UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Transparent Spatial Sharing of Multiple and Heterogeneous Accelerators

by

Emmanouil Pavlidakis

B.Sc., School of Engineering, Department of Information and Communication Systems Engineering, University of Aegean, Greece, 2012

M.Sc., Computer Science, Vrije Universiteit Amsterdam, Netherlands, 2016

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, January 2024

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Transparent Spatial Sharing of Multiple and Heterogeneous Accelerators

PhD Dissertation Presented

by **Emmanouil Pavlidakis**

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Emmanouil Pavlidakis

Supervisor: Angelos Bilas, Professor, University of Crete

Committee Member: Manolis G.H. Katevenis, Professor, University of Crete

Committee Member: Polyvios Pratikakis, Associate Professor, University of Crete

Committee Member: Vassilis D. Papaefstathiou, Assistant Professor, University of Crete

Committee Member: Giorgos Vasiliadis, Assistant Professor, Hellenic Mediterranean University

Committee Member: Leonidas Kosmidis, Computer Sciences, Barcelona Supercomputing Center, Spain

Committee Member: Vasileios Karakostas, Assistant Professor, National and Kapodistrian University of Athens

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, January 2024

Dedicated to my wife Elina, and my son Ioannis.

Acknowledgments

During this fantastic trip called PhD, I was fortunate to collaborate with amazing people. First of all, I am grateful to my supervisor Prof. Angelos Bilas. He gave me the space, the time, the resources, and most importantly the guidance to sharpen my technical skills as well as develop my research taste. He was constantly there to discuss my ideas and concerns, providing directions and invaluable advice on my research. His mentorship and his interdisciplinary approach were a source of inspiration during my PhD studies.

I am grateful to my thesis committee members Manolis G.H. Katevenis, Polyvios Pratikakis, Vassilis D. Papaefstathiou, Giorgos Vasiliadis, Leonidas Kosmidis, and Vasileios Karakostas for their feedback during my defense and for their comments that helped me prepare the final version of this thesis.

I want to thank every single co-author I had all these years: Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Nikos Chrysos. We worked together, we got rejected together, and we resubmitted together. I really enjoyed working with all of you.

After spending more than seven years in the Computer Architecture and VLSI Systems (CARV) lab of the Institute of Computer Science (ICS) in Foundation for Research and Technology-Hellas (FORTH), I feel like I have found a second family there. I met and worked alongside great people in this lab, and I want to express my deepest gratitude to Iacovos G. Kolokasis, Yannis Sfakianakis, Nikos Papakonstantinou, Anastasios Papagiannis, Giwrgos Saloustros, Giorgos Xanthakis, Eleni Kanellou, Theocharis Vavouris, Christos Kozanitis, and all other past and present members of CARV for preserving the balance between work and real life, making the lab a fun place to be.

I sincerely thank Manos Pigounakis, Giannis Silignakis, Giannis Marinos, and all other friends for their support, love, tolerance, and all the great moments we have shared.

I am grateful to my wife Elina Kiaoulia, for her patience, continuous support, and en-

couragement during the period of my doctoral thesis research. Additionally, I would like to express my gratitude to my newborn son Ioannis who joined us when I was writing my dissertation, for giving me unlimited happiness and pleasure.

Last but not least, I would like to express my deepest and most sincere gratitude to my parents, Ioannis and Maria, and my brother Vasilis for their love and support throughout all these years. Without your support I would have never been able to complete the dissertation.

I would also like to thank the Institute of Computer Science (ICS) in Foundation for Research and Technology-Hellas (FORTH), which supported me with graduate scholarships throughout my doctoral studies. Funding comes from several European projects that include: Vineyard (GA 687628), EVOLVE (GA 825061), EUPILOT (GA 101034126), DEEP-SEA (GA 955606), and HiPEAC (GA 871174).

Abstract

Today, effectively utilizing multiple heterogeneous accelerators within applications and high-level Machine Learning (ML) frameworks like TensorFlow, PyTorch, and Caffe presents notable challenges across four key aspects: (a) sharing heterogeneous accelerators, (b) allocating available resources elastically during application execution, (c) providing the required performance for latency critical tasks, and (d) protecting application's data under spatial sharing.

In this dissertation, we introduce a novel runtime system designed to decouple applications from the intricacies of heterogeneous accelerators within a single server. Our approach entails a client-side API that allows applications to be written once without considering any low-level details, such as the number or type of accelerators. By leveraging our system, applications are liberated from the burdens of accelerator selection, memory allocations, and memory management operations. A backend service seamlessly manages these intricate tasks—referred to as the server—which is shared among all applications and boasts four primary features.

First, the server defers the assignment of a task to an accelerator until the latest feasible moment, setting it apart from current methods that allocate an application to an accelerator during its initialization phase. Subsequent to the task assignment decision but just prior to task execution, the server promptly transfers the necessary data to the designated accelerator. This dynamic task assignment and the lazy data placement enable adaptation to application load changes.

Second, to ensure that latency-critical GPU applications will have the desired performance under time-sharing, the server revokes the execution of long-running kernels. Our revocation mechanism stops a task by prematurely terminating the ongoing GPU kernel without preserving any state and replays it later. The server uses a runtime scheduler that prioritizes latency-critical tasks over batch and instructs the revocation mechanism when to kill a running kernel.

Third, to facilitate spatial accelerator sharing across applications, the server establishes multiple streams for GPUs and command queues for FPGAs. Regarding FPGAs, the server loads multi-kernel bitstreams and can (re)program the FPGA with the appropriate bitstream required from each application task. While spatial accelerator sharing enhances accelerator utilization and application response time compared to time-sharing, it does come at the expense of data isolation.

Finally, GPU spatial sharing lacks protection due to the single accelerator address space, leaving application data susceptible to exposure to other applications. Consequently, the feasibility of sharing in broad multi-user settings becomes compromised. To resolve this issue, we design and implement a software-based sandboxing approach that applies bitwise instructions in the virtual assembly code of kernels. Our approach does not require extra or specific hardware units and supports ML frameworks that use closed-source domain-specific libraries.

