytecodes and OpenCL kernels that are later called at runtime. In order to perform data transfers, the Lime runtime serializes Lime values to byte arrays, passes them to JNI and then deserializes them to C-style values that can be later accessed by the OpenCL kernels. Even though it is specified as a potential improvement, the presented work does not support memory pinning or batch processing.

JOCL [57] and JCUDA [58] are OpenCL or CUDA wrappers for Java. Using these frameworks, programmers must implement their kernels in OpenCL or CUDA. Therefore, it requires knowledge about the programming model and hardware. Both frameworks allow the declaration of page-locked pinned buffers but lack any batch processing functionality, relying on the user to implement it in code. Additionally, it is possible to allocate CUDA Unified Memory, but not as part of the JVM heap.

Aparapi [6] transparently performs data transfers between the host and the device but does not have support for off-heap buffers or pinned memory. In contrast, TornadoVM was enhanced to support such features. Additionally, MaxineVM with the added extensions does not use a framework to perform data transfers, since the Unified Memory page faulting mechanism is transparent to the JVM.

IBM J9 [4] is a JVM that contains a JIT compiler for Java to PTX. It is able to generate GPU code from a Java program written using the Java 8 Parallel Streams API. Additionally, it optimizes redundant transfers and automatically generates the required API calls to the CUDA driver. However, it does not use pinned memory, unified memory or support data batching.
6 Conclusion and future work

In this thesis, a comprehensive analysis and optimizations of the performance of data transfers in heterogeneous managed runtime systems was conducted. The CUDA and OpenCL programming models and a high-level overview of the hardware abstraction were presented in the context of the current status of heterogeneous computing. The challenges that a managed runtime faces in the context of heterogeneous computing were described: the inability to perform asynchronous data transfers with objects residing in the JVM heap, the lack of an API to declare pinned memory and the imposed limitations from using off-heap buffers.

Then two different solutions were proposed:

- TornadoVM was enhanced to enable allocation of pinned off-heap buffers using Project Panama and parallel batch processing was implemented as an optimization. Future work on this solution can include a hybrid approach when irregular sizes are used for batch processing. As presented in Section 3.4 the TornadoVM bytecode interpreter is locked when the final kernel of an irregular sized batch must be recompiled. However, this could be avoided by improving TornadoVM to perform precompilation of all the required kernels before starting to interpret the bytecodes.

- The heap allocation of MaxineVM was improved by utilizing the CUDA Unified Memory, therefore enabling the device to directly access Java objects allocated in the JVM heap. Further improvements can include implementing a Java to PTX JIT compiler in MaxineVM and using it as a final compilation tier. This would remove the necessity of using the MaxCudaLib and would completely be transparent to the programmer. Currently, code compiled with C1X does not record any runtime profiling information such as method invocation counts and therefore the existing optimizing compiler would need changes too. Another improvement would be to transparently trigger prefetching of memory pages to the GPU when compute offloading is detected.

With the proposed solutions, significant data transfer and overall performance improvements in heterogeneous managed runtime systems were obtained.
Bibliography


Nvidia Runtime API official documentation - cudaMallocManaged - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1qd228014f19cc0975eb3e0dd2af6dd1b - Accessed on 06/09/2021

Nvidia Runtime API official documentation - cudaFree - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1qa042655cbb3408f010661652a075e094 - Accessed on 06/09/2021


Design and Implementation (PLDI '12). Association for Computing Machinery, New York, NY, USA, 1–12. DOI:https://doi.org/10.1145/2254064.2254066