2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-6460-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPSW63119.2024.00132

GrOUT: Transparent Scale-Out to Overcome UVM's Oversubscription Slowdowns

Ian Di Dio Lavore
Politecnico di Milano, Italy
ian.didio@polimi.it

Davide Maffi
Politecnico di Milano, Italy
davide1.maffi@mail.polimi.it

Marco Arnaboldi
Oracle Labs, Switzerland
marco.arnaboldi@oracle.com

Arnaud Delamare
Oracle Labs, Switzerland
arnaud.d.delamare@oracle.com

Daniele Bonetta
VU Amsterdam, Netherlands
d.bonetta@vu.nl

Marco D. Santambrogio
Politecnico di Milano, Italy
marco.santambrogio@polimi.it

***Abstract*—Hardware accelerators have always been difficult to approach. In recent years, we have experienced great efforts to simplify their programming paradigms, especially on GPUs. This led to the development of various domain-specific frameworks and microarchitectural features that facilitated some aspects of this multifaced problem. One such feature is the Unified Virtual Memory (UVM) oversubscription mechanism that allows the developer to handle datasets with a bigger memory footprint than the HW accelerators. Although promising, current UVM faces extreme overheads when running large workloads that reach an oversubscription factor (allocated vs. available memory) ampler than a per-workload threshold. In this work, we propose GrOUT, a language- and domain-agnostic framework that tackles the slowdowns brought by the UVM oversubscription mechanism. In particular, we highlight how a scale-out approach is a feasible solution to solve the slowdowns brought by UVM on workloads from various domains. Moreover, we design a framework capable of autonomously scaling out user-provided workloads, reaching a speedup of more than $24.42\times$ with minimal changes to the application logic.**

***Index Terms*—GPU, Distributed Systems, Unified Virtual Memory, Oversubscription, Polyglot Programming, CUDA**

I. Introduction

Due to their inherent multithreading capabilities, GPUs are widely utilized in diverse applications, ranging from Deep Learning to Genome Analysis [1], [2]. With the rising demand for handling and extracting insights from Big Data and AI workloads, compute machines equipped with multiple GPUs have been increasingly adopted in cloud data centers [3]. Unfortunately, the immense amount of computational power available in those multi-GPU architectures does not come for free to the developer. Indeed, effectively utilizing the full capabilities of a multi-GPU architecture requires expertise in asynchronous programming, and developers must consider the problem of processing large-scale datasets. The research community has highlighted the importance of this multi-faced problem and proposed multiple solutions to abstract the complexity away from the end-user. However, those solutions struggle to maintain a good ratio of usability over obtainable performance [4]. Parallel to constructing high-level frameworks, we have seen a massive effort to simplify the GPU programming paradigm. In particular, in 2014, NVIDIA released the 6th edition of the CUDA programming model that introduced **Unified Virtual Memory (UVM)** support inside the Kepler and Maxwell GPU architectures [5]. UVM aims to simplify the programming paradigm by relieving the developer of the need to manage data transfers from and to the hardware accelerators manually. This is accomplished via Unified Virtual Addressing (UVA), which internally adopts a page-faulting architecture to expose a convenient virtualized unified memory space to users. Today's workloads commonly process large-scale datasets that exceed the memory size of the hardware accelerators. On top of that, the CPU's main memory is typically at least an order of magnitude bigger than the GPU's (e.g., 80 GB vs. 2 TB on the latest NVIDIA DGX H100 system [6]). Therefore, an essential aspect of UVM is supporting the **oversubscription of the GPU memory**, that is, the ability to allocate on the HW-accelerator memory regions bigger than the on-device one, exploiting the UVA technology. Multiple GPU programming frameworks have been extended

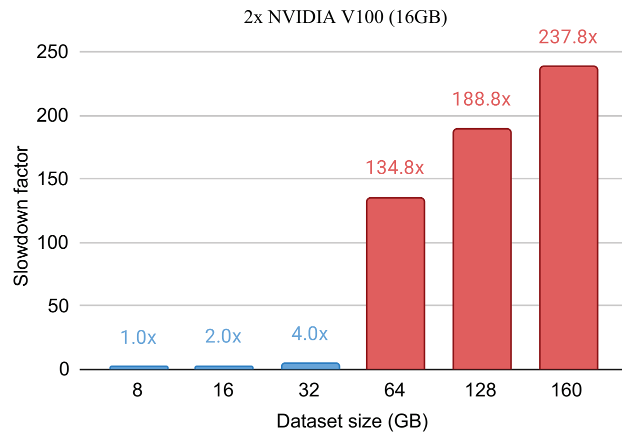


Fig. 1: Impact of UVM oversubscription in the execution time of Black–Scholes algorithm when increasing the input size. Red bars indicate runs that exceed the GPUs' memory.

This work was completed while D. Bonetta was working at Oracle Labs.

979-8-3503-6460-6/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPSW63119.2024.00132

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

to include support for UVM, enabling them to handle out-of-memory scenarios by oversubscribing the memory of the GPU. Unfortunately, as highlighted in [7], UVM oversubscription is very decremental to the workload's performance, even when working with a small oversubscription factor. In Figure 1, the slowdown factor can be observed on a massively parallel algorithm (Black–Scholes [8]) for increasing dataset sizes on a multi-GPU system with two NVIDIA server-grade GPUs. To solve this issue, users can resort to two main approaches. The first one involves multiple iterations of profiling the current UVM's oversubscription behavior and then **hand-tuning the CUDA runtime** by instrumenting the CPU-side code (e.g., with prefetching and memory advises) [9]. Moreover, developers must fine-tune the kernels' memory access logic to match the black-box behavior of the UVM. This is only sometimes feasible given that the access patterns might be intrinsically determined by the application's logic (e.g., sparse accesses). On the other hand, the second approach solves this issue at its root cause by **scaling out the available resources**, directly decreasing the oversubscription factor of each HW accelerator by distributing the workloads on multiple servers. The common practice for performing such distribution is a complex and time-consuming effort of rewriting the entire code base to match a different programming paradigm, such as CUDA-aware MPI or NVIDIA Collective Communications Library (NCCL) [10], [11]. In this work, we focus on simplifying the latter approach by providing GrOUT, a framework capable of **solving the problem of UVM's oversubscription performance loss by removing its root cause**. We provide an alternative to standard practices targeting users who might not have the expertise required to hand-tune and restructure CUDA C++ code or want to accelerate problems in novel-emerging domains. In such cases, the time-to-prototype and integration with existing pipelines is an essential aspect to consider. The key contributions of this work are the following:

1) Highlight how a **scale-out approach can solve the slowdowns brought by UVM's oversubscription** mechanism when handling large-scale workloads. Although resolute, this approach is currently limited by the engineering effort required to restructure the overall application.
2) Design of GrOUT, a **framework that enables autonomous scale-out** of GPU-accelerated applications, highly reducing the effort required to distribute a pre-existing code base to handle large-scale problems.
3) Extensive experimental evaluation on **oversubscription factors larger than previous literature** on multi-GPU architectures.
4) Implementation of a **native multi-language API** to access the functionalities of the framework from major programming languages (e.g., Python, JavaScript, Java).

The framework is released as a completely open-source project to let users exploit its capabilities[1]. The rest of the work is organized as follows. In Section II, we introduce the reader to the necessary background to fully understand the content of this paper. Section III presents the current state of the literature concerning UVM. Then, in Section IV, we present the architecture and design choices of the proposed framework. Finally, Section V details the experimental evaluation applied to validate this research.

[1]https://github.com/necst/grout

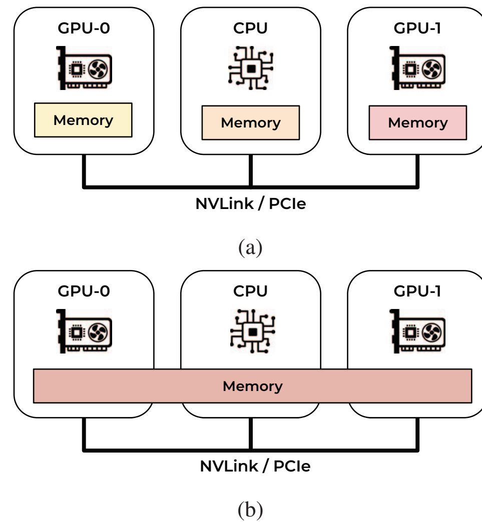


Fig. 2: High-level view of the memory model without (a) and with (b) UVM for multi-GPU systems.

II. Background

This section will introduce the necessary background on the technologies involved in this work. In particular, we will provide a brief overview of the UVM technology and of GraalVM, the base layer upon which GrOUT is built.

A. *Unified Virtual Memory*

Starting from the Kepler architecture, NVIDIA's GPUs are now supporting the UVM paradigm, enabling them to provide the end-user with a unified address space when dealing with allocated data [5]. Figure 2 depicts the high-level view of the new programming model when adopting UVM. The physically distinct memories of the hardware accelerators are now exposed to the user as a single, unified, consistent memory address space. Thanks to the addition of page-faulting and automatic migration of pages, the user can now access the same pointer in the host code (e.g., during the initialization phase) and pass it to the device during kernel launch, drastically reducing the code base complexity. Additionally, the developer can still manage the UVM through its APIs to manually optimize the data transfers (e.g., by prefetching memory regions), thereby offering mechanisms to tune memory placements. Nonetheless, it is worth noticing that the complete architecture of UVM (e.g., adopted heuristics) is not released to the public. Therefore, without manual intervention, the behavior of the UVM should be treated as a black-box model, increasing the effort required to extract maximum performance from the devices. This scenario is analogous to, e.g., cache coherence

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

protocols where users do not control what gets stored in L2/L3 caches and have to ”trust” the underlying hardware.

B. GraalVM

GraalVM is a high-performance Java Virtual Machine (JVM) created and maintained by Oracle, built upon the widely adopted HotSpot JVM. Besides Java, GraalVM can run polyglot applications written in popular programming languages like Python, JavaScript, or C/C++ [12], [13]. GraalVM integrates an advanced Just-In-Time (JIT) compiler with various optimizations (e.g., polymorphic and aggressive inlining) [14], [15], [16]. Among the multiple features of the GraalVM's compiler is the ability to run guest programming languages exploiting an independent Intermediate Representation (IR). Indeed, guest programming languages, such as the aforementioned ones, can be executed thanks to multiple interpreters built using the Truffle Language Implementation Framework (Truffle) [17]. Moreover, Truffle can be exploited to implement new libraries within GraalVM that will be directly available to all the JVM-supported languages. This aspect removes the need to develop custom bindings for multiple languages, which brings additional maintenance costs for the developer.

III. Related Works

In the literature, we can find multiple works that have focused their attention on characterizing the behavior of the CUDA UVM throughout the evolution of its architecture and programming model enhancements [7], [9], [18], [19]. Shao et al. analyze the behavior of CUDA 11.0 on the Turing architecture, highlighting the impact of Frequently Accessed but Low Locality (FALL) pages on the overall application performance [7]. Similarly, multiple works throughout different generations of the CUDA runtimes and GPU architectures have demonstrated how the effect of prefetching and memory advise is highly dependent on the application logic and therefore, they are not always a solution to optimize the UVM behavior [18], [9], [19]. Indeed, Shao et al. showed that the advanced optimization features of UVM have different outcomes (slowdowns/speedups) for the same workload when adopted under different oversubscription factors [7].

Lots of effort has been placed into the design of novel methodologies to enhance the performance of UVM. Multiple works propose new architectural designs to tackle specific application scenarios. However, given the complex runtime optimization problem of page eviction, migration, and prefetching, none are definitive for all the possible ones. Before the direct integration of a Page Migration Engine inside NVIDIA's GPUs, a notable effort has been made towards integrating and subsequently optimizing the page-faulting mechanism [20], [21]. Nonetheless, researchers have proposed multiple HW designs in the following years to cope with the inefficiency of the original design or solve new types of workloads (e.g., highly irregular applications) [22], [23], [24], [25].