To minimize the porting effort of existing CUDA applications, we examine the interception of CUDA API calls at various levels, i.e., driver, runtime, and high-level library functions. We show that intercepting only the CUDA runtime and driver library is adequate to run complex ML frameworks, such as Caffe and PyTorch. Additionally, this level of interception is more robust than the ones used from previous approaches because it requires handling fewer and much simpler functions.

We use Caffe, TensorFlow, PyTorch, and Rodinia to demonstrate and evaluate the proposed runtime system in an accelerator-rich server environment using GPUs, FPGAs, and CPUs. Our results show that applications that use our system can safely share accelerators without any modifications at low overhead and with latency guarantees.

Keywords: Runtime System, Accelerators, Heterogeneity, Spatial Sharing, GPU Memory Protection, Preemption, Scheduling

Supervisor: Angelos Bilas
Professor

Computer Science Department

University of Crete

Περίληψη

Σήμερα, η αποτελεσματική χρήση πολλαπλών ετερογενών επιταχυντών σε εφαρμογές αλλά και σε δομές μηχανικής μάθησης (Machine Learning Frameworks) όπως το TensorFlow, το PyTorch και το Caffe παρουσιάζει τέσσερις βασικές προκλήσεις: (α) Την κοινή χρήση ετερογενών επιταχυντών, (β) την ελαστική κατανομή των διαθέσιμων πόρων κατά την διάρκεια εκτέλεσης των εφαρμογών, (γ) την εξασφάλιση της απαιτούμενης απόδοσης σε εφαρμογές που η χρονική καθυστέρηση είναι σημαντική και (δ) την προστασία των δεδομένων των εφαρμογών που διαμοιράζονται ένα επιταχυντή.

Σε αυτή τη διατριβή, εισάγουμε ένα νέο σύστημα χρόνου εκτέλεσης που έχει σχεδιαστεί για να αποσυνδέει τις εφαρμογές από τις περιπλοκές διαδικασίες που απαιτούνται για την χρήση ετερογενών επιταχυντών. Η προσέγγισή μας περιλαμβάνει μια διεπαφή προγραμματισμού εφαρμογών (application programming interface) που χρησιμοποιείται από τις εφαρμογές και έτσι τους επιτρέπει να γράφονται μία φορά χωρίς να λαμβάνονται υπόψη λεπτομέρειες όπως ο αριθμός ή ο τύπος των επιταχυντών. Με τη χρήση του συστήματός μας, οι εφαρμογές απελευθερώνονται από την επιβάρυνση της επιλογής επιταχυντή, της δέσμευσης μνήμης και της διαχείρισης μνήμης. Όλες αυτές οι περίπλοκες διεργασίες διεκπεραιώνονται από μια υπηρεσία υποστήριξης – που αναφέρεται ως διακομιστής (server)– η οποία είναι κοινή και την διαμοιράζονται όλες οι εφαρμογές που εκτελούνται σε ένα κόμβο. Ο διακομιστής έχει τέσσερα βασικά χαρακτηριστικά.

Πρώτον, η ανάθεση μιας διεργασίας σε ένα επιταχυντή πραγματοποιείται την τελευταία στιγμή και όχι κατά την αρχικοποίηση της εφαρμογής όπως συμβαίνει με τις υπάρχουσες μεθόδους. Μετά την απόφαση ανάθεσης της διεργασίας και ακριβώς πριν από την εκτέλεση αυτής, ο διακομιστής μεταφέρει τα απαραίτητα δεδομένα στον επιλεγμένο επιταχυντή. Αυτή η δυναμική ανάθεση εργασιών και η καθυστερημένη τοποθέτηση δεδομένων επιτρέπουν την προσαρμογή στις αλλαγές φόρτου εφαρμογής.

Δεύτερον, για να διασφαλιστεί ο χρόνος απόκρισης σε συγκριμένες εφαρμογές όταν αυτές

διαμοιράζονται χρονικά μια κάρτα γραφικών, με άλλες που έχουν πυρήνες (kernels) που ο χρόνος εκτέλεσης τους είναι πολύ μεγάλος, ο διακομιστής μπορεί να σταματήσει την εκτέλεση αυτών των μεγάλων πυρήνων χρησιμοποιώντας ένα μηχανισμό ανάκλησης (revocation). Ο μηχανισμός ανάκλησής σταματά μια διεργασία τερματίζοντας πρόωρα τον πυρήνα που βρίσκεται σε εξέλιξη χωρίς να αποθηκεύει τα δεδομένα που χρησιμοποιεί και τον ξανά ξεκινάει αργότερα. Ο διακομιστής χρησιμοποιεί έναν προγραμματιστή χρόνου εκτέλεσης (scheduler) που δίνει προτεραιότητα σε κρίσιμες εφαρμογές έναντι άλλων χωρίς αυστηρές χρονικές απαιτήσεις και καθοδηγεί τον μηχανισμό ανάκλησης πότε πρέπει να σταματήσει ένα πυρήνα που εκτελείται.

Τρίτον, για να υποστηρίξει την χωρική διαμοίραση επιταχυντών μεταξύ εφαρμογών, ο διακομιστής δημιουργεί πολλαπλές ουρές εντολών σε κάθε επιταχυντή. Όσον αφορά τις FPGA, ο διακομιστής φορτώνει κυκλώματα (bit-streams) πολλαπλών πυρήνων και μπορεί να (επανα)προγραμματίσει την FPGA με το κατάλληλο bit-stream που απαιτείται για κάθε διεργασία. Ο χωρικός διαμοιρασμός επιταχυντών αυξάνει τη χρήση των πόρων του επιταχυντή και βελτιώνει τον χρόνο απόκρισης των εφαρμογών σε σχέση με τον χρονικό διαμοιρασμό, όμως εις βάρος της προστασίας των δεδομένων.

Η δυνατότητα που έχει μια εφαρμογή να διαβάσει και να γράψει τα δεδομένα μιας άλλης όταν χρησιμοποιούν ταυτόχρονα την ίδια κάρτα γραφικών κάνει τον χωρικό διαμοιρασμό αυτου του τύπου τον επιταχυντών σε περιβάλλοντα σύννεφου (cloud environments) που υπάρχουν πολλοί χρήστες να είναι απαγορευτική. Για να επιλύσουμε αυτό το ζήτημα, σχεδιάσαμε και εφαρμόσαμε μια τεχνική που εφαρμόζεται σε εικονική γλώσσα μηχανής (virtual assembly), δεν χρειάζεται παραπάνω ή ειδικές μονάδες υλικού (hardware units) και τέλος υποστηρίζει ML frameworks που χρησιμοποιούν κλειστές βιβλιοθήκες.

Για να ελαχιστοποιήσουμε την προσπάθεια μεταφοράς των υπαρχουσών CUDA εφαρμογών στην δικιά μας διεπαφή προγραμματισμού (API), εξετάζουμε την υποκλοπή κλήσεων του CUDA API σε διάφορα επίπεδα, δηλαδή το CUDA runtime, το CUDA driver, και κλήσεις σε υψηλού επιπέδου βιβλιοθήκες. Σε αυτήν την διατριβή δείξαμε ότι αν υποκλέψουμε μόνο τις CUDA runtime και CUDA driver βιβλιοθήκες είναι αρκετό για να τρέξουμε περίπλοκλα Machine Learning Frameworks. Επιπλέον αυτού του είδους η προσέγγισή είναι πιο αποδοτική σε σχέση με προηγούμενες διότι απαιτεί την διαχείριση λιγότερων κλήσεων.

Για να αξιολογήσουμε το σύστημα μας χρησιμοποιούμε πραγματικές εφαρμογές όπως τα

Caffe, TensorFlow, PyTorch και Rodinia. Επιπλέον χρησιμοποιήσαμε πολλαπλούς και διαφορετικούς τύπου επιταχυντές όπως GPUs, FPGAs, και CPUs. Τα αποτελέσματά μας δείχνουν ότι οι εφαρμογές που χρησιμοποιούν το σύστημά μας μπορούν με ασφάλεια να μοιράζονται πολλούς και διαφορετικού τύπου επιταχυντές χωρίς καμία τροποποίηση, με χαμηλό κόστος και με εγγυήσεις καθυστέρησης.

Λέξεις κλειδιά: Σύστημα Χρόνου Εκτέλεσης, Επιταχυντές, Ετερογένεια, Χωρικός διαμοιρασμός, Προστασία Μνήμης Επιταχυντών, Ανάκληση Εργασιών, Χρόνο-προγραμματισμός Διεργασιών

Επόπτης: Άγγελος Μπίλας
Καθηγητής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Bibliographic Notes

The publications related to this dissertation (ordered by date) are:

(i) Stelios Mavridis, **Manos Pavlidaki**s, Ioannis Stamoulias, Christos Kozanitis, Nikos Chrysos, Christoforos Kachris, Dimitrios Soudris, and Angelos Bilas. 2017. *VineTalk: Simplifying software access and sharing of FPGAs in datacenters.* In Proceedings of the 27th International Conference on Field Programmable Logic and Applications **(FPL '17)**.

(ii) **Manos Pavlidakis**, Stelios Mavridis, Nikos Chrysos, and Angelos Bilas. 2020. *TReM: A Task Revocation Mechanism for GPUs.* In Proceedings of the 22th IEEE International Conference on High Performance Computing and Communications **(HPCC '20)**.

(iii) **Manos Pavlidakis**, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. *Arax: A Runtime Framework for Decoupling Applications from Heterogeneous Accelerators.* In Proceedings of the 13th ACM Symposium on Cloud Computing **(SoCC '22)**.

(iv) **Manos Pavlidakis**, Giorgos Vasiliadis, Stelios Mavridis, Antony Chazapis, and Angelos Bilas. *Guardian: Data Isolation for Multi-Tenant GPU Sharing.* **(Under submission)**.

More specifically, Chapter 2 based on *(iii)*. Chapter 3 is based on *(i)*. Chapter 4 based on *(ii)*, and Chapter 5 on *(iv)*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing need for high performance at low energy consumption has resulted in the proliferation of heterogeneous accelerators, such as GPUs, FPGAs, and TPUs [29, 26, 5, 96, 99, 85]. Recent estimates [85, 5, 11, 100, 113] indicate that servers will include a plethora of processing units and specialized accelerators [15, 19, 58, 92]. This trend poses significant challenges in how applications and higher-level frameworks, such as TensorFlow [1], PyTorch [78], and Caffe [41], can fully utilize the capacity of heterogeneous accelerators.

Today, a large percentage of applications or frameworks is *statically bound to specific accelerators throughout their execution*. In particular, many applications are directly written for one accelerator type, e.g., NVIDIA GPUs, to allow for device-specific optimizations. Over the last years, unified programming models, e.g., SYCL [84] and oneAPI [3], aim to offer portability to different accelerator types. However, applications are still required to explicitly select the desired accelerators during initialization and prior to starting their execution. As a result, each application execution is statically bound to a specific set of accelerators or types that cannot change at runtime. This results in poor resource and application efficiency in two ways: (a) reduced sharing of resources and (b) lack of adaptation over time.

Existing resource assignment techniques fully allocate accelerators to a single application. Although practical, this exclusive assignment creates significant *load imbalance* in heterogeneous setups with multiple accelerators and results in resource under-utilization. Time-sharing approaches [108, 109, 113, 35, 63] cannot address this issue effectively because accelerators become “beefier” and individual applications and, even more so, individual kernels often fail to fully utilize all the available accelerator resources [113, 20, 69, 54, 8, 74, 115]. To address these limitations, several research approaches propose spatial sharing mechanisms [110, 20, 115, 31, 104, 112, 103, 57]. However, existing approaches are limited to specific accelerator types and require applications to perform manual task assignment and data placement. Regarding GPUs, NVIDIA offers Multiple Process Service (MPS), which allows different CUDA applications to execute concurrently in the same GPU. MPS offers a single GPU context, i.e., a common accelerator address space, that is a requirement for NVIDIA GPUs to allow kernels from different applications (and users) to execute concurrently. AMD GPUs offer by default this shared context, avoiding the need for such a service. However, this single/shared context introduces a significant concern: GPU kernels execute in the same GPU address space, hence they can modify (either inadvertently or deliberately) memory locations that belong to other applications [21, 76, 81, 65, 53, 51, 8] leading to confidentiality and integrity violations. This lack of memory protection makes spatial sharing impractical for generic multi-user environments.

Applications often exhibit dynamic behavior and fluctuating load requirements [35, 108]. Current approaches assign resources to each application that remain fixed throughout its execution. Assigning resources statically to applications is challenging due to the difficulty in accurately estimating their resource demands prior to execution. The lack of *dynamic task assignment* and *data-migration* results in application under- or over-provisioning and eventually to poor resource utilization. Existing approaches [35, 108] can dynamically assign whole *applications* to NVIDIA GPUs –not individual tasks– while their migration mechanisms rely either on domain-specific application features (TensorFlow checkpoints) or vendor-specific accelerator mechanisms (unified memory). Apart from application load, GPU workloads also have a dynamic behavior. Consequently, a GPU preemption mechanism is required to provide low latency in high-priority tasks in the presence of long-running batch tasks. A preemption mechanism consists of three parts; (1) stop the currently executing task, (2) save its state, and (3) replay the task later. Previous works [106, 68, 116] do not provide a bounded latency due to their “task stop” and state-saving mechanisms. Therefore, high-priority tasks still incur high tail latencies.

This dissertation proposes a runtime system that decouples applications from heterogeneous accelerators within a single server. Our approach is based on RPC, a mechanism that is proven to be very successful in decoupling complex software stacks, using clear and conceptually simple boundaries. We offer a client-side API that allows applications to be written once without considering any low-level details, such as the number or type of accelerators. Applications that use our system do not need to perform accelerator selection, memory allocations, or memory management operations. All these operations are handled transparently by a backend service, the server. The server is common and shared from all applications and offers four main features.

(1) Our server assigns a task to an accelerator (NVIDIA GPU, AMD GPU, FPGA) or a CPU as late as possible, differently from existing approaches that assign an *application* to an accelerator during its initialization (Arax § 2). After the task assignment and just before the task execution, the server transfers the required data to the selected accelerator. This dynamic task assignment and the lazy data placement enable adaptation to application load changes.

(2) To ensure that latency-critical applications will have the desired performance when they time-share a GPU, the server revokes the execution of long-running kernels. Our revocation mechanism stops a GPU task by aborting its currently executing GPU kernel without saving any state and replays it later (TReM § 4). The server uses a runtime scheduler that prioritizes latency-critical tasks over batch and instructs the revocation mechanism when to kill a running kernel.

(3) To enable spatial accelerator sharing across applications, the server creates multiple streams for GPUs and command queues for FPGAs (Arax § 2). Regarding FPGAs, it loads multi-kernel bitstreams and can (re)program the FPGA with the appropriate bitstream required from each application task (VineTalk § 3). Spatial accelerator sharing improves accelerator utilization and application turnaround time.

(4) Spatial sharing offered by our system and also from previous works [110, 20, 115, 31, 104, 112] does not offer protection for applications that share a GPU. Consequently, application data are exposed to other applications, making sharing impractical for generic

multi-user environments. To resolve this issue, we design and implement a PTX-based sandboxing approach to isolate data for applications that spatially share a GPU (Guardian § 5).

(5) Application porting to our *agnostic* client-side API requires manual effort, which is prohibitive in complex frameworks. To reduce the porting effort, we design and implement an automatic stub generator in Arax (§ 2) that ports a CUDA application to our client API. We focus on CUDA since NVIDIA GPUs dominate the accelerator market, and thus, complex frameworks such as Caffe, PyTorch, and TensorFlow can run to heterogeneous accelerators. The only requirement that our tool has is the existence of the kernels for the different accelerator types. Our approach intercepts the CUDA runtime (e.g., `cudaMalloc(), cudaLaunchKernel()`), the CUDA driver (e.g., `cuMemAlloc(), cuMemCpy()`), and high-level calls to CUDA accelerated libraries (e.g., `cublasIsamax(), cudnnActivation-Forward()`). Due to the complexity of this approach due to the interception of more than 1600 high-level calls that are complex and change rapidly in Guardian (§ 5) we move our interception one level deeper and intercept only the CUDA runtime and driver libraries. Although challenging because we need to handle an undocumented call (i.e., `cudaGetExportTable()`), it is proved to be more robust.

Thesis statement: Provide transparent and efficient sharing of heterogeneous accelerators for real-world applications in a server.

Contributions

The specific contributions of this dissertation are:

1. We remove static application to accelerator assignment that affects utilization using a generic client-side API and a backend service. The client-side API abstracts accelerator type and number, whereas the service manages multiple and heterogeneous accelerators in a server.

2. To reduce the programming effort required to port existing applications to our client-side API, we investigate the interception of CUDA API calls at various levels, i.e., driver, runtime, and high-level library functions calls. We concluded that the ap-

propriate level of interception is the CUDA runtime API.

3. Our backend service improves accelerator utilization and adapts to the fluctuating load requirements using six main mechanisms that can be applied to different accelerator types (i.e., NVIDIA GPUs, AMD GPUs, and FPGAs): spatial sharing, elasticity, dynamic task assignment, lazy data placement, live-migration, and task revocation.

4. Spatial GPU sharing imposes confidentiality and integrity violations. As a result, we examine the memory protection of kernels running concurrently on NVIDIA GPUs and provide a transparent memory protection mechanism. Our mechanism applies address fencing instructions before every load and store in the PTX-level of CUDA kernels.

Organization

The rest of this dissertation is organized as follows. Chapter 2 presents Arax the runtime used to decouple ML applications from heterogeneous accelerators. Chapter 3 explains in detail VineTalk, which includes the mechanisms that simplify the use of FPGAs and FPGA sharing. Chapter 4 shows the kernel revocation mechanism and the scheduling policies required to ensure the Service Level Agreement (SLA) for latency-critical applications when they time-share a GPU, named TReM. Chapter 5 presents Guardian a mechanism that enables protected GPU spatial sharing. Chapter 6 reviews related work and compares our approach with existing systems. Chapter 7 outlines prospective directions for future work, and chapter 8 concludes this thesis.

Chapter 2

Decouple Applications and Accelerators

This chapter shows how to avoid static application to accelerator assignment that leads to underutilization. To achieve that, we propose an RPC-based approach that decouples applications from heterogeneous accelerators *within* a single server, named Arax. The client-side stubs of Arax allow applications to be written once using a simple API without considering any low-level details, such as the number or type of accelerators. Additionally, Arax applications do not need to perform accelerator selection, memory allocation, or task assignment operations; all are handled transparently by a backend service, the Arax server. Our server assigns applications tasks dynamically (not at application initialization) and performs memory allocations-transfers lazily and only after a task is assigned to a specific accelerator. To improve accelerator utilization while ensuring application performance Arax provides three capabilities:

(a) **Spatial sharing** that manages existing mechanisms in heterogeneous accelerators, transparently, and across all applications in a server. We use asynchronous host-threads to issue tasks to GPU streams and FPGA command queues. Regarding FPGAs, Arax loads bitstreams with multiple kernels that need to be collocated in the same FPGA. The advantage of our approach is that it moves all the related management from individual applications to the shared Arax runtime and can make decisions across all applications.

(b) **Elasticity and dynamic resource assignment** to applications at runtime. To achieve this, Arax requires fine-grain access to application tasks and their data. Arax uses asynchronous operations to issue independent tasks across different accelerators, while ensuring that tasks with dependencies execute in-order.

(c) **Live-migration** that moves application tasks across heterogeneous accelerators. Unlike existing approaches, our migration mechanism does not require application modifications or specialized accelerator support. Arax uses task arguments to keep track of the data used by each task and transfers only relevant data upon task migration. Although arbitrary pointers may result in moving large amounts of memory, our approach is adequate to support real applications, such as TensorFlow and Caffe.

Finally, Arax includes **Autotalk**, a generator that creates stubs for a given accelerator API based on a description of the target API provided by the user once. Applications are then linked dynamically with the stub library that internally calls the Arax API. Currently, Autotalk generates stubs for a subset of CUDA that can support Caffe and TensorFlow.

We evaluate Arax using Caffe, TensorFlow, and Rodinia. Our results show that Arax applications can run without any modifications at low overhead—up to 12% compared to native—when other approaches, i.e., AvA [113], result in up to 30% overhead for the same applications running on a GPU. In addition, Arax provides elasticity, decreasing total application turnaround time by 2× compared to native execution without elasticity support. Our migration mechanism adds 7% overhead compared to standalone execution. Finally, our sharing mechanism provides up to 20% improvement in total execution time compared to NVIDIA MPS.

The specific contributions of this chapter are:

1. We propose an RPC-based approach to decouple applications from heterogeneous accelerators within servers.

2. We present a mechanism for spatial sharing of heterogeneous accelerators and dynamic and transparent assignment of tasks to accelerators.

3. We present an application live-migration mechanism that reduces data movement based on data ownership by tasks.

4. We present a stub generator that allows existing applications to use Arax with minimal effort and demonstrate our approach with Caffe and TensorFlow.

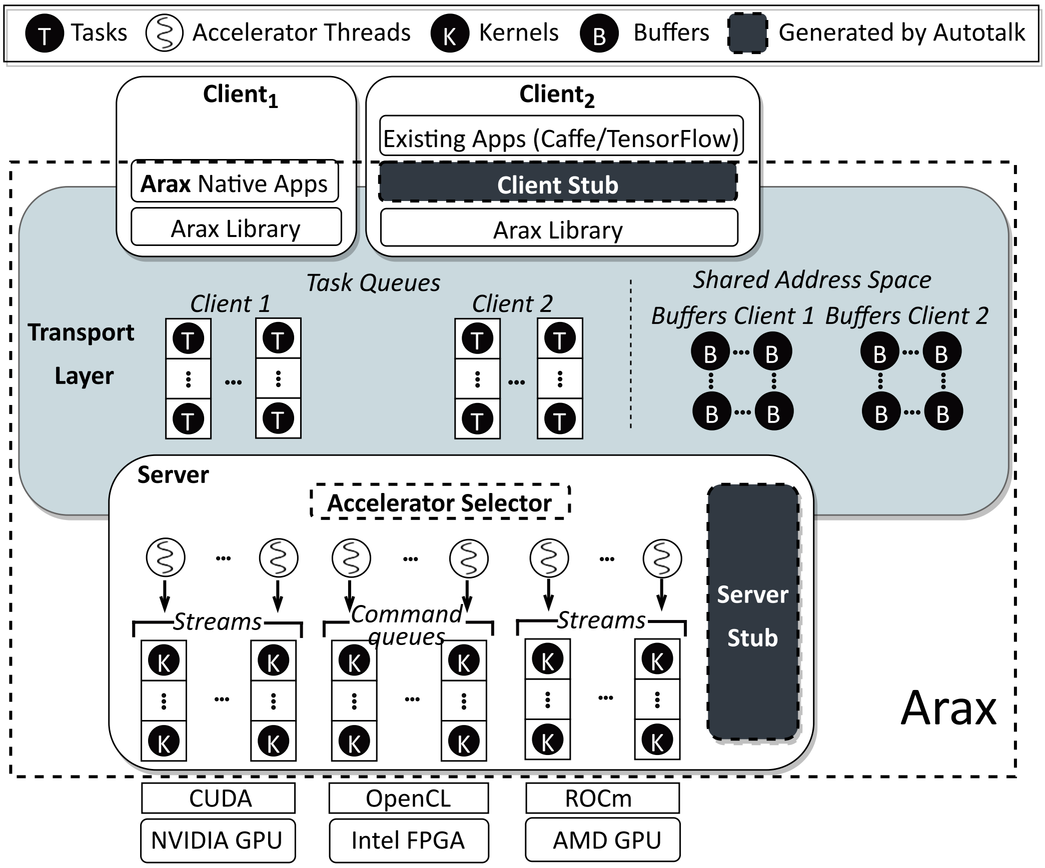


Figure 2.1: Arax high-level overview. The main components of Arax are: Clients, Server, Transport layer, and Autotalk.

5. We demonstrate and evaluate Arax in an accelerator-rich server environment, using GPUs, FPGAs, and CPUs, with Caffe, TensorFlow, and Rodinia.

2.1 Design

Figure 2.1 shows a high-level overview of Arax. Applications use the Arax API to access available accelerators, regardless of their types. Applications create task queues and issue tasks, providing their data in the form of Arax buffers. Tasks and buffers are being transported to the Arax server via a transport layer over shared memory, mapped to both the application and server address spaces. The Arax server assigns dynamically and asynchronously application tasks to accelerators, managing accelerator streams and command queues, maintaining task ordering, and handling data dependencies. Finally, Arax's stub generator, Autotalk, allows generating the stub library automatically for a particular accelerator API, given a description file of the API calls. Next, we discuss each component of Arax in more detail.

2.1.1 Client

Arax provides three basic abstractions to hide accelerator types and number from applications: (a) *tasks*, (b) *task buffers*, and (c) *task queues*. Table 2.1 shows an overview of the main Arax API calls.

Tasks: A task can be either a compute or a transfer task and is used to hide accelerator-specific information. A compute task is an accelerator kernel, while a transfer task is a data transfer between the host and the accelerator and vice versa. Both tasks are executed without interruption and are asynchronous. Arax provides synchronization primitives to allow applications to wait for their completion. A compute task takes the kernel name and its corresponding arguments as parameters, i.e., inputs, outputs, and arguments required from a kernel. The kernel name is associated with the actual kernel at the server (§2.1.2). Unlike existing accelerator APIs, task arguments do not include accelerator-specific information, such as thread number or thread size. The parameters for a transfer task include the task buffers provided by Arax and any data from the application address space.

Task buffers: A buffer is an opaque identifier that represents the input and output data of a task and is used to hide accelerator memory. Multiple tasks or applications can operate on the same buffer concurrently. It is important to note that Arax decouples the accelerator memory management from applications using a lazy memory allocation strategy. When an application requests memory, Arax stores the requested allocation size but does not allocate this memory on the accelerator (§2.1.2). The actual allocation will be performed only after the task is successfully assigned to an accelerator. In the meantime, applications can continue issuing tasks since buffers are implemented as opaque types in the shared memory. For all allocations in the shared memory, we use the Doug Lea allocator. This abstraction hides accelerator memory, and applications are unaware of which

Abstraction	API call	Description
Tasks	a_issue()	Issue a task
	a_wait()	Wait for a task
Buffers	a_allocate()	Allocate Buffer
	a_free()	Free Buffer
	a_sync_to(), a_sync_from()	Transfer Data
Task Queues	a_acquire()	Acquire a virtual accelerator
	a_release()	Release a virtual accelerator

Table 2.1: Methods of Arax API.

accelerator hosts their data.

Task queues: Applications issue tasks to task queues, similar to existing programming models, e.g., CUDA/ROCm streams and OpenCL command queues. The main difference of Arax is that these queues are not assigned directly to an accelerator. Instead, Arax is responsible for assigning them to one or more accelerators at runtime (§2.1.2), while ensuring that asynchronous tasks will be executed in-order. Each task queue holds tasks with dependencies. To denote independent sets of work, applications need to acquire different task queues. This approach works well for the ML frameworks we examine due to the inherent serialization of NN layers.

2.1.2 Server

The Arax server is responsible for maintaining task issue order and managing data dependencies while performing dynamic task assignment and data placement to accelerators. These mechanisms allow Arax to provide efficient spatial sharing and elastic allocation of resources.

Spatial Sharing: The spatial sharing mechanism of Arax is based on streams/command queues and host-threads (Arax accelerator threads). In particular, to execute kernels in parallel, the server spawns multiple threads per physical accelerator. Each accelerator thread internally creates different streams (CUDA and ROCm) or command queues (OpenCL). The design of spatial sharing in Arax can support advanced task assignment policies that do not rely on low-level accelerator-specific APIs. To enable spatial sharing for NVIDIA GPUs, we require a single context; thus, the Arax server is implemented as

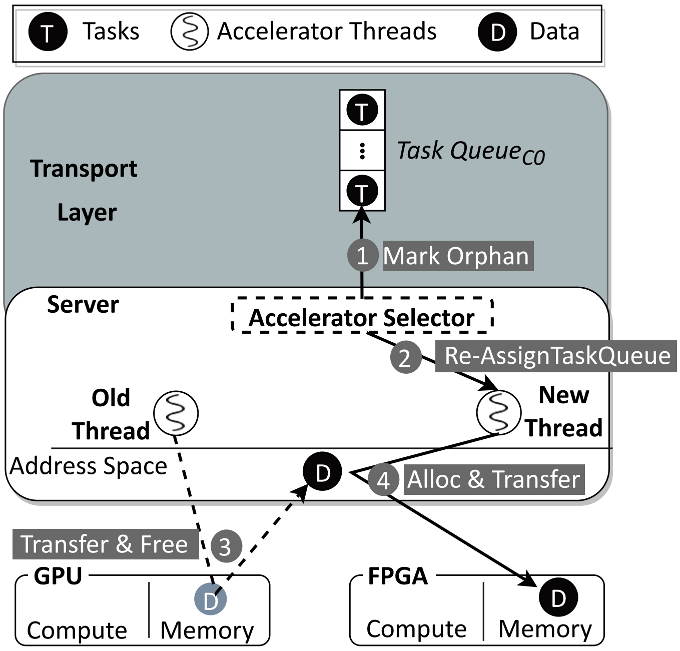


Figure 2.2: The steps required for an application migration. The task queue is marked orphan (1) and reassigned to a new thread (2). The relevant data are then transferred to the new accelerator via the server memory (3,4).

a single process for all accelerators. Regarding FPGAs, the Arax server loads a bitstream that contains multiple kernels, similar to Vinetalk [63]. The server can select and load the appropriate bitstream to serve each task.

Application migration: Even when accelerators are shared, there can be load imbalances. Arax offers an application migration mechanism to correct load imbalances. This migration mechanism can move application tasks and their data across heterogeneous accelerators. The migration mechanism cannot stop a task during execution. Instead, it waits for the task to finish and moves any pending tasks and their data to another accelerator. There are *three challenges* that our migration mechanism needs to tackle:

(i) Migrate an application without interrupting its execution. Arax offers task queues to applications to issue their tasks. The Arax server stops and resumes the execution of a task queue, and thus it does not affect the execution of the application. In particular, Arax performs the following steps: (a) The server marks this task queue as an orphan (Figure 2.2; step ➊). At this point, accelerator threads cannot launch tasks from this task queue. (b)

Since then, there could have been tasks issued for execution; the server waits for them to finish before re-assigning this task queue to a different accelerator thread (Figure 2.2; step ❷). (c) From here on, any remaining task from this particular task queue will be invoked to the new accelerator. We note that, during the migration, the application continues issuing tasks to its task queues.

(ii) Move only the data of the migrated task. The server should move only the data required from the migrated task and not all the application state. Existing checkpoint approaches [108, 16] migrate all the application state, which involves transfers in the range of gigabytes. The Arax server maintains metadata for each task and is aware of the data required. After assigning the task queue to a new accelerator thread, the server instructs the previous accelerator thread to copy the task data from its accelerator memory to the server memory and free the corresponding allocations (Figure 2.2; step ❸). The server then notifies the new accelerator thread to allocate and copy that data from the server's memory (Figure 2.2; step ❹) using the native accelerator API. We note that the server memory is an intermediate buffer to transfer data across different accelerators. As part of our future work, we plan to eliminate this extra copy using accelerator-to-accelerator transfers, at least for the cases supported [86].

(iii) Migrate the most recent version of the data. Before a data migration, we must ensure that the data required from the migrated task(s) are up-to-date. To achieve that, the server allows only one valid copy of the data (at any given time) to the distinct accelerator memories in multi-accelerator setups.

Dynamic task assignment: The server assigns the incoming task queues to the underlying accelerators. Individual tasks from the same task queue can be assigned to different accelerators. This assignment involves task and data migrations for tasks with dependencies. When the server detects an unassigned, non-empty task queue, it assigns it to an accelerator using a round-robin policy (default). Advanced assignment policies can be implemented with relatively low effort. This is facilitated by the fact that Arax already collects information regarding the memory footprint of each task, the number of tasks per accelerator, and the data ownership.

As a proof of concept that our accelerator selector can host advanced assignment policies, we also implement an elastic assignment policy. This policy is essential to handle load fluctuations or data bursts by performing dynamic task assignments. The server keeps track of the assigned task queues per accelerator and knows the owner of each task queue. Consequently, the accelerator selector can increase/decrease the accelerators assigned to an application based on the load.

For instance, lets assume that we have a low-priority application with two task queues, i.e., *task queue1* and *task queue2*. Initially, both task queues are assigned to the same accelerator. When the accelerator selector detects idle accelerators, it expands the resources used by the low-priority application by assigning *task queue2* to the idle accelerator. Reversely, when another high-priority application arrives, the server shrinks the accelerators used from the low-priority application by moving *task queue2* to the accelerator where *task queue1* executes. This re-assignment requires moving the application state between accelerators, i.e., application migration. Consequently, the high-priority application can make exclusive use of the idle accelerator.

To perform memory management, the server maintains internally a mapping of the allocated buffers per task queue and their corresponding sizes. We note that the actual memory allocation is performed only after its corresponding task queue has been assigned to a physical accelerator (Figure 2.3; step ❶). After the selection of the physical accelerator, the thread of that accelerator gets a task from the task queue (Figure 2.3; step ❷) and checks if any memory has already been allocated in that particular accelerator memory. If not, it performs the actual allocation (Figure 2.3; step ❸) and keeps a reference to that memory segment so that it can be used for deallocation purposes. After that, the accelerator thread can issue the task to the accelerator. If the task is a data transfer, the accelerator thread copies the data from the client address space to the accelerator memory (Figure 2.3; step ❹).

To support different accelerator types, the server spawns separate accelerator threads. Each thread uses the accelerator's native API to communicate with that particular accelerator. Currently, Arax supports NVIDIA GPUs using CUDA, Intel Altera FPGAs using OpenCL, and AMD GPUs using ROCm. When receiving a compute task, the accelerator thread uses the kernel name—passed as a task parameter—to find the appropriate kernel

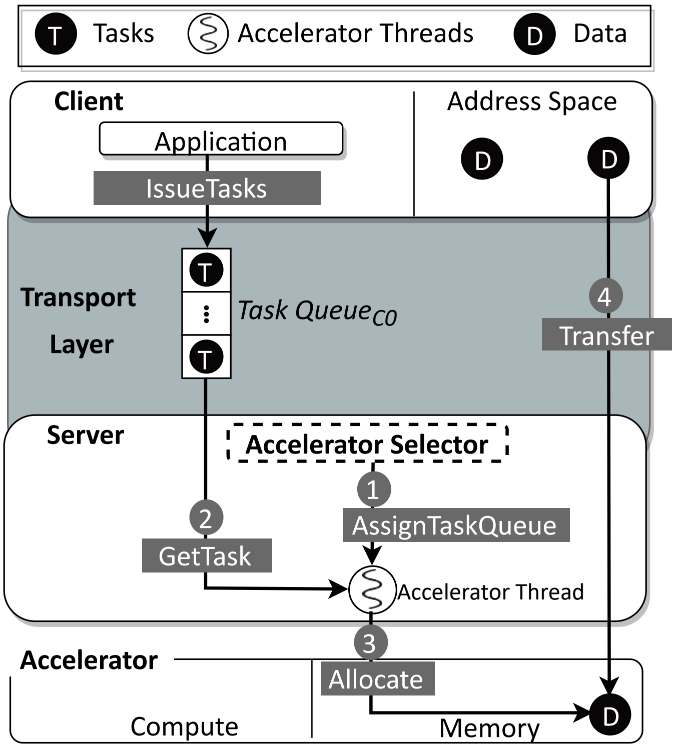


Figure 2.3: Arax dynamic task assignment. Application issues tasks to a task queue. Initially, the task queue is assigned to an accelerator (1), then the accelerator thread gets a task (2). It allocates accelerator memory for that data (3) and copies the data from the application (4).

program and loads it to the physical accelerator for execution. For this reason, the server maintains a dispatch table that associates kernel names with the actual kernel programs in the server stub.

We assume that kernels are implemented by third-party experts using the native accelerator's API. Accelerators offer libraries such as `RAND` (Random Number Generation) and `BLAS` (Basic Linear Algebra Subroutine). The function calls in these libraries can involve multiple kernel invocations internally, which cannot be extracted in case the library is closed-source (e.g., NVIDIA `cuBLAS` and `cuRAND`). To overcome this limitation, we incorporate these libraries into Arax, as-is, forming different server stubs, one for each accelerator. The server stubs are compiled using the accelerator-specific compilers. For NVIDIA GPUs we use NVCC, for Intel FPGAs we use AOCL, and for AMD GPUs we use HIPCC.

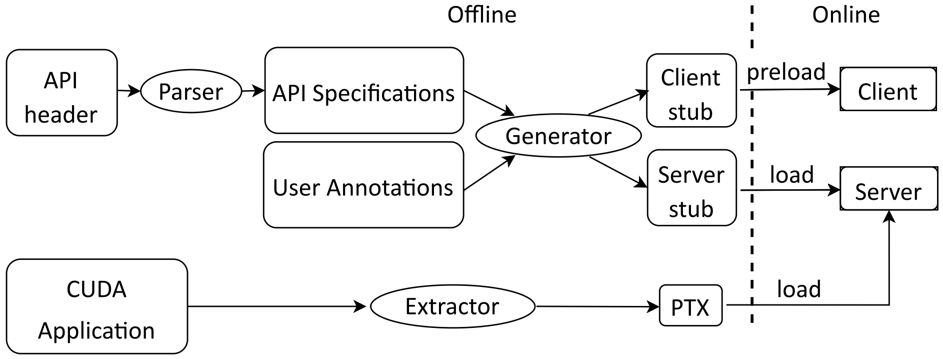


Figure 2.4: Client and Server stub generation (offline phase) and loading (online phase). The three steps of the offline phase are performed by the parser, the generator, and the extractor.

2.1.3 Transport Layer

Arax applications and the Arax server are separate processes. Consequently, Arax requires an IPC mechanism for the applications and the server to exchange tasks and data. We use a shared memory approach to avoid system calls in the common path. Our initial implementation of the shared memory transport layer uses an extra copy of the data. In particular, application data are copied in the shared memory segment. Then, the server copies the data to the accelerator memory. We evaluate the impact of this copy in Section 2.3.1. We believe that future versions of Arax should consider zero-copy mechanisms by using shared pointers between the application and server address spaces.

2.1.4 Autotalk: stub-generator

Existing frameworks are complex and require considerable manual effort to port them to different accelerator APIs. Arax reduces this effort by providing Autotalk, a generator implemented as a python script that creates client and server stubs for each accelerator API offline (Figure 2.4; Offline). The generated stubs are linked with the applications and the Arax server during their initialization (Figure 2.4; Online). The offline phase is performed once and consists of three main steps: *parse*, *generate*, and *extract*.

Step 1: Parse. The Autotalk parser gets as input an accelerator API header and produces an API specification file (Figure 2.4; API specifications). The specification file contains for each API call, the number of arguments, their order, and the return value. The current version of Autotalk targets the CUDA API (v10.1) and can automatically create the API specification file for 85% of the existing functions (1800 in total) without requiring any user intervention.

Step 2: Generate. The Autotalk generator takes as input the API specification file that has been produced from the parser and an annotation file provided by the user (Figure 2.4; User Annotations). This user-provided annotation file contains information about the function calls that cannot be auto-produced from the Autotalk parser and require manual effort. The parser fails for some API calls because they take pointers as parameters, the bounds of which cannot be generated automatically in C/C++, and the address space they belong to (host or device), cannot be found automatically. The user annotation file provides this information with size expressions that calculate the bounds of each pointer. It also specifies the address space of the pointer parameter based on each API's documentation. The user annotation file is created once and consists of 2-3 lines of code for each function that cannot be generated automatically. Currently, these functions are about 270 (out of the 1800 in CUDA API v10.1). The generator produces the client and server stubs using the API specification and the user annotation files. The client stub contains an implementation of the accelerator API used by applications over the Arax API. The server stub contains the function calls to accelerator libraries (e.g., `BLAS, RAND`).

Step 3: Extract. Autotalk uses `cuobjdump` [65, 95] to extract kernels from the native CUDA applications that are not included in accelerator libraries (Figure 2.4; Extractor); these kernels are in `PTX` format [71] and are dynamically linked with the server executable so they can be invoked at runtime.

2.1.5 Implementation issues

The current version of Arax supports the execution of kernels on CPU and three accelerator types: NVIDIA GPUs, