Unified Shared Memory: Friend or Foe? Understanding the Implications of Unified Memory on Managed Heaps

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

Florin Blanaru*
florin.blanaru@axelera.ai
Axelera AI
Netherlands

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

Steve Dohrmann
steve.dohrmann@intel.com
Intel
United States

Sandhya Viswanathan
sandhya.viswanathan@intel.com
Intel
United States

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

Abstract

Adopting heterogeneous execution on GPUs and FPGAs in managed runtime systems, such as Java, is a challenging task due to the complexities of the underlying virtual machine. The majority of the current work has been focusing on compiler toolchains to solve the challenge of transparent just-in-time compilation of different code segments onto the accelerators. However, apart from providing automatic code generation, another paramount challenge is the seamless interoperability between the host memory manager and the Garbage Collector (GC). Currently, heterogeneous programming models that run on top of managed runtime systems, such as Aparapi and TornadoVM, need to block the GC when running native code (e.g, JNI code) in order to prevent the GC from moving data while the native code is still running on the hardware accelerator.

To tackle the inefficacy of locking the GC while the GPU operates, this paper proposes a novel Unified Memory (UM) memory allocator for heterogeneous programming frameworks for managed runtime systems. In this paper, we show how, by providing small changes to a Java runtime system, automatic memory management can be enhanced to perform object reclamation not only on the host, but also on the device. This is done by allocating the Java Virtual Machine's object heap in unified memory which is visible to all hardware accelerators. In this manner -although explicit data synchronization between the host and the device is still required to ensure data consistency- we enable transparent page migration of Java heap-allocated objects between the host and the accelerator, since our UM system is aware of pointers and object migration due to GC collections. This technique has been implemented in the context of MaxineVM, an open source research VM for Java written in Java. We evaluated our approach on a discrete and an integrated GPU, showcasing under which conditions UM can benefit execution across different benchmarks and configurations. We concluded that when hardware acceleration is not employed, UM does not pose significant overheads unless memory intensive workloads are encountered which can exhibit up to 12% (worst case) and 2% (average) slowdowns. In addition, if hardware acceleration is used, UM can achieve up to 9.3x speedup compared to the non-UM baseline implementation for integrated GPUs.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**.

Keywords: CUDA, Data Transfers, GPUs, JVM, LevelZero, Memory Management, Unified Memory, Virtual Machines

ACM Reference Format:
Juan Fumero, Florin Blanaru, Athanasios Stratikopoulos, Steve Dohrmann, Sandhya Viswanathan, and Christos Kotselidis. 2023. Unified Shared Memory: Friend or Foe? Understanding the Implications of Unified Memory on Managed Heaps. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617651.3622984

*Work done while he was associated at The University of Manchester.

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622984

1 Introduction

Heterogeneous hardware has become ubiquitous and is present in almost every modern computing system. However, supporting fully automatic heterogeneous hardware execution from managed languages still has several unresolved

challenges due to the following reasons: i) parallelism identification: how to represent and identify parallel programs that can run on many types of hardware accelerators such as GPUs and FPGAs; ii) runtime compilation: how to compile high-level programs to accelerator-compatible bare-metal code while achieving high-performance; iii) data management: how to efficiently manage memory between the Virtual Machine (VM) and the accelerator's memory. The latter involves avoiding data marshaling and unmarshaling between host memory buffers (e.g., Java heap objects) and device memory buffers (e.g., data buffers on the accelerator's memory). In addition, the Garbage Collector (GC) may collect and move buffers and objects from different memory regions while the accelerator is executing the parallel code resulting in segmentation faults.

The first two challenges have been well-researched during the last decade with a plethora of programming models [14], Application Programming Interfaces (APIs) and frameworks [7, 15, 18, 20, 21, 32, 35, 48, 54, 55] having been introduced for various programming languages. Amongst said solutions, different levels of integrations exist ranging from Just-In-Time (JIT) compilation of high level programming languages [17, 19, 21, 28, 56] to heterogeneous code via external invocations of pre-built binaries [8, 25, 43].

Data management is a more challenging topic because, regardless of the quality of a JIT compiler, if the memory manager is not optimized, it is very difficult to achieve high end-to-end performance by using hardware accelerators; even for applications perfectly suitable for acceleration. Although there are some works focused on improving data management [18], the question of *how the memory allocator and GC of a VM can interact with the accelerators' memories* is hardly researched. For example, even state-of-the-art parallel programming frameworks for Java, such as Aparapi [17] and TornadoVM [11, 19], might fail if the GC collects objects while the parallel application is running on the accelerator.

This paper focuses on the memory management of managed runtime systems and heterogeneous programming models by taking advantage of the **Unified Memory (UM)** of modern computing systems. Note that NVIDIA uses the term Unified Memory while Intel uses the term Unified Shared Memory. For consistency, in this paper we use the term Unified Memory (UM) for both Intel and NVIDIA systems.

We propose a technique to allocate the Java heap on the UM of integrated and discrete GPUs. With the proposed scheme, when GPU buffers are allocated in UM, the driver can automatically migrate memory pages from the host (CPU) to the device memory and vice-versa under demand. This enables the seamless interoperability of the hardware accelerator with the VM runtime and its GCs. The outcome of this research is to provide valid execution in which the managed runtime systems can safely execute programs on hardware accelerators without any memory corruption when the GC collects and moves memory segments.

By enabling seamless interoperability during GC cycles, the only remaining task that developers must do is to ensure data consistency between the host and the accelerator. We propose to perform this sync point during safepoint polls before Stop-The-World (STW) GC cycles. We prototyped our solution in MaxineVM [30, 52], a research VM for Java written in Java. Regarding parallel programming models for GPUs, we experimented with NVIDIA CUDA [50], and Intel Level Zero [26], which allow running programs on both discrete and integrated GPUs. Despite the selected platforms, all techniques presented in this paper can be implemented for other VMs, such as the JVM and .NET.

To summarize, the contributions of this paper are:

- It proposes a novel technique that allows managed runtime systems to allocate Java objects on a shared memory space between the CPU and the accelerators. Due to automatic page migration, we demonstrate that this technique can work in combination with common GCs.
- It presents an implementation prototyped in the context of MaxineVM that has been extended to support UM for discrete and integrated GPUs with two different parallel programming models (CUDA and Level Zero).
- It studies the effects of Unified Memory on managed workloads, even if hardware acceleration is not invoked revealing a potential overhead of up to 12% in the worst case.
- It performs a detailed performance evaluation of the whole system across different GPU workloads and configurations. We show that our UM system can achieve speedup of up to 9.3x on integrated GPUs compared to the non-UM baseline implementation.

2 Background

This section provides the background on MaxineVM, CUDA, Level Zero and Unified Memory.

2.1 MaxineVM Overview

Our work is prototyped as an extension to MaxineVM [31, 52]. MaxineVM is a virtual machine implementation for Java that is compatible with the standard Java Development Kit (JDK). It features a modular architecture that permits alternative implementations of subsystems, such as GC and compilation to be plugged in.

Since MaxineVM is mostly implemented in Java, it facilitates prototyping of new ideas and research directions for VMs and new hardware. The substrate implements the native launcher and encapsulates, in a platform-independent API, the native services from the Operating System, e.g., virtual memory operation, and signal handling.

Memory Handling within MaxineVM. The Java components of MaxineVM are architected around a set of components that collaborate via public interfaces that correspond to *schemes*. One particular scheme relevant to this work is the *Heap-Scheme*, that configures how objects (data) and code

are allocated and managed during GC. In the current default configuration of MaxineVM, application code is allocated in a heap called the code cache while application objects are allocated in a separate heap. MaxineVM supports different GC implementations with varying degrees of maturity. In the context of this work, we use the single threaded stop-the-world semi-space collector [29], since we are interested in the worst case scenario and overheads that UM may result. In the *SemiSpace* heap scheme, upon VM startup, two continuous regions of memory that constitute the object heap are allocated. At runtime, the mutator threads allocate objects in a single region of memory at a time. Upon GC, all mutator threads are stopped, and a single thread (the GC thread) traverses the object graph and marks the root objects. Next, the live objects are copied to the other memory region following Cheney's algorithm [9]. After all objects are copied, the previously used memory region is marked as empty and all new allocations will be performed in the region where the objects have been copied to. Consequently, the mutator threads resume execution.