On the other hand, always related to this work, we find literature concerned with the introduction of the UVM paradigm inside HW-acceleration pipelines. Fumero et al. investigated

```python
1  import polyglot
2  # Initialization
3  build = polyglot.eval(GrOUT,
     ↪  "buildkernel")
4  square = build(KERNEL, KERNEL_SIGNATURE)
5  x = polyglot.eval(GrOUT, "int[100]")
6  # Normal execution flow
7  for i in range(100):
8      x[i] = i
9  square(GRID_SIZE, BLOCK_SIZE)(X, 100)
10 print(x)
```

Listing 1: A minimal Python example of GrOUT's APIs.

the opportunities given by the direct integration of UVM inside a research-oriented JVM [26]. Parravicini et al. extended a joint effort of NVIDIA and Oracle Labs, called GrCUDA, to enable asynchronous and transparent execution of CUDA C++ code from within all of the major programming languages in a single GPU system, exploiting UVM [27]. This work shares with GrOUT the usage of a polyglot runtime to expose native CUDA C++ code to guest programming languages, although it mainly focuses on single-node, single-GPU execution. Indeed, as we present in Section V-C, their work is greatly affected by the UVM oversubscription slowdowns when scaling the memory footprint of the workloads.

To the best of our knowledge, this is the first work that examines the advantages/disadvantages of a scale-out solution to the UVM's oversubscription slowdowns. Moreover, no language-agnostic framework currently supports autonomous scale out of multi-GPU accelerated workloads with minor to no modifications to the workload logic.

IV. GrOUT Design and Implementation

In this Section, we first provide an informal introduction to the GrOUT APIs (Section IV-A). Subsequently, we delve into the details of the implementation. Section IV-B provides a bird's-eye view of the framework's architecture, going into the specifics of the scheduling procedure in Section IV-C and the implemented scheduling policies in Section IV-D.

A. GrOUT APIs

GrOUT exposes GPU functionalities to GraalVM languages such as Python or JavaScript. Using the GrOUT APIs, users can register new or existing kernels (e.g., open-source implementations) directly in the language and can allocate data that is seamlessly shared between the CPU and the GPU. Listing 1 provides an example of the usage of such APIs. In the first line, the `polyglot` package is imported into the environment. It exposes the polyglot functionalities available inside the Python implementation of GraalVM [12]. Among those functionalities, we find GrOUT, built as a library inside the GraalVM ecosystem. In Lines 3-4, the user accesses the `build` functionality of GrOUT. In particular, at runtime, it issues a build command to the NVIDIA Runtime Compiler (NVRTC) that compiles a GPU kernel from a string containing

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

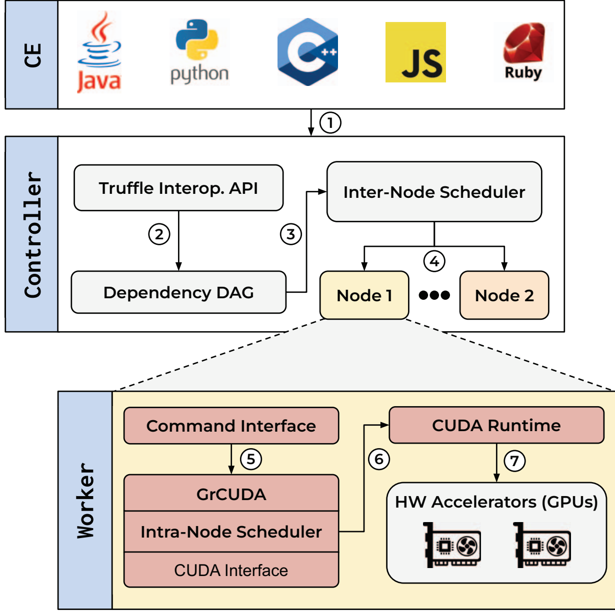


Fig. 3: High-level view of the architecture of GrOUT.

Algorithm 1: Node-level scheduling

```
input : Current CE
output: CE is assigned to a specific stream

// Add CE to Global DAG's frontier
forall frontierCE in globalDAG.frontier do
    dependencies ← computeDependencies(CE, frontierCE);
    if dependencies.size > 0 then
        ancestors.add(frontierCE);
    end
end
filteredAncestors ← filterRedundant(ancestors, CE);
DAG.addEdges(filteredAncestors, CE);
DAG.updateFrontier();

// Apply node-level scheduling policy
scheduledNode ← nodeManager.assignNode(CE, policy1);

// Issue necessary data movements
forall param in CE.parameters do
    if ¬ param.upToDateOn(scheduledNode) then
        if upToDateOnlyOnController(param) then
            scheduledNode.send(param);
        else
            P2PNode ← param.upToDateNodes();
            P2PNode.send(scheduledNode);
        end
    end
end
```

the CUDA C++ code (pre-compiled kernels are also supported). Moreover, it associates it with the variable `square`. Line 5 creates a UVM array managed by GrOUT that will be later initialized (Lines 7-8) for successive use. Finally, in Lines 9 and 10, the user launches a kernel as a regular function call and prints the modified array. It is relevant to notice that one of the framework's benefits is completely hiding away from the user the need to manage data transfer between the hardware accelerators and the host system. Moreover, transfer-computation overlap, when possible, is automatically achieved without user intervention; that is, if a kernel was launched between the `square()` function call and the `print()` with no dependencies to prior scheduled kernels, it will run in parallel with respect to the first kernels and the data movements. Indeed, we adopted the same set of APIs exposed by the GrCUDA framework [27] that supported only single-node architectures, but we transparently distribute the application for the end-user with minimal to no change in the code.

B. GrOUT Architecture: Bird's-Eye View

In Figure 3, we present the overall high-level view of the framework; it is composed of two major components: the `Controller` and the `Worker`. The `Controller` is implemented using the Truffle Language Implementation that allows it to gain polyglot capabilities, that is, the ability to interface with the major programming languages natively. The user interacts with the framework (Figure 3, ①), creating a language-independent **Computational Element (CE)**. A CE is a lightweight wrapper around all the GPU kernel launches in the host code and read/write operations on memory regions handled by the framework (e.g., array's initialization). By exploiting the information inside a CE, the framework maintains, at runtime, an up-to-date graph of the dependencies among all of the different CEs (Figure 3, ②). The scheduler uses this **Directed Acyclic Graph (DAG)** to optimize the CE placement among all the system nodes, guaranteeing the overall computation's correctness and achieving computation/computation and transfer/computation overlap when possible. After a CE is added to the dependency DAG, it is forwarded (Figure 3, ③) to the inter-node scheduler that optimizes its placement (see Section IV-C). The scheduler applies different scheduling policies that the user can select to match the behavior of its workload better (we discuss those policies in Section IV-D). Finally, the CE is assigned to a specific `Worker` (Figure 3, ④).

C. Hierarchical Scheduling

GrOUT adopts a hierarchical scheduling architecture that treats the optimization of the placement of a CE (e.g., GPU kernel) to a specific device with a two-layered architecture.

At the first layer, the `Controller` schedules tasks to nodes considering its local dependency DAG without worrying about assigning a task to a specific Stream/GPU. Since all of the CEs coming from the workload will, at some point, pass through the `Controller`, we will refer to its DAG as `Global DAG`. Algorithm 1 depicts the node-level scheduling procedure in a simplified way. The first step is to add the current CE to the `Global DAG`'s frontier. To do so, GrOUT's scheduler iterates over the current frontier of the DAG and checks if dependencies are present with respect to the current CE; if so, the frontier's element (`frontierCE`) is added to the list of ancestors. After this step, we filter the ancestor list to remove redundant links (e.g., A and B have dependencies

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: Intra-node scheduling

```
input : Assigned CE
output: CE placed for execution on a GPU's stream

// Add CE to Local DAG's frontier
// ... Omitted

// Apply the intra-node scheduling policy
selectedStream ← streamManager.assignGPUstream(CE, policy);

// Exec. CE & add sync events on ancestors
forall ancestorsComp in filteredAncestors.getComputations() do
    selectedStream.addAsyncWait(ancestorsComp.endEvent);
end
selectedStream.execute(CE);
```

against a new CE called C, but B depends on A). Finally, the DAG is updated, creating the necessary edges and modifying the frontier. The second step is to apply the selected scheduling policy to the current CE, also considering the recently updated DAG; additional details are provided in Section IV-D. The third and final part of the algorithm deals with issuing the necessary data movements between the nodes in the system. In particular, the scheduler iterates over the CE arguments, which can be either up-to-date on the scheduled `Worker` or not. Nothing needs to be done in the former case, and the computation proceeds with no unnecessary data movements. In the latter case, data movements need to be issued. If the current parameter is up-to-date only on the `Controller`, the scheduler will directly transfer the data to the assigned node. Otherwise, since some of the other nodes have the parameter in a consistent state, we issue a peer-to-peer (P2P) data transfer between the selected node and a candidate `P2PNode`. The output of the overall procedure is a node where the current CE is scheduled to be executed, which can be either a remote node (`Worker`) or directly the `Controller` in case of read/write operations on UVM-managed arrays.

At the second layer, each `Worker` receives tasks to be executed and exploits the intra-node runtime scheduler of GrCUDA [27] to optimize the utilization of the available GPUs within the single node (Figure 3, ⑤). Algorithm 2 depicts a high-level view of the intra-node scheduling procedure. In each `Worker`, the DAG represents only a partial view of the overall workload; therefore, we refer to it as `Local DAG`. Similar to the `Controller`, the first step consists of adding the assigned CE to the DAG (refer to Algorithm 1 exchanging `Global` with `Local` DAG). After this initial step, the scheduler assigns a `CUDA Stream` to the current CE by applying the selected policy on the available HW accelerators. A `CUDA Stream` can be seen as a FIFO queue where CEs are placed for execution. A single GPU can manage multiple Streams to achieve a higher overlap between transfer/computation or unrelated computations. The adopted intra-node scheduler exploits this aspect to increase GPU utilization autonomously and at runtime. Finally, asynchronous `Wait Events` are added inside the selected `CUDA Stream` to guarantee correctness with the other active CEs. In this case, the output of the scheduling procedure is where to place the current CE in terms of the Stream on a specific device. This elastic approach relieves the `Controller` of the additional overhead of handling the overall distributed system, reducing the framework's scheduling cost, which would otherwise have to keep track of the status of each GPU/Stream on each node.

D. Scheduling Policies

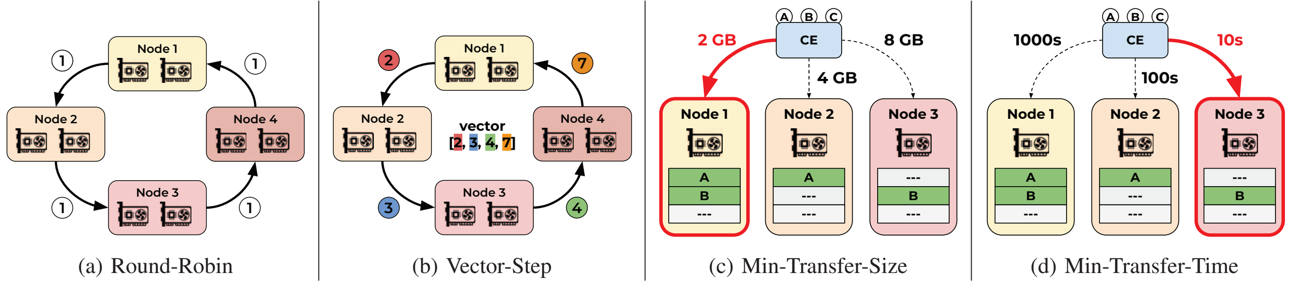
Policies can be easily implemented into the framework to match user-specific scenarios. Nevertheless, during the development of this work, we developed multiple workload-agnostic policies.

- **Round-Robin**: A simple, yet in some cases effective, policy that schedules a CE on a different node each time, following a circular pattern (Figure 4a).
- **Vector-Step**: Similar to Round-Robin, this policy accepts a vector as a parameter and assigns a pre-defined number of CEs to a specific node before switching to the next one. For example, with a vector of $[1, 2, 3]$ and two nodes, we would assign the first CE to the first node, then two CEs to the second one, and finally three CEs to the first node (Figure 4b).
- **Min-Transfer-Size**: this more informed policy considers the input parameters of the CE during scheduling (Figure 4c). In particular, it reduces the data size to be moved around the system by allocating the CE to the node where the majority of up-to-date data resides.
- **Min-Transfer-Time**: Similar to Min-Transfer-Size, this policy takes advantage of the runtime knowledge of the input parameters of a CE to assign it to the node requiring the empirically lowest transfer time of the needed data. Indeed, during the initialization of the framework, an interconnection matrix containing the bandwidth between all the nodes is constructed for later use. This is particularly relevant when working with heterogeneous interconnection types between the nodes in the systems or, for example, with Virtual Network Interface Cards (VNICs) with different SLAs.

V. Experimental Evaluation

In this section, we present an evaluation of our framework over a set of multi-GPU UVM workloads. In particular, we first explore the experimental setup adopted during our evaluation (Section V-A). Then, we present the multi-GPU UVM workloads that have been selected (Section V-B). After that, in Section V-C, we create a baseline for our implementation by investigating the slowdowns brought by the UVM oversubscription mechanism inside another GraalVM library that supports only single-node execution (GrCUDA [27]). Consequently, we perform an initial evaluation on two nodes of the same workload by implementing them with our framework (Section V-D). This served two purposes: first, we experimented with the usability of the framework, highlighting how easily it is for the end-user to adopt our framework from a GrCUDA single-node implementation of the same workloads. Secondly, we provide experimental results on the efficacy of

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.



(a) Round-Robin (b) Vector-Step (c) Min-Transfer-Size (d) Min-Transfer-Time

Fig. 4: A visual representation of the implemented inter-node scheduling policies.

our solution to eliminate (or reduce) the impact of UVM oversubscription on the overall execution times of the workloads. In Section V-E, we delve into the autonomous scaling capabilities of the framework, applying the different policies that are available to the end-user. Finally, we characterize the overhead when dealing with a larger cluster of nodes (Section V-F).

A. Experimental Setup

During our evaluation, we employed Oracle Cloud Infrastructure (OCI) resources. In particular, the GPU-equipped servers (one per `worker`) have two NVIDIA Tesla V100 (16 GB each, 32 GB total), an Intel Platinum 8167M with 180 GB of RAM and network card with a bandwidth of 4000 Mbit/s. Therefore, the oversubscription factor (allocated vs. available memory) is considered $1\times$ when the memory footprint of the workloads reaches 32 GB. On the other hand, the `controller` server is equipped with an Intel Xeon 6354 with 256 GB of RAM and a peak network bandwidth of up to 8000 Mbit/s. No particular setup is created on OCI for the allocated resources (e.g., Bare-Metal, Clusters optimizations) to provide an evaluation to the end-user of the more cost-effective setup. Each test is repeated ten times adopting the arithmetic mean to average the results. Given that single run execution times of some of the tests scaled more than exponentially, we capped them to a maximum of 2.5 hours for each run.

B. Workload Suite

We selected three workloads from the publicly available GrCUDA's suite [27]. They represent a valid candidate in the scope of our work since they have already been implemented and optimized using NVIDIA's UVM. In Figure 5, we present the `Global DAG` of each workload. The number inside of each circle represents a possible schedule on different nodes. In particular, we selected the Machine Learning Ensemble Model (MLE), Conjugate Gradient (CG), and Dense Matrix-Vector Product (MV). **MLE** is an inference on an ensemble model composed of two different pipelines that present an imbalance in the execution time of each branch. On the other hand, **CG** is composed of multiple inter-dependent CEs that stress network communication. Finally, **MV** is a row-partitioned matrix-vector product on a dense matrix. Each

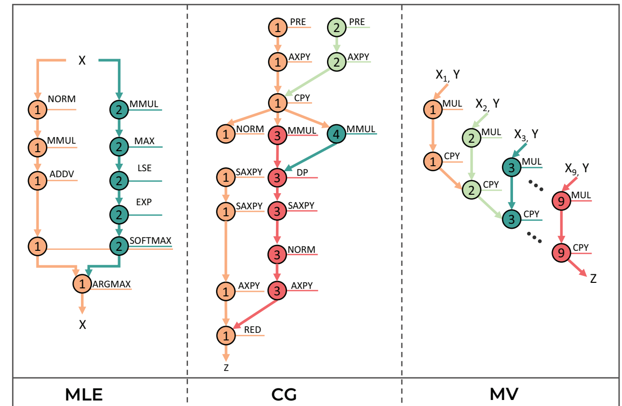


Fig. 5: Workloads CEs dependencies.

```
1 import polyglot
2 ### GrCUDA ###
3 X = polyglot.eval(GrCUDA, "float[SIZE]")
4 ### GrOUT ###
5 X = polyglot.eval(GrOUT, "float[SIZE]")
```

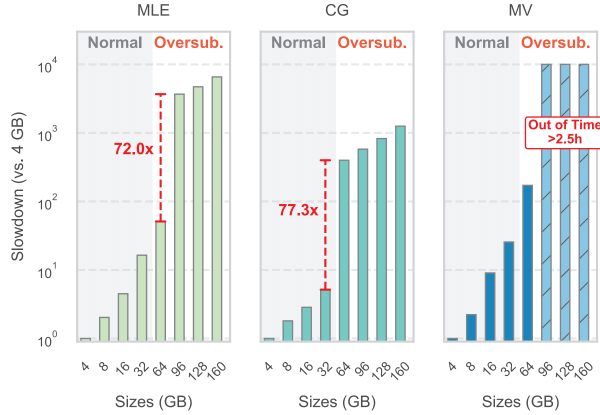
Listing 2: Code changes required to distribute the workload from the GrCUDA's workload suite.

workload has been profiled to find the correct input sizes to generate a memory footprint for the desired oversubscription level (4 GB to 160 GB). It is essential to note that thanks to the APIs of the framework, minimal changes were required to the original code of the workloads to distribute them on multiple nodes. In particular, as highlighted in Listing 2, the developer needs to just specify the language of `GrOUT`.

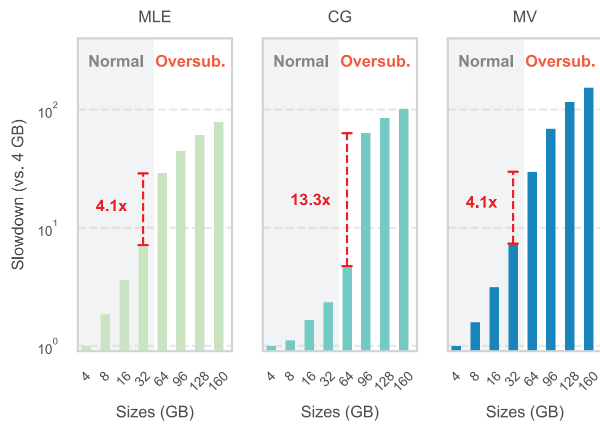
C. Workloads' UVM Behaviour

As a first step, we inspected the performance of the open-source workloads. This has been done for two main reasons: firstly, showcasing that the degradation of performances under oversubscription is also present in the GrCUDA framework and not only in plain CUDA UVM code, and secondly, verifying that the selected workloads were affected by it. Figure 6a depicts the results of our characterization. In particular, all the workloads scale almost linearly when increasing the dataset

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.



(a) Single Node



(b) GrOUT - Offline Policy

Fig. 6: Slowdowns with respect to 4 GB execution when increasing the dataset size up to 160 GB (5× of oversubscription factor) for both single-node and two nodes (GrOUT).

size, but this behavior drastically changes when reaching a specific oversubscription level, namely 2x (64 GB) or 3x (96 GB). CG and MLE show a comparable performance degradation after reaching that threshold with values in the range of 70× of slowdowns. For the massively parallel MV, the scenario is even more exacerbated, arriving at execution times slower than 342× (instead of a theoretical 1.5×) when the working set of the workload is increased only by 50% over the previous one. Therefore, the selected workloads from the GrCUDA suite are indeed affected by performance degradations when working with out-of-memory sizes and UVM.

D. Initial Evaluation on Two Nodes

- *Are we eliminating (or reducing) the performance degradation of UVM when distributing the workload?*

To investigate this aspect, in Figure 6b, we show the GrOUT execution times for the workloads on two nodes. We use the smallest size as a baseline (4 GB equivalent to an oversubscription factor of 0.125×) and `vector-step` as scheduling policy. Comparing the results with the one from Figure 6a, the impact of UVM oversubscription is significantly reduced. The highest slowdown between different oversubscription levels was 342.6× in the MV workload; now, the same step is almost linear with a value of 4.1×. For what concerns CG and MLE, we reach respectively 13.3× (instead of a slowdown of 77.3×) when switching from 64 GB to 96 GB and 4.1× (instead of a downturn of 72.0×) from 32 GB to 64 GB. Therefore, scaling out user-provided workloads renders it possible to reduce the effect of UVM's oversubscription.

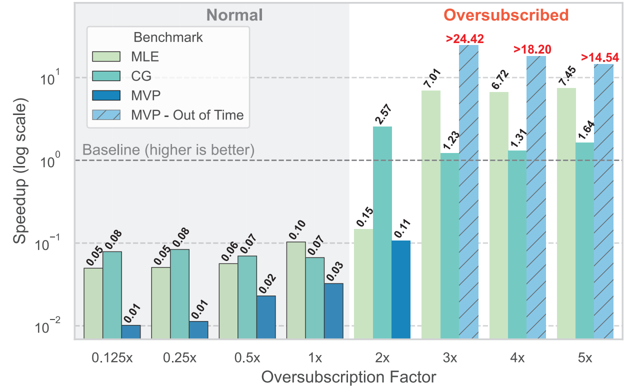


Fig. 7: Characterization over increasing values of oversubscription factor of the speedup of GrOUT against a single-node execution respectively on two and one equally equipped nodes with two NVIDIA V100 (16GB).

- *Is scaling out a solution to achieve better performances for the same size over a single-node execution?*

We performed an initial evaluation of the performance of GrOUT, optimizing the placement of different CEs using an offline policy (`vector-step`) provided by the end user. We executed each workload using GrOUT on two nodes against the original versions implemented with GrCUDA on a single node. Figure 7 depicts our study showing on the y-axis the speedup compared to the single-node execution for the same level of oversubscription factor. **Under normal conditions**, i.e., when the workload size does not reach the oversubscribed scenario, the single-node execution performs better. This is unsurprising as GrOUT pays the network cost of moving data between the different systems. **Switching to the oversubscribed scenario**, things get interesting, particularly at the 2× oversubscription level; only CG benefits from workload distribution, while MLE and MV are still perform better on the single node. But, by just going a single step further, all the different workloads are performing better on a distributed setup (including network transfers and scheduling overhead). GrOUT enables speedups of up to 1.64× for MLE, 7.45× for CG, and above 24.42× for MV, where we went out-of-time in the single-node execution.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

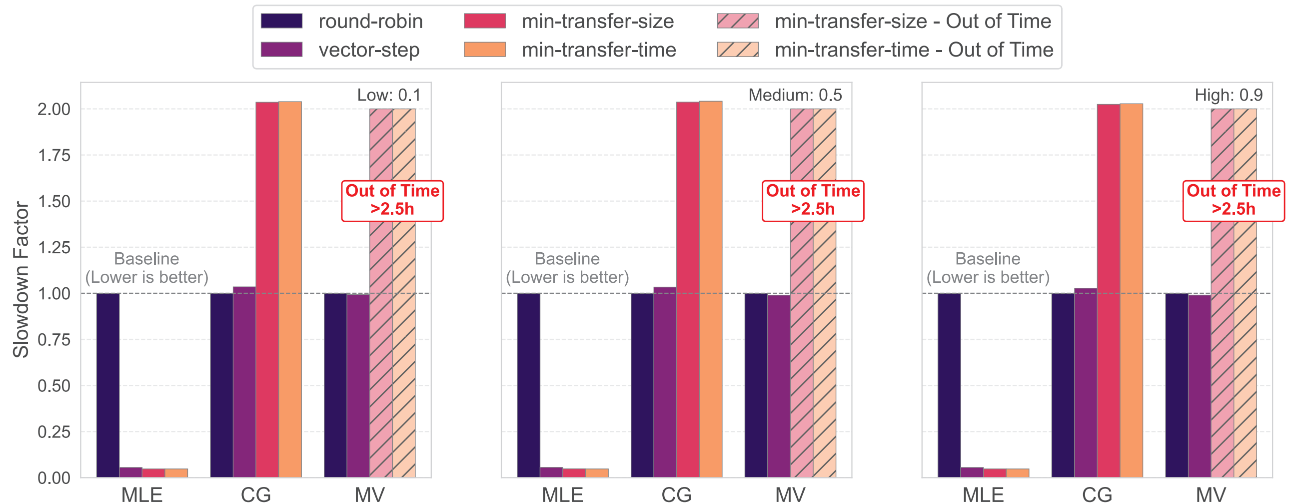


Fig. 8: Execution time of the different workloads while adopting online and offline scheduling policies with 3× oversubscription factor (dataset size of 96 GB). `round-robin` policy as a baseline (lower is better). The three plots depict the behavior under different exploration vs. exploitation heuristics ratios.

E. Autonomous Scheduling: Considerations & Evaluation

In the previous subsection, we presented the results of GrOUT concerning its ability to transparently scale out workloads for the end user, reducing the slowdowns brought by the UVM oversubscription mechanism. Here, we focus our attention on our efforts to provide online scheduling policies that can automatically, **at runtime**, place CEs on different nodes based on data locality and network transfer speeds. It is essential to consider that our framework is designed to accommodate any user application in a workload and domain-agnostic way. In particular, we do not pose constraints to the developer while he is creating its application, keeping the original logic intact. This serves to ease the usage of multiple HW accelerators even in distributed environments for **inexperienced users and novel-emerging domains** where the experience of multi-GPU programming and distributed systems might be limited. Not surprisingly, this comes with significant challenges and disadvantages to our scheduler. To accommodate workloads in an agnostic way, we do not base our scheduling heuristic and logic on the inner code inside CEs but rather on the dependencies created among them. In particular, we evaluated the benefits of adopting the `min-transfer-size` and `min-transfer-time` policies presented in Section IV-D. Those policies consider the interdependencies between CEs and combine this knowledge with data-locality considerations (the `size`) and network metrics (the `time`). As a baseline, we selected the naive `round-robin` policy. Moreover, for what concerns the roofline for the performance, we consider the user-defined vector-step policy that, being an offline policy, can be customized to better map to the workload if the CE dependencies patterns are known a priori. Figure 8 depicts our results. Each plot shows a different configuration of the heuristic present inside each online policy to set the level of exploration (assigning CEs to different nodes) vs. exploitation (increasing the usage of a single node before switching to other nodes). In particular, we have three levels: Low, Medium, and High, corresponding to a threshold in the amount of available (up-to-date) data on a specific node before considering it viable during the scheduling procedure; otherwise, each policy will apply the `round-robin` one in favor of exploration. The general trend is that, for the selected workloads, the heuristic greediness has no noteworthy impact on the overall execution times. On the other hand, the impact of online scheduling policies is more relevant and correlated with the different characteristics of each workload. In MLE, both `min-transfer-size` and `min-transfer-time` policies can match the performance of user-defined scheduling approaches. On the other hand, CG sees a 2× slowdown when using online scheduling policies since the complex inter-dependencies (shown in Figure 5) are not known, and the run-time scheduler has limited information on the overall structure. It is relevant to notice that, although the online policies reach a slowdown in this workload compared to the best offline policy, **the workloads are still faster than a single-node execution**. Finally, MV highlights the need for UVM-aware policies. Indeed, the online policies are trying to minimize the data movements (in terms of `size` or `time`) by assigning CEs to the same node since moving from one node to another is an expensive operation. Nevertheless, the exponential growth of the execution time given by the oversubscription mechanism of UVM reaches levels where a pure exploration policy (like the simple `round-robin`) reduces its impact by at least 100×. This result considers our execution time cap for a single run of the workloads of 2.5 hours.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

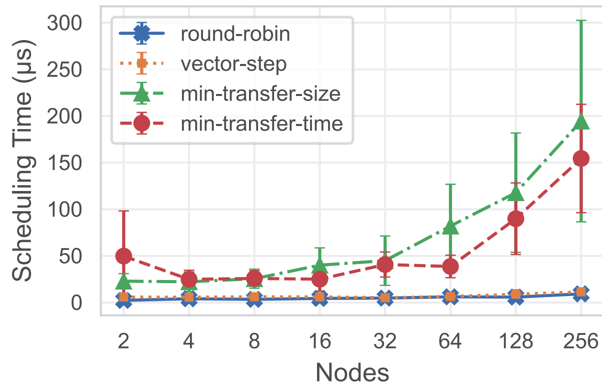


Fig. 9: Overhead of executing the scheduling policies to assign a `Worker` to CEs insides the `Controller` for an increasing number of nodes.

F. Strong Scaling Evaluation

- *Is infinite scale-out a definite solution?*

To answer this question, a relevant factor is how the user has defined its workload logic. Autonomously distributing user-provided workloads at runtime, with no hints provided by the user, is a difficult task to optimize. Nevertheless, we recognize that a heuristical model could be built to autonomously allocate more resources at runtime after reaching the steep increase in execution time that we highlighted for all the workloads in Sections V-C and V-D and also for the original pure CUDA C++ example in Figure 1. Indeed, there exists a direct link between execution time and oversubscription factor that might be exploited to set desired Key Performance Indicators (KPI) to be maintained during the workload execution, such as throughput or budget needs. This limit is bound to be reached at some point if the workload is put under more pressure to perform Big Data analytics tasks as the scale-up capabilities of **cloud GPU resources are limited to up to 16 GPUs inside of a single system**. After reaching this cap, the user will eventually encounter oversubscription and, therefore, a steep increase in execution times. Using GrOUT, the user can, almost effortlessly, distribute its application for rapid time-to-PoC of their solution.

- *Is the proposed runtime scheduler capable of handling large clusters?*

In Figure 9, we study the time required by the `Controller` to perform the scheduling decision and send the CEs to the `workers` while applying different policies on an increasing number of nodes. Static policies like the `round-robin` or `vector-step` are not influenced by the number of nodes to manage since they perform constant time operations. In particular, the former iterates over the nodes (`workers`), maintaining a circular list, while the latter switches between the nodes after assigning a pre-defined number of CEs. On the other hand, we have that, as expected, the two `min-transfer-[size/time]` policies increase the overhead of each assignment based on the number of nodes in the system. The hierarchical scheduling approach presented in Section IV-C relieves the `Controller` of the additional burden of assigning a specific Stream on each HW accelerator by delegating this choice to each `Worker`, effectively lowering the scheduling cost. Generally, the time required by each static policy is well under $30\mu s$, while for the more informed ones, it reaches a peak of $200\mu s$ with 256 nodes to manage. In particular, in our evaluation, we found that those overheads do not significantly impact the overall execution time of the workloads since they can be interleaved with `workers` placement of CEs into the HW-accelerators.

VI. Conclusion

In this paper, we have highlighted how UVM can have a significant impact on GPU-accelerated applications, and we have shown how scaling out can mitigate the problem effectively. Manually rewriting the workload logic to exploit common distribution paradigms can be time-consuming. Moreover, users in novel emerging domains might not have the expertise required to reach satisfactory performances. To face this problem, we designed a multi-language workload-agnostic framework that can autonomously distribute user-provided workloads on multiple multi-GPU nodes. We experimentally evaluated the GrOUT capabilities on a set of open-source workloads reaching speedups of up to $24.42\times$ based on the inner characteristic of each application. Moreover, we studied the behavior of the runtime scheduler under multiple offline and online scheduling policies. Although custom offline policies usually perform better, online ones are still faster than oversubscribed scenarios on a single node. Finally, we investigated the scaling capabilities of the high-level scheduler when working on larger clusters of nodes (up to 256 nodes). We observed a maximum of $200\,\mu s$ of overhead for each CE, even with the more informed online policies, a value acceptable throughout the different workloads. Additionally, the proposed framework can be adopted in all major programming languages thanks to its inner polyglot capabilities, simplifying the scale-out process of user-provided workload to multiple nodes with multiple HW accelerators. We wish to point out that although other vendors, such as AMD and Intel, provide their implementation of Unified Virtual Memory [28], [29], we focus specifically on NVIDIA's implementation in this work. Still, the methodology and framework can be easily extended to encompass other proprietary technologies.

Acknowledgements

The Authors would like to thank Oracle Cloud Infrastructure and the Oracle for Research program for the Oracle Cloud Credits that were essential for the creation of this work.

References

[1] M. Pandey, M. Fernandez, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, "The transformational role of gpu computing and deep learning in drug discovery," *Nature Machine Intelligence*, vol. 4, no. 3, pp. 211–221, 2022.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.

[2] A. Zeni, G. Guidi, M. Ellis, N. Ding, M. D. Santambrogio, S. Hofmeyr, A. Buluç, L. Oliker, and K. Yelick, "Logan: High-performance gpu-based x-drop long-read alignment," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 462–471.

[3] K. Ranganath, J. D. Suetterlein, J. B. Manzano, S. L. Song, and D. Wong, "Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[4] S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn, "Memory harvesting in {Multi-GPU} systems with hierarchical unified virtual memory," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 625–638.

[5] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014.

[6] J. Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, 2023.

[7] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 67–75.

[8] N. E. Karoui, M. Jeanblanc-Picquè, and S. E. Shreve, "Robustness of the black and scholes formula," *Mathematical finance*, vol. 8, no. 2, pp. 93–126, 1998.

[9] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in cuda unified memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019.

[10] "MPI Solutions for GPUs — developer.nvidia.com," https://developer.nvidia.com/mpi-solutions-gpus, [Accessed 19-10-2023].

[11] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, vol. 2, 2017.

[12] M. Šipek, B. Mihaljević, and A. Radovan, "Exploring aspects of polyglot high-performance virtual machine graalvm," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019.

[13] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "Towards polyglot adapters for the graalvm," in *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, 2019, pp. 1–3.

[14] A. Prokopec, G. Duboscq, D. Leopoldseder, and T. Wïrthinger, "An optimization-driven incremental inline substitution algorithm for just-in-time compilers," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 164–179.

[15] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: application initialization at build time," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[16] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck, "Cross-language interoperability in a multi-language runtime," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, no. 2, pp. 1–43, 2018.

[17] C. Wimmer and T. Würthinger, "Truffle: a self-optimizing runtime system," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 13–14.

[18] M. Knap and P. Czarnul, "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus," *The Journal of Supercomputing*, vol. 75, no. 11, pp. 7625–7645, 2019.

[19] T. Allen and R. Ge, "Demystifying gpu uvm cost with deep runtime and workload analysis," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 141–150.

[20] J. Lee, M. Samadi, and S. Mahlke, "Vast: The illusion of a large memory space for gpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 443–454.

[21] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 345–357.

[22] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1357–1370.

[23] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in cpu-gpu unified memory," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1212–1217.

[24] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon, and W. W. Ro, "Early-adaptor: An adaptive framework forproactive uvm memory management," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 248–258.

[25] X. Long, X. Gong, B. Zhang, and H. Zhou, "An intelligent framework for oversubscription management in cpu-gpu unified memory," *Journal of Grid Computing*, vol. 21, no. 1, p. 11, 2023.

[26] J. Fumero, F. Blanaru, A. Stratikopoulos, S. Dohrmann, S. Viswanathan, and C. Kotselidis, "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*, 2023, pp. 143–157.

[27] A. Parravicini, A. Delamare, M. Arnaboldi, and M. D. Santambrogio, "Dag-based scheduling with resource sharing for multi-task applications in a polyglot gpu runtime," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 111–120.

[28] Z. Jin and J. S. Vetter, "Evaluating unified memory performance in hip," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 562–568.

[29] "SYCL* Unified Shared Memory Code Walkthrough — intel.com," https://www.intel.com/content/www/us/en/developer/articles/code-sample/dpcpp-usm-code-sample.html, [Accessed 19-10-2023].

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on December 17,2024 at 17:29:34 UTC from IEEE Xplore. Restrictions apply.