2.2 CUDA and GPU Unified Memory

CUDA [50] is a parallel programming framework and an application programming interface (API) for parallel programming on NVIDIA GPUs. CUDA allows applications written in C/C++ to accelerate certain types of workloads by exploiting GPU resources, achieving high-performance compared to multi-threaded CPU execution.

GPU Programming Model. In a nutshell, the way GPU applications are executed with CUDA is as follows: developers need to allocate the data on the CPU's and the GPU's memories, perform a data transfer from the host (CPU) to the device (GPU), launch the compute-kernels on the GPU, wait for the kernels to finish, and copy the results back from the GPU's main memory to the CPU's memory. Prior to the introduction of unified memory, developers needed to manually handle data copying and consistency between the host and the devices. In the Java world, however, some systems such as Aparapi [17], TornadoVM [12], and IBM J9[28] could assist developers in performing these actions in a transparent manner. Nevertheless, developers today can utilize unified memory or not, or even have a combination of the two techniques.

Unified Memory. CUDA UM is a feature that allows developers to have a single memory address space view between CPUs and GPUs [57]. To achieve that, data migration is automatically handled by the GPU driver which migrates memory pages between the host and device under demand. This feature enhances GPU programmability because there are no explicit copies between the different memory systems, as it facilitates management of memory by providing a single and consistent view of the host and device memories. UM became available from CUDA version 6, and it was fully implemented

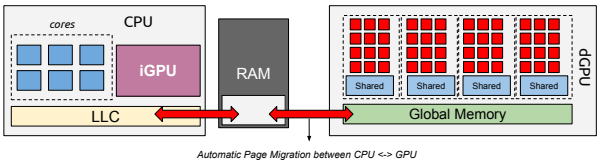

Figure 1. High-level representation of a Unified Memory System for integrated and discrete GPUs.

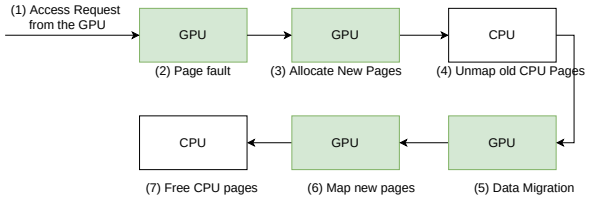

Figure 2. Page-fault Mechanism in CUDA Unified Memory.

in the NVIDIA Pascal micro-architecture [37]. Please note that unified memory is different from zero-copy [49].

Figure 1 shows a high-level representation of CPU and GPU hardware architectures, and how UM can be transparently managed by the driver via memory page migration. As shown, the CPU has a set of compute-cores, an integrated GPU, and a Last Level Cache (LLC) shared between them. Applications run on the CPU and both code and data are loaded onto the main RAM memory. Since the unified memory is also accessible from discrete GPUs, such as NVIDIA and Intel integrated GPUs, pointers can be also shared between the main host and the GPU memory, as we can see in the right-hand side of Figure 1.

GPU memory page migration. Figure 2 shows the steps of handling a page fault when the GPU attempts to access a page that resides on a CPU. When a memory page that is not mapped to a physical address on the current device (CPU or GPU) is accessed, a page fault is triggered. In step 1, the GPU tries to access a memory position that has not been yet mapped to a physical memory address. Thus, a page fault is generated (step 2). The page fault provokes the GPU driver to allocate new physical pages for the requested memory access (step 3). Then, the CPU must unmap the page associated with the memory request (step 4) and perform data migration from the CPU's memory to the GPU's memory (step 5). Once the data transfer has been completed, the GPU driver maps the new pages to the GPU's memory (step 6). Finally, the GPU driver frees the memory page at the CPU side (step 7).

Resolving a page fault can have high overhead, and therefore, decrease significantly the system performance when the same memory pages are accessed by the CPU and GPU [33]. In addition, the massive parallelism of GPUs further exacerbates the page fault overhead. This occurs because processing stalls while page faults are being resolved, and multiple threads in different warps accessing the same page can cause multiple duplicated faults [49].

2.3 Intel Level Zero and Unified Memory

Level Zero [26] is an API initiated by Intel as part of the Intel oneAPI [27] software ecosystem for CPU, GPU, and FPGA parallel programming. In contrast to CUDA, the oneAPI programming model implements the SYCL standard for C++. This allows programmers to write standard C++ programs (without any extensions) in the same source files that the main C++ programs are located. Level Zero fits into the oneAPI stack as a driver connector (or resource manager) between the software runtime and the actual hardware, and it can be used as a standalone API for other applications.

Similarly to CUDA, Level Zero offers a Unified Memory API for memory types with which programmers can allocate buffers in a common memory region between the CPUs and the GPUs. Since Level Zero is a more generic programming framework for accelerator resources, it works in different ways depending on the type of the accelerator. For shared memory devices (e.g., an Intel integrated GPU that shares memory with the main CPU), the advantage of using UM is that there is no memory copy (truly zero-copy) between the host and the device. For discrete accelerators (those that have their own memory pool, such as discrete GPUs) the Level Zero driver automatically migrates the memory pages between the CPU and the GPU and vice-versa when they are requested, simplifying memory management. This behaviour is almost identical to the CUDA UM for discrete NVIDIA GPUs that we described in Section 2.2.

Research Gap. Our work makes use of the CUDA UM and Level Zero UM to allocate a shared memory Java heap with the goal of having shared pointers that can be migrated under demand by the GPU driver. During a GC cycle, GPU pages can be automatically reclaimed by the CPU. In turn, the CPU performs the GC and continues execution without having to lock the GC while the application is running. It is important to note, however, that although the GC can work in tandem with the processing on the GPU, the shared pointers between the CPU and the GPU can creates race conditions. It is up to the developer to ensure memory consistency by adding sync points that block the GC during the syncing phase. In this work, we integrate a GPU synchronization point before performing a GC within the VM. Thus, our solution is fully transparent to programmers. In the rest of the paper we showcase our approach and we analyze the performance on discrete and integrated GPUs.

3 UM for Managed Runtime Systems

Our technique has been developed for two implementations and setups: one for discrete NVIDIA GPUs and one for Intel integrated GPUs. The implementation for the NVIDIA GPUs utilizes the CUDA driver API [13], while the implementation for the integrated GPUs uses the Level Zero API [26]. The technique used to accommodate MaxineVM for Level Zero UM represents a more generic approach which can be applicable to other parallel programming models, such as SYCL [24] and OpenCL [23]. The reason is that all these programming models operate in a similar manner, and there is a number of data structures (e.g., GPU command queues) that can be mapped to all of them. Thus, the potential accommodation of multiple heterogeneous programming models (and not only CUDA), makes the VM design suitable for targeting not only NVIDIA GPUs but also any type of hardware accelerators from various vendors.

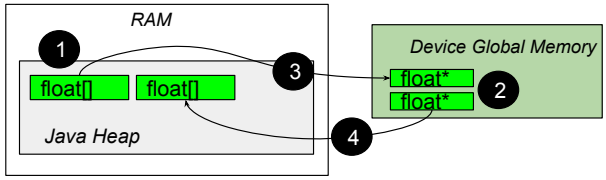

Figure 3. Example of a program workflow of CUDA/LevelZero interaction with the Java memory management.

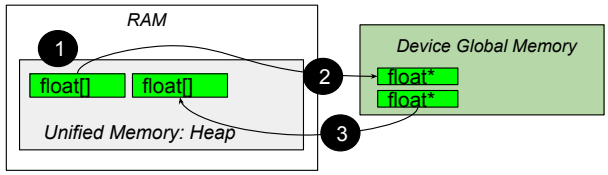

Figure 4. Program workflow using UM as Java heap space.

3.1 System Overview

Figure 3 shows an example of a typical program workflow in Java that uses a GPU as an accelerator (without using UM). The example uses two `float` arrays as input/output data. The figure also highlights the main steps that involve the buffer allocation and data transfers between the Java heap and the device heap when UM is not used.

The usual workflow is as follows: in step ❶, programmers allocate the data using the Java constructors (`new`). This data is kept on the Java heap that resides in the CPU's main memory. Then, developers need to allocate the device buffers (step ❷) for each buffer to be used on the target GPU. Once the buffers have been allocated, developers need to perform a data transfer from the Java heap to the device (step ❸), launch the kernel functions, and finally obtain the results by moving the data from the GPU's global memory to the Java heap (step ❹). Note that, only when the data transfer has finished, the GPU kernel can be launched.

MaxineVM reserves virtual memory for the JVM heap by invoking the Operating System (OS) calls to the POSIX-compliant function *mmap*. In our approach, instead of calling the *mmap* function, we allocate the Java heap using the corresponding API for Unified Memory (`cudaMallocManaged`, in the case of the CUDA driver API, and `zeMemAllocShared` in the case of the Level Zero API). This allows user applications to directly use the Java arrays and buffers without the need of performing data marshaling, unmarshaling, and explicit data transfers.

Figure 4 shows a representation of a common workflow using our approach. Data is allocated using the usual Java

constructors. Since the Java heap has been allocated using the Unified Memory functions, all pointers to that data are also accessible from the device (step ❶ from Figure 4). Then, developers can directly invoke the compute-kernels on the accelerators without the need of marshaling, or performing any data transfers between the Java heap and the device's global memory. The corresponding GPU/Accelerator drivers will take care of memory page migration during runtime (steps ❷ and ❸) under demand. Furthermore, only the data that is needed will be transferred while the GPU kernel can still run.

System Components Overview. Figure 5 shows how applications interact with the VM to gain native access to the unified memory space. The JVM exposes a set of accelerator interfaces (what we call *XPU Interfaces*) to programmers. These interfaces contain the minimum set of parameters for resource instantiation that programmers need to define for using the UM. In general, depending on the parallel programming model used to access GPUs and heterogeneous hardware, allocations of memory regions in the unified memory may require access to other resources, such as a driver, device and command queue objects.

Furthermore, although the Java heap is allocated using UM with CUDA or Level Zero, programmers do not necessarily have to run on heterogeneous hardware. Thus, it is possible to use UM space and still run typical workloads on CPUs, as we will show in Section 4.

3.2 Enabling Unified Memory within MaxineVM

We extended the MaxineVM implementation to perform calls to the CUDA runtime and allocate UM when the Java heap is allocated. To achieve that, the C substrate of MaxineVM has been extended. Our extensions include two new native functions (`allocateCUDA` and `deallocateCUDA`) which are called from the *Semi Space Heap* scheme within MaxineVM.

The `allocateCUDA` function internally invokes the *cudaMallocManaged* function from the CUDA Driver API, and it returns an address that points to the start of the allocated unified memory region. Similarly, the `deallocateCUDA` function invokes the CUDA `cudaFree` API function and returns the status code of the deallocation process.

To enable UM in Intel Integrated Graphics, we took a similar approach by calling the shared memory buffer allocation functions `zeMemAllocShared` and `zeMemFree` respectively. The main difference is that, for Intel Integrated GPUs, the shared memory resides in the same space as the host memory (main CPU's memory), and the memory page migration is not actually performed.

3.3 XPU Interface

Apart from the buffer pointers (either CUDA or Level Zero UM pointers), the VM also needs to keep some GPU objects, such as a *context-id*, *device-id* and *driver-id* objects. In our implementation, we also provide a synchronization point over the GPU command stream (a GPU object that is used to submit commands, such as kernel launch and synchronization points) before a GC is invoked. This is because, while UM memory simplifies the GPU execution, it does not prevent the GC to run while the GPU application is still running. UM memory allows this to happen without the VM to raise a segmentation fault.

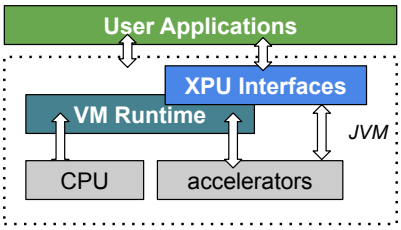

Figure 5. System overview of how applications can use heterogeneous resources with our approach.

However, as any other shared resource in concurrent and parallel programming, the VM needs to flush the pending commands from the GPUs's command queue to obtain the correct behavior. In our implementation, we provide this functionality automatically, by enabling the VM to access the execution context and the GPU command queues. Thus, a new data structure is shared between the VM and the application code called *XPU Interface*. With this approach, user applications directly share and query GPU resources from the VM using the *XPU Interface*, and launch GPU kernels using a native interface (e.g., JCUDA).

We provide, as a proof-of-concept, our own native GPU code dispatcher in JNI. When the JNI code for launching the kernels sets the parameters for the CUDA or Level Zero kernels, it directly passes pointers that are obtained from the JNI call (without any data transformation or previous memory allocation). This is only possible because the heap is allocated using UM. As follows, we will define the *XPU Interface* data structure, and how programs interacts with it.

Interacting with the XPU Interface. The design of the *XPU Interface* represents a generic approach to use UM between CPUs and GPUs, and it accommodates both CUDA and Level Zero heterogeneous programming models. For one side, the CUDA implementation only needs to share a context object (a GPU context is an object that handles the execution on heterogeneous devices, which is linked to a device and a driver implementation and it is needed to perform, for instance, synchronization operations).

On the other side, synchronization in Level Zero is performed through two objects, called command queue and command list. Instead, the context in Level Zero is used by client applications to build SPIR-V modules[1]. Thus, the Level Zero implementation represents a more generic approach, and similar to other parallel programming models such as OpenCL [23], AMD HIP [3], and Intel oneAPI [27].

[1]SPIR-V is a standard common binary intermediate representation for compute-kernels and graphics for hardware accelerators.

Listing 1. XPU Java interface provided by the VM.

```
1 public interface XPU_Interface {
2     long getDriver(int heapSpaceIndex);
3     long getDevice(int heapSpaceIndex);
4     long getContext(int heapSpaceIndex);
5     long getCmdQueue(int heapSpaceIndex);
6     long getCmdList(int heapSpaceIndex);
7 }
```

Listing 2. XPU Implementation by the VM.

```
1 public class VM_XPUImpl implements XPU_Interface {
2     public static VM_XPUImpl getInstance() {...}
3     public Pointer getDriver(int index) {...}
4     public Pointer getDevice(int index) {...}
5     public Pointer getContext(int index) {...}
6     public Pointer getCmdQueue(int index) {...}
7     public Pointer getCmdList(int index) {...}
8 }
```

During the VM's bootstrap, the VM allocates, and initializes the Java heap, and it prepares the VM to start running applications on GPUs. At this time, MaxineVM obtains the driver, device, context and command queue/list objects the CUDA and Level Zero APIs. Consequently, MaxineVM stores all of these pointers in a class that will be exposed to user applications.

Listing 1 shows the *XPU* Java interface that MaxineVM exposes to the user. This interface represents the basic functionality to be used by applications. Through this interface, applications that aim to run on GPUs with UM can obtain the driver object (native pointer), the device object, and the context that are associated to a particular heap space. Since MaxineVM uses a single memory region for the heap, we only use one space in our implementation. However, this approach can be extended to use multiple Java heaps and select the correct space by using a heap index. Besides, each heap index can be associated to a different accelerator (e.g., a different GPU). The VM exposes an implementation of this interface through a singleton class, as shown in Listing 2.

Using the XPU Interface. The communication with the *XPU Inteface* is done through the singleton class exposed by the VM. Listing 3 shows a sketch of a user code example implemented in Java that invokes the `XPUImpl` singleton class. The application imports the new utility from the core set of Java APIs. Line 3 invokes the XPU singleton interface to get the `xpu` reference from the VM. Then, in line 6-8 we invoke the `getContext` to obtain the raw pointer of the context, device and command list associated with the first heap space. From this point of the execution, the user application can launch kernel on the GPU, add sychronization points, etc.

Improving hardware resources through the XPU Interface. Executing on heterogeneous hardware through the *XPU interface* enables the VM to control hardware resources. By invoking this interface, the VM can track all compute

Listing 3. Example of XPUImpl usage by applications.

```
1  public import jdk.VM_XPUImpl;
2  public class Sample {
3    VM_XPUImpl xpu = VM_XPUImpl.getInstance();
4    void doSomethingOnGPU() {
5      // Query GPU shared objects
6      Pointer context = xpu.getContext(0);
7      Pointer device = xpu.getDevice(0);
8      Pointer cmdList = xpu.getCmdList(0);
9      ...
10 }}
```

kernels that want to access GPUs for acceleration and hence, it can implement resource management to improve the performance of the whole system.

But, why is the XPU-Interface needed? It is important to understand why the *XPU Interface* is needed when shared memory is used. For the majority of heterogeneous programming models, in order to invoke kernels, perform synchronization points, and attach events, a device pointer, command queue and command lists are needed. From one side, the VM process also needs these objects to guarantee consistency and flush command queues, for example, before invoking the GC. From the other side, applications also need these pointers to communicate and run kernels on the target accelerator.

To expand on this, Figure 6 shows an example in the context of Level Zero about how applications are executed on a GPU when using UM to allocate the Java heap. The figure is divided into two parts: the top part represents the compilation workflow from OpenCL to SPIR-V. This step is performed ahead-of-time by the user. As an example, for this step, developers can use `clang` [44]/LLVM [34] compiler and its `llvm-spirv` utility to transform the OpenCL C into a SPIR-V binary modules.

The bottom part of the figure shows the execution workflow to run on GPUs using *XPU interface* with Level Zero. The user application must use the same command list and queue objects that were associated with a device from the VM side at bootstrap. Thus, developers can orchestrate the entire Level Zero application needed to launch GPU kernels.

GC interactions. As already mentioned, although UM allows the GC to operate while the GPU is processing heap data, race conditions on the buffers shared between the host and the device may occur. Hence, in order to guarantee memory consistency developers must sync the data between the CPU and the GPU, after the GPU finishes processing. This can be achieved either by an explicit call to the CUDA function `cuCtxSynchronize` or the Level Zero function `zeCommandQueueSynchronize`.

We opted for a different approach by triggering these calls upon safepoint polls. Every time a GC is being invoked and the safepoints are being polled, a call to the CUDA context

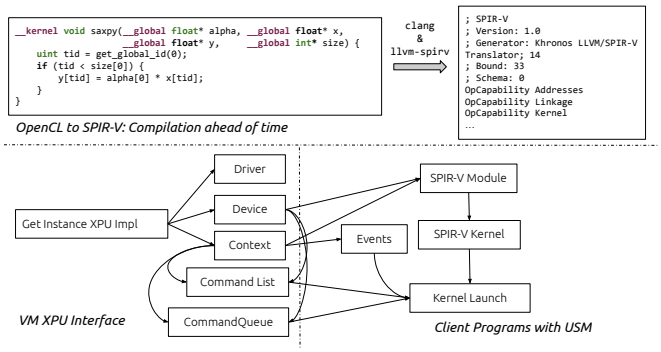

Figure 6. Execution Workflow to run SPIR-V applications in the context of Level Zero on Intel HD Graphics with MaxineVM.

synchronize or Level Zero command queue synchronization functions is performed to synchronize the shared memory buffers between the GPU and the CPU. As a side note, the Level Zero synchronization point also needs to close the command lists before the synchronization point, and it resets the list afterwards. This approach allows to have a full consistent view of the UM memory between the Java heap and the accelerators, without relying on the user to add these synchronization points.

Note that our work does not aim to improve GC performance. Instead, this paper shows the potential of using UM as a Java heap to reduce redundant copies of data between Java and JNI for offloading computations on the GPUs (using shared pointers between Java memory objects and the GPU memory). The employment of UM enables GC to be triggered while the GPU runs applications without provoking segmentation faults. The sync point is needed to guarantee consistency since the GPU is a shared resource (similar to other concurrent programming models). Thus, our work solves these two issues with: a) GC-aware command queues for internally blocking GC tasks while the GPU runs kernels using the same memory buffers without segmentation faults; b) simplification of native code that dispatches the GPU code, because pointers are shared.

3.4 Case scenarios

Our paper aims to perform a worst case performance analysis in order to expose any significant drawbacks that UM might have on typical execution scenarios. To that end, we devised three different scenarios that aim to represent how users might typically use hardware acceleration within a managed runtime environment. Table 1 shows the three scenarios that we used to evaluate our approach in Section 4.

Allocate Always. In this scenario, data is always being newly allocated before every kernel launch. This scenario aims to represent hardware acceleration in a streaming fashion where there is no data reuse between different kernel launches. In this case, since we create new data before launching a kernel, we guarantee that the GPU driver, on discrete GPUs, will always perform memory page migration from the host to the device. In the case of integrated GPUs, there is no page migration. Furthermore, when the application is executed multiple times on the GPU, since new data is being allocated, it can trigger a GC on the host side. In addition, the application can result to oversubscription [51] on the device side, which provokes memory pages to migrate back and forth between the host and the device.

Table 1. Different scenarios using UM as a Java heap.

Name mode	Scenario	Behavior
AA	Allocate Always	Data is newly allocated every time on the CPU. Thus, page faults are generated every time the GPU kernel runs.
AOGC	Allocate once with GC	Data is allocated once, but an explicit GC is invoked every time a GPU kernel is launched. Thus, pages will be migrated back and forth between the CPU and the GPU.
AO-NOGC	Allocate scenarios once with no GC	Data is allocated once and there is no GC. Thus, data is not reclaimed at the host side and there is only one data transfer from the host to the device.

Allocate Once with GC. In this scenario, data is allocated only once during the first kernel invocation. After that, data is being reused between kernel launches. In addition, we force a GC at the host side between kernel launches. Since we use a *SemiSpace* collector, this will result to a full heap GC, which will provoke page faults if the memory pages are not present on the CPU. With this scenario we aim to understand the implications of a full GC on UM, specifically when the CPU and GPU share pre-existing data (in contrast to the *Allocate Always* where we always allocate new data before running on the GPU). Note that, in this case, the GPU can still be running the compute-kernels, while a GC is being triggered. Since the memory is a shared resource, to guarantee correct results, the VM must perform a synchronization point to wait for the GPU to finish the kernel before performing the full GC.

Allocate Once without a GC. This scenario is similar to the previous one with the omission of GC. Again, we allocate data only once (before the first kernel launch), and then we reuse it across different subsequent operations. In addition, since we guarantee no GC to occur in-between, we maximize data reuse and attempt to minimize page faults. This scenario represents the *best case* where there should be few page migrations triggered. In a general GPU workflow, this case is quite unlikely to happen since the results computed by the GPU will probably be consumed by the host.

4 Experimental Evaluation

This section presents the evaluation of the proposed technique of exploiting UM in Java. We first describe the setup and benchmarks, and then we discuss the performance results with a detailed analysis.

Table 2. Hardware/Software testbed characteristics.

CPU	Intel Core i7-12700K
Hyper-Threading	Disabled
Main Memory	32 GB
NVIDIA GPU	NVIDIA GeForce GTX 3070
NVIDIA GPU RAM	8 GB
Intel Graphics/RAM	Intel(R) UHD Graphics 770
PCIe	Gen 4 (16 lanes enabled)
JVM	MaxineVM
JVM Heap Size	25 GB
OS	openSUSE Leap 15.4 - 5.14.21-150400
CUDA Driver	515.76
Level-Zero Driver	22.23.23405

Table 3. Applications and data sizes used.

Benchmark	Small (MB)	Large (MB)
Saxpy, WC, BW, HB, MC, BS	128	4096
DFT	0.01	0.5
NBODY	0.05	1

4.1 Experimental Setup

We selected a set of kernels commonly used for GPU acceleration from different domains, such as linear algebra, physics simulation, and Fintech. The benchmarks were selected based on three characteristics: a) a group of low-compute and high-bandwidth to exercise the worst case scenarios in which a lot of data must be transferred to the GPU to compute a few operations per thread; namely, *saxpy*, *writeConstant* (WC); b) a group of benchmarks in which data transfers and compute are more balanced, namely *black and white* (BW) which transforms an RGB image to monochrome, *montecarlo* (MC) simulation, *Hilbert* (HB) computation, and *black-scholes* (BS); c) a group of benchmarks for compute-bound [42] and low communication between the CPU and GPU, namely *Discrete Fourier Transform* (DFT), and *N-Body* (NB) simulations. The compute-bound kernels were ported from C++ using CUDA and OpenCL from the AMD SDK [2] and NVIDIA CUDA SDK [38].

Methodology. Table 2 presents the specifications of our testbed along with the configuration of the JVM that we used to run all benchmarks. For each experiment, we performed a warm-up phase that includes 10 executions. After the warm-up phase, we execute the benchmarks 15 times and report the average execution time. The same Java process runs for all 25 iterations. This methodology is by design to provide more accurate and reliable results. Besides, it mimics more realistic workloads in which the GPU can be used under demand for long-running applications multiple times, not just for a single kernel.

Table 3 details the two input sizes (small and large) used for each benchmark. Note that we run with a maximum number of 4GB. This is because, as we compare against GPU programs using device memory, we are limited to the maximum memory we can allocate on the device, which, in practice, (for Intel architectures) it is less than the maximum physical memory, due to reserved space for the GPU driver. Thus, to easy compare in the future with discrete Intel GPUs, we kept 4GB as maximum size.

Each benchmark was executed with two implementations: a) the equivalent code expressed in CUDA dispatched through the CUDA Driver API to run on a discrete NVIDIA GPU; and b) the equivalent SPIR-V binary code compiled with LLVM [34] from the OpenCL source code, and dispatched through the Level Zero API to run on the Intel integrated GPU. Besides, we analyze the best and worst case scenarios using the different configurations described in Table 1.

Note that our work does not compare MaxineVM versus other mainstream VMs, such as OpenJDK. Other work has already analyzed such comparison [31]. Instead, we compared our approach using both GPU code (as baseline as well as enabling UM). In the case of the performance on CPUs, we compared with MaxineVM for both UM and without UM.

Table 4. Speedup of UM for the Renaissance Benchmarks compared to unmodified MaxineVM without UM.

Benchmark	CUDA	Level Zero
akka-uct	0.88	1
db-shootout	0.96	0.99
dotty	0.95	1.02
fj-kmeans	1.08	1
future-genetic	0.99	0.99
mnemonics	0.96	0.99
par-mnemonics	0.97	0.98
philosophers	1.01	0.98
reactors	0.99	0.99
rx-scrabble	1	0.99
scala-doku	1.01	0.99
scala-kmeans	0.97	0.99
scala-stm-bench7	0.97	1.03
scrabble	1.13	1.03
Geomean	0.989	0.997

4.2 Performance on CPUs

First, we analyze if there is any performance penalty of using UM when the GPU is not used. To answer this question, we executed the DaCapo [6] and Renaissance [45] benchmark suites on the CPU using UM in CUDA and Level Zero. Then we compared the results against unmodified MaxineVM runs that do not use UM. Tables 4 and 5 show the performance results of each benchmark for Renaissance and DaCapo normalized to the execution without UM (> 1 is speedup).

In general, we observe two trends. The first one is that UM with Level Zero poses less overhead compared to the CUDA implementation. Results indicate that when running with Level Zero UM, we observe on average the same performance compared to the runs that do not use UM. On the contrary, we notice that CUDA UM results in almost consistent performance degradations for both benchmark suites (2-3% on average). This difference is attributed to the different underlying implementations of the two drivers.

Note that, although some benchmarks show small speedups (e.g., scrabble/fj-kmeans benchmarks) we do not consider

Table 5. Speedup of UM for the DaCapo Benchmarks compared to unmodified MaxineVM without UM.

Benchmark	CUDA UM	Level Zero UM
avrora	1.011	1.003
eclipse	0.977	1
fop	0.973	0.998
h2	0.95	0.996
jython	0.989	0.995
luindex	1.011	1.016
lusearch	1.009	1.032
pmd	0.972	1
sunflow	1.007	1.028
xalan	0.984	0.998
Geomean	0.988	1.006

this number to provide significant proof of speedup. This is because there are many software layers that can influence performance [30, 47], even if we run multiple iterations. Instead, what this demonstrates is that when using UM and the program does not use a GPU, the application does not slowdown.

The second trend is that the discrete GPU using UM exhibits its worst performance in a certain application of the Renaissance suite; namely, `akka-uct`. This particular benchmarks is significantly more memory intensive compared to the rest [41], and hence any additional overheads of the UM library implementations are further exacerbated. From our results, we conclude that for non memory-intensive workloads there is no significant overhead when utilizing UM. However, for memory-intensive workloads, the overheads can exceed 12% in the case of CUDA UM for discrete GPUs.

4.3 Performance on GPUs

The second set of experiments examines the effects of UM when GPU hardware acceleration is utilized. First, we provide a detailed analysis of the performance obtained with our approach compared to GPU acceleration without using UM. Consequently, we evaluate the GPU memory page faults.

4.3.1 End-2-End GPU Performance. Figure 7 shows the performance of the GPU accelerated applications when UM is employed normalized to the non-UM executions (baseline) - values over one denote a speedup of UM. The baseline implementation sends data to the GPU upfront by using the cuda copy functions (`cudaMemcpyHostToDevice` CUDA command). By the time the GPU kernel starts running, all data has already been transferred to the GPU's global memory. In the case of UM, the GPU kernel can start running without the data being transferred to the GPU's global memory. Then, the CUDA (or the GPU) driver will page fault for every memory page that is missing during execution of the kernel. The same memory page fault can be initiated by many threads concurrently as stated by NVIDIA [49]. Similarly to the copy-in, the baseline implementation also performs a copy-out after the kernel has been finished using the CUDA command `cudaMemcpyDeviceToHost`, which performs a full data migration of the data without page faulting.

For the CUDA implementations with UM (Figure 7, left two plots), we can clearly observe that in two configurations (*Alloc-Always* and *Alloc-Once-GC*), UM results in slowdowns. This is due to the page migrations that these two configurations pose and the associated overheads when using the CUDA managed memory (i.e., many concurrent threads triggering memory page faults and the driver invalidating and updating page tables between the CPU and the GPU) [57]. It is the work of the GPU driver to bring the missing pages and perform the data transfer needed. This creates an additional overhead. Thus, depending on the data layout, and memory access patterns, these overheads can be more significant. From Figure 7, the `BS` benchmark is memory intensive, while the `DFT` and `NBODY` benchmarks are compute intensive.

The only configuration that shows a performance benefit compared to the baseline is the *Alloc-Once-No-GC*. This is because, in the UM version, there are no memory pages reclaimed back from the host (CPU), once they have been migrated to the GPU, while the baseline version always performs an explicit data transfer after the kernel is finished. Otherwise, the performance would be similar to the baseline implementation, which performs the data transfer to the host again[2]. One benefit of UM, as discussed in Section 3, is that a GC can be invoked in the middle of the GPU execution, while the version executed for the baseline will provoke segmentation faults if data is moved. Thus, only the baseline implementation blocks the GC while the application is running in order to provide correct behaviour.

For the `DFT` and `NBody` benchmarks and the large data sets, the *Alloc-Always* mode offer very low performance when using UM compared to their respective baselines (0.001x). This is due to the small set of data to be transfered compared to the computation and the high number of page faults from the host side, as we will see in Figure 8.

Regarding the Level Zero (L0) implementation for the integrated GPU (Figure 7, right), the baseline version uses device-type buffers. L0 provides three types for UM: *Shared*, *Host* and *Device* types [26]. In our baseline we selected the device-type, which maximizes memory throughput within the GPU, and it is the most common type when programming for GPUs. We notice that the performance of the UM in the *Alloc-Always* mode is slightly higher (up to 9.3x) compared to the one using device memory. This is because device memory types in Level Zero are owned by the device, and not by the host (CPU). Thus, although there is no data migration on an integrated GPU, there is an explicit copy from one type of buffer (host memory) to the device memory. The *Alloc-Once-GC* and *Alloc-Once-No-GC*

[2]Note that the performance of data transfers in CUDA using UM can be further increased by using the memory advise and memory prefetch CUDA functions, in which the runtime system can provide hints about when a certain amount of memory pages can be transferred to the device [10, 49].

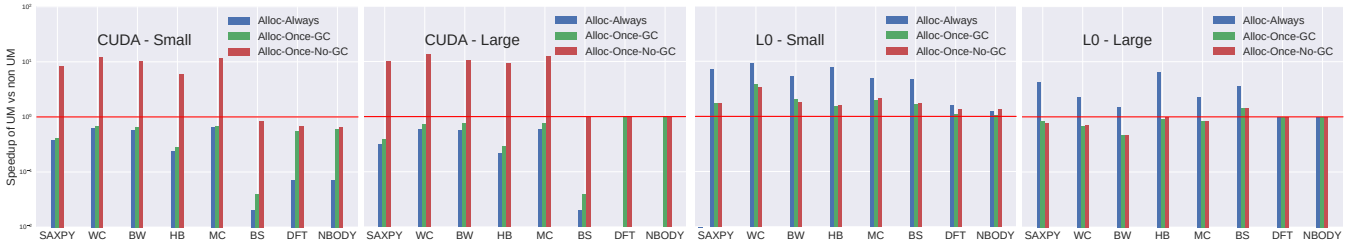

Figure 7. GPU Performance (CUDA for the two on the left-hand side and Intel Level Zero for the two on the right-hand side) compared to the GPU Accelerated version without shared memory (baseline). The higher, the better.

modes perform slightly better when using UM for memory-bound benchmarks (*saxpy*, *WC*, *MC*, *BW*, *HB* and *BS*) and small input sizes. When running with the large sizes, Level Zero with UM performs up to 0.46x compared to Level Zero without UM for the *Alloc-Once-GC* and *Alloc-Once-No-GC*. This is due to GPU driver overheads when accessing the same memory page comparing to dedicated memory on the GPU (baseline). Regarding the compute-bound benchmarks (NBody and DFT), the performance of UM is almost the same as the baseline.

From these results we conclude that using discrete GPUs for hardware acceleration with UM, the driver might introduces overheads but it simplifies the data management and it prevents segmentation faults when the GC runs. When using integrated GPUs, applications are expected to run at least at the same speed as programs with device memory for small input sizes, and at the same speed in *Alloc-Always* mode.

4.3.2 GPU Memory Page Migrations.

We analyze the number of page faults on both the CPU and GPU, the number of bytes transferred between them, and their correlation with the kernel execution time. To achieve that, we used the *Nsys* profiling tool for NVIDIA GPUs. Unfortunately, there is no similar driver tool for Level Zero and Intel integrated GPUs. Thus, for this particular analysis, we focused on the CUDA UM on the discrete GPU.

Figure 8 shows the performance results for all the benchmarks. We report the number of page faults for both the CPU and the GPU and the number of bytes transferred from the host to the device and vice-versa.

CPU and GPU Page Faults. The first two bars of each plot in Figure 8 show the number of memory page faults for the CPU (first bar) and the GPU (second bar) reported by the NVIDIA *Nsys* tool. In general, we observe that the GPU page faults is 10-20 times higher than the CPU page faults, and this is consistent across all configurations and benchmarks used. A GPU memory page fault reported by *Nsys* means that the GPU accesses a memory page that resides on the CPU, and therefore, provokes a memory page migration. This is because, a memory page fault on the GPU can be originated from multiple threads, and the GPU driver coordinates and handles page faults accordingly [49]. A trend we see is that, for the *Alloc-Always* (AA), and *Alloc-Once-GC* (AOGC), the number of page faults is almost the same. This is due to memory page replacement since we run the benchmarks multiple times.

We also see that the mode *Alloc-Once-No-GC* (AONOGC) reports less page faults than the other modes. This is because, as we saw in Figure 7, once the data is transferred from the first execution, it remains on the GPU.

Data Transfers. We can also relate the number of page faults with the amount of data being transferred. The last two bars of each configuration and size from the plot in Figure 8 show the amount of megabytes transferred between the CPU and the GPU. For the *AA* (Alloc-Always) configuration, all data is being transferred to the device, and no data should be transferred to the host, unless the data is reclaimed again on the host side. However, we observe that for the large data size, there is also data being transferred to the host. This is because not all data fits on the GPU (after all 25 runs - 10 warmup runs plus 15 runs), and the GPU driver applies page-replacement policies, which transfers data back to the host to create space in the GPU's memory. We can see this effect for the *AA* configuration between the small and large data sizes for the *saxpy* computation. Note that, due to small sizes used for the *DFT* and *NBody* benchmarks, this scenario is not visible.

For the *AOGC* configuration, the amount of data to transfer to both sides (host to device and device to host) is exactly the same. This is because, after the GPU kernel is launched, an explicit GC is invoked triggering memory page migration back and forth. Finally, as we discussed in Figure 7, there is no migration from the host to the device once the kernel is finished when using the *AONOGC* mode. This is because the final results are not claimed back from the host side in these configurations and benchmarks.

4.3.3 GPU Kernel Runtime.

We also study the kernel execution time for each configuration. Table 6 provides all kernel execution times for each mode and data size. The fastest kernel configuration is achieved with the *AONOGC*, which transfers the whole data set to the GPU on the first iteration when the application is executed. This is because, as soon as data have been migrated to the GPU after the first GPU execution, there are no memory page faults.

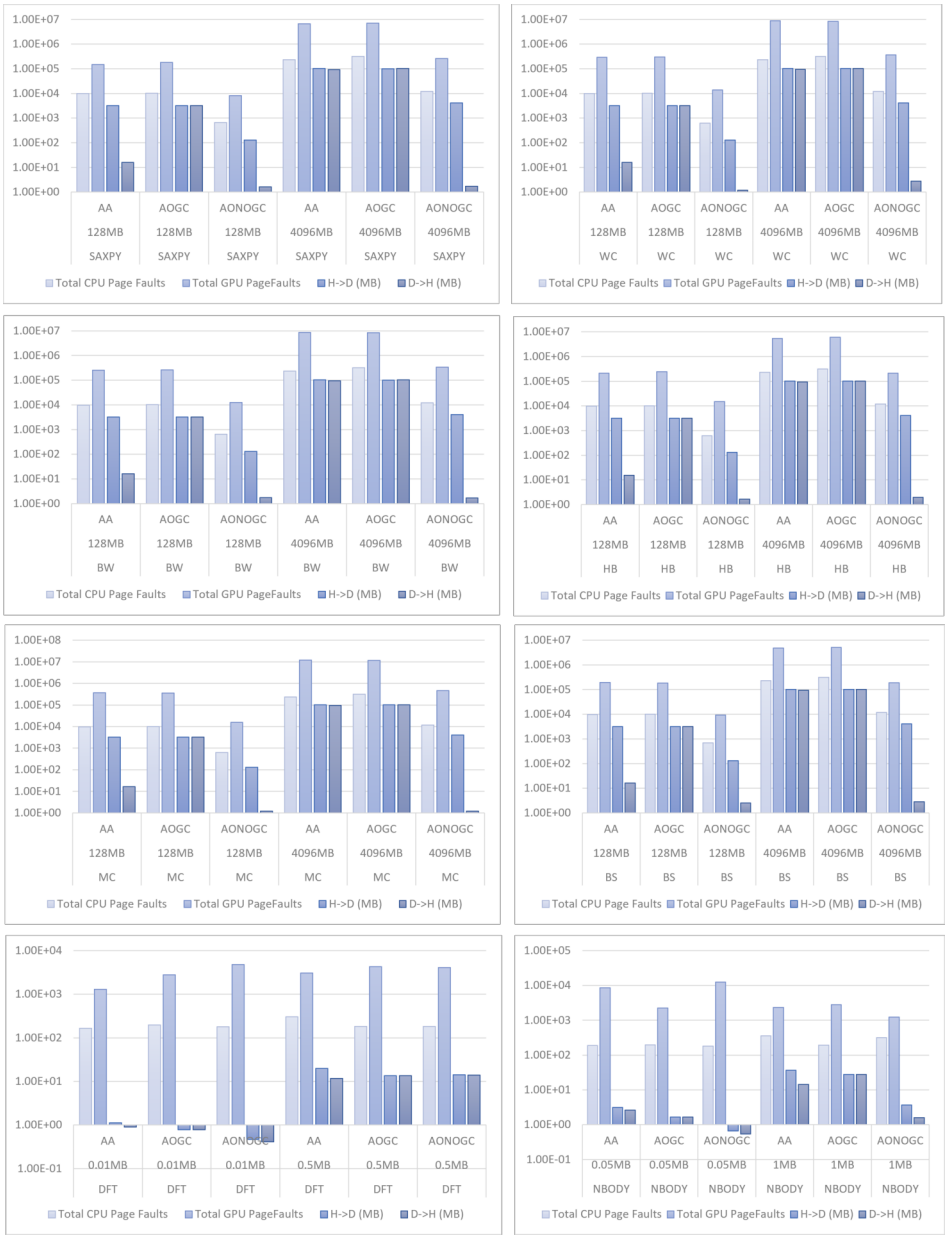

Figure 8. Profiling metrics reported by the NVIDIA NSys tool for small and large data sizes and the three configurations.

An interesting result is the kernel runtime for both *AOGC* and *AA* configuration modes. With these two modes, the kernel runtimes are almost identical. This is due to the amount of memory page faults being also identical: for the *AA* configuration, new pages are migrated for every new execution, and for the *AOGC* configuration, the same pages are migrated back and forth between the CPU and the GPU.

For compute-bound benchmarks (*DFT* and *NBody*) the kernel runtime is almost the same for all modes. This is because the computation hides the performance penalty of the memory page-faults and migrations.

4.4 Analyzing GC overheads

Finally, we also evaluated the impact on the GC when using CUDA and Level Zero UM using the *AOGC* mode. Figures 9 and 10 show the slowdown of the `System.gc()` function call in MaxineVM, which performs a full GC and move all objects from one semi-space to the other. We measured the time that takes to run System.gc() for all benchmarks with a) Unmodified MaxineVM (without UM as the baseline) and b) with UM enabled for both NVIDIA CUDA and Intel Level Zero APIs. Due to space limitation, we show the worst case scenario for the largest data set (4GB). We observe that the GC can take up to 2.13x more time when running on CUDA

Table 6. GPU Kernel Runtime per Mode.

Benchmark	Size	AA (ms)	AOGC (ms)	AONOGC (ms)
Saxpy	128MB	16.055	15.284	0.682
Saxpy	4096MB	571.437	537.863	16.965
WriteConstant	128MB	16.98	15.709	0.871
WriteConstant	4096MB	534.681	468.986	22.849
BlackAndWhite	128MB	19.055	17.774	1.217
BlackAndWhite	4096MB	612.217	555.206	31.740
Hilbert	128MB	16.782	15.838	0.526
Hilbert	4096MB	480.353	468.887	11.573
Montecarlo	128MB	17.788	16.453	0.99
Montecarlo	4096MB	569.769	527.349	26.932
BlackScholes	128MB	16.036	15.062	0.630
BlackScholes	4096MB	518.642	452.554	16.011
DFT	0.01MB	0.322	0.384	0.303
DFT	0.5MB	39.949	39.836	39.963
NBody	0.05MB	0.372	0.479	0.358
NBody	1MB	11.739	12.440	11.056

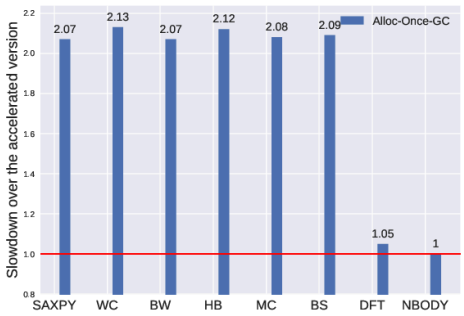

Figure 9. Slowdown of the full GC when CUDA UM compared to the default CPU memory allocator of MaxineVM. The lower, the better.

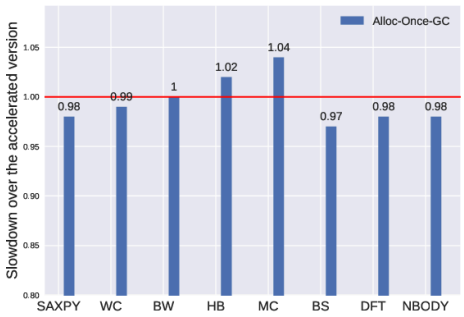

Figure 10. Slowdown of the full GC when for Level Zero UM compared to the default CPU memory allocator of MaxineVM. The lower, the better.

UM (Figure 9). This is the worst case scenario, in which the data that corresponds to the GPU execution has already been migrated to the GPU's memory. In turn, the GC, since it touched the whole heap, generated page faults in order to migrate the data from the GPU to the CPU. In contrast, for small data sizes in UM, there is no performance penalty when performing a full GC, as we see from the benchmarks *DFT* and *NBody*.

Figure 10 shows the performance overheads for the GC call when using the UM from Level Zero on integrated Intel GPUs. In contrast to discrete GPUs, there is no page migration on integrated GPUs, and therefore, we observe reduced performance penalties when a full GC is executed. At most we observe a 4% performance difference compared to the default non-shared memory allocator for MaxineVM which is within the error margin. This number refers to the deviation range as obtained by the number of runs that we ran for each benchmark.

What we conclude is that running UM on integrated GPUs introduces little overheads over the default GC, while using CUDA UM on discrete GPUs the performance penalty on GPUs is high. This means that, if users want to run with large data sizes while keeping low GC overheads, other alternatives are needed, e.g., use off-heap data.

Number of GCs. Concerning the number of GCs per configuration, the *AOGC* always trigger 25 GCs (10 for a warm up phase and 15 for the measurement iterations) for both input sizes. The *AA* and *AONOGC* modes triggered zero GCs for small input sizes, and 12 GCs and zero GCs respectively for the largest data size. The low number of GCs for these two configurations is due to the use of large heaps as UM.

4.5 Discussion: Integration with Concurrent GCs

The results presented in this section aim to demonstrate the best and worst case scenarios (along with a middle ground) of using UM in managed programming languages regardless of the GC type used. Although we conducted this work mainly on a STW SemiSpace GC, below we outline how UM could be integrated with more widely used GC such as the G1 [40] or Shenandoah [16].

Since most modern GCs do not provide a linear memory region for hosting all generations but rather they split the heap into multiple regions, we propose to designate a number of these regions to be allocated in UM. The size or the amount of these regions could be user configurable. In typical GC cycles, these regions are not evacuated by the GC. Instead, only when specific conditions are met (defined by the user or by the VM), those regions are being handled by the GC invoking the sync points explicitly during this phase. A challenge with this proposed solution is the optimization of the fast-path for allocating objects. Since all GCs use Thread-Local-Allocation-Buffers (TLABs) for fast allocation, adding control flow in order to decide where to allocate an object (normal space or UM region) would impose significant slowdowns. A potential solution would be to annotate the objects or arrays that we want to run on the GPU (and hence allocate in the UM region), and augment the compiler to force slow-path allocation for those objects where this control flow can be added. Depending on the use case, this slowdown may vary.

5 Related Work

To the best of our knowledge, this work is the first to implement and evaluate a managed heap in Unified Memory.

Thus, we focus on related work that specializes the Java heap and provide automatic memory management across heterogeneous devices.

Specialized Java Heap. Although we provide the first implementation that allocates the Java heap in a Unified Memory Space, there are other works that have demonstrated the potential of specializing the Java heap for different purposes. Espresso [53] extends the JVM with the capability to use Non-Volatile Memory through a Persistent Java Heap. It also exposes a user-level API for manipulating objects in persistent memory through Java Persistent Objects. Similarly, we also expose an interface to the user for querying the driver, device, and context objects from the VM.

Performance-Impact Memory Allocation (PIMA) [1] is a system that partitions the Java heap into regions to use the Intel Optane non-volatile memory. PIMA introduces an interesting concept, in which heaps with different purposes can live in the the same system. The objects that need to be persistent can be promoted to the non-volatile space. Our technique for Unified Memory could be also used by runtime systems in which executions on hardware accelerators can benefit from data migration to a shared memory space.

Gomez et al. [22] proposed a Java heap partition system organized in several memory banks to improve energy efficiency during GC. Although we did not create more than two partitions (in a *SemiSpace* GC), our approach can be also used in similar partitioned heaps (e.g., in Level Zero it is possible to create host and device shared memory types).

Liu et al. [36] proposed a shared Java heap partition system divided into several segments for storing server and application components for Java server applications. While our focus is a single process that can run efficiently on hardware accelerators, the Unified Memory region could be also shared with other Java processes. This is possible because, in our approach, the VM exposes to the applications the same driver, device, and context low-level objects.

Handling Automatic Memory Management. Bertels et al. [4] proposed an extension of the JVM to automatically manage the GPU and CPU memory. In Bertels' approach, the GPU memory is used as an extension of the Java heap memory space, and his work is focus on maximixing object placement between the CPU and the GPU. Our approach differs in that the heap is directly used as a UM space, rather than having two separate heaps.

Another common solution to perform data management for heterogeneous execution from Java is to lock the GC while the application runs on native code on the target accelerator (this is how Aparapi and TornadoVM currently work). However, in most cases, this is not desirable since it stops the Java application for a non-deterministic amount of time while the native code runs on the accelerated code. Our approach can complement existing frameworks such as Aparapi [17], Marawacc [18], IBM GPU J9 [28] and TornadoVM [19], which, by default, do not use custom types off-heap. Additionally, our work can be extended to other runtime systems such as Dandelion [48], Python [46], R [20], and Julia [5],

6 Conclusions

Unified Memory (UM) between CPUs and GPUs is a resource rarely exploited by managed runtime environments. In this paper, we explore the implications of storing the Java heap inside UM. By doing this, Java applications accessing GPUs (or any other accelerator) automatically send data back and forth between the CPU and the GPU. Besides, the JVM is aware of the GPU data and, in the case of a GC, data can be migrated to the CPU without causing memory faults by inserting a GPU command queue synchronization point before the GC. To the best of our knowledge, this is the first work that combines UM in the context of a managed heap.

Our technique has been implemented in MaxineVM, a research VM for Java written in Java. The heap allocation mechanism of MaxineVM has been enhanced to utilize the UM on CUDA-compatible discrete and Intel integrated GPUs. This enabled GPUs to directly access Java objects allocated in the JVM heap. We evaluated our approach on a discrete and an integrated GPU, and we showcased that while user applications can still benefit from the GPU's performance, the managed runtime system can also perform full GCs. We also show that the GC using UM introduces 4% overheads when running on Intel integrated graphics, and up to 2.13x against Java default non-shared memory allocator. Furthermore, the results show the potential of using UM by default, in which it can still run CPU workloads with an overhead of up to 12% (worst case) and 2% (average) for DaCapo and Renaissance benchmarks. We conclude that, due to the low overheads on integrated GPUs, UM is a suitable resource to be exploited as a Java heap, which benefits execution on heterogeneous devices and works seamlessly with the GC. Furthermore, we showed that, if hardware acceleration is used, UM can achieve up to 9.3x speedup compared to the non-UM baseline implementation. For future work, we propose the integration of UM with Project Panama [39] in order to allow users to define whether or not they intend to utilize this feature during hardware acceleration.

Acknowledgments

This work is partially funded by Intel Corporate Research Council and by the European Union's Horizon 2020 programme under grant agreement No 957286 (ELEGANT). Additionally, it is funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for grant numbers 10048318 (AERO), 10048316 (INCODE), 10039809 (ENCRYPT) and 10039107 (TANGO). The authors would also like to thank the anonymous reviewers as well as Peng Tu (from Intel) and Alberto Magni for fruitful discussions that helped to improve the paper.

References

[1] Shoaib Akram. 2021. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Trans. Archit. Code Optim.* 18, 3, Article 29 (apr 2021), 26 pages. https://doi.org/10.1145/3451342

[2] AMD. Last Access: February 2023. AMD Tools and SDKs. https://developer.amd.com/tools-and-sdks/

[3] AMD. Last access: June 2023. HIP Programming Manual. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html

[4] Peter Bertels, Wim Heirman, Erik D'Hollander, and Dirk Stroobandt. 2009. Efficient Memory Management for Hardware Accelerated Java Virtual Machines. *ACM Trans. Des. Autom. Electron. Syst.* 14, 4, Article 48 (aug 2009), 18 pages. https://doi.org/10.1145/1562514.1562516

[5] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 827–841. https://doi.org/10.1109/TPDS.2018.2872064

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190. https://doi.org/10.1145/1167515.1167488

[7] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 47–56.

[8] Cen Chen, Kenli Li, Aijia Ouyang, Zeng Zeng, and Keqin Li. 2018. GFlink: An In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (2018), 1275–1288. https://doi.org/10.1109/TPDS.2018.2794343

[9] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (nov 1970), 677–678. https://doi.org/10.1145/362790.362798

[10] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014

[11] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. ACM, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3237009.3237016

[12] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. ACM, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3237009.3237016

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

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[15] Naila Farooqui, Christopher J Rossbach, Yuan Yu, and Karsten Schwan. 2014. Leo: A profile-driven dynamic optimization framework for {GPU} applications. In *2014 Conference on Timely Results in Operating Systems ({TRIOS} 14)*.

[16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[17] Gary Frost. 2011. Aparapi in amd developer website.

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

[19] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware.. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. Association for Computing Machinery. https://doi.org/10.1145/3313808.3313819

[20] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/3050748.3050761

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

[22] Ricardo Gomez, Flavius Gruian, and Liang Liu. 2016. Memory Power Management for Java Processors Using Heap Partitioning and Power Gating. In *Proceedings of the 14th International Workshop on Java Technologies for Real-Time and Embedded Systems* (Lugano, Switzerland) *(JTRES '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/2990509.2990514

[23] Khronos® OpenCL Working Group. Last Access: Feb 2023. The OpenCL™ Specification. https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

[24] The Khronos® SYCL Working Group. Last Access: Feb 2023. SYCL™ 2020 Specification (revision 6). https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[25] IBM. Last Access: Feb 2023. Writing Java applications that use a graphics processing unit. https://www.ibm.com/docs/en/sdk-java-technology/8?topic=egpulwo-writing-java-applications-that-use-graphics-processing-unit-linux-windows-only

[26] Intel. Last Access: Feb 2023. Intel, Level Zero. https://spec.oneapi.io/versions/latest/elements/l0/source/index.html

[27] Intel. Last Access: Feb 2023. Intel oneAPI. https://spec.oneapi.io/versions/1.2-rev-1/introduction.html

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

[29] R. Jones, A. Hosking, and E. Moss. 2016. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press. https://books.google.it/books?id=TKOfDQAAQBAJ

[30] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. *SIGPLAN Not.* 52, 7 (apr 2017), 74–82. https://doi.org/10.1145/3140607.3050764

[31] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. Association for Computing Machinery, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[32] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.

[33] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. 2014. An Investigation of Unified Memory Access Performance in CUDA. In *IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC.2014.7040988

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[35] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. 2015. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 347–348. https://doi.org/10.1109/NAS.2015.7255222

[36] Tiancheng Liu, Ying Li, Andrew Schofield, Matt Hogstrom, Kewei Sun, and Ying Chen. 2008. Partition-Based Heap Memory Management in an Application Server. *SIGOPS Oper. Syst. Rev.* 42, 1 (jan 2008), 98. https://doi.org/10.1145/1341312.1341331

[37] NVIDIA. Last access: Feb 2023. NVIDIA Pascal Microarchitecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[38] NVIDIA. Last Access: February 2023. NVIDIA CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit

[39] Oracle OpenJDK. Last Access: Feb 2023. Project Panama: Interconnecting JVM and native code. https://openjdk.org/projects/panama/

[40] Oracle. Last Access: June 2023. Garbage-First (G1) Garbage Collector. https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html

[41] Orion Papadakis. 2022. *Performance analysis and optimizations of managed applications on Non-Uniform Memory architectures*. Ph. D. Dissertation. University of Manchester.

[42] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. 2020. Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. *Programming 2020* abs/2010.16304 (2020). arXiv:2010.16304 https://arxiv.org/abs/2010.16304

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[44] LLVM Project. Last Access: February 2023. Clang: a C language family frontend for LLVM. https://clang.llvm.org/

[45] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[46] Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz. 2018. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:29. https://doi.org/10.4230/LIPIcs.ECOOP.2018.16

[47] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. 2017. Type Information Elimination from Objects on Architectures with Tagged Pointers Support. *IEEE Trans. Comput.* 67, 1 (29 June 2017), 130–143. https://doi.org/10.1109/TC.2017.2709739

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

[49] Nikolay Sakharnykh. Last Access: Feb 2022. Maximizing Unified Memory Performance in CUDA. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/

[50] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[51] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 67–75. https://doi.org/10.1145/3489525.3511691

[52] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (jan 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[53] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3173162.3173201

[54] Maria Xekalaki, Juan Fumero Alfonso, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos-Efthymios Kotselidis. 2022. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware.

[55] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*. 273–283. https://doi.org/10.1109/BigData.2016.7840613

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

[57] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. https://doi.org/10.1109/HPCA.2016.7446077

Received 2023-06-29; accepted 2023-07-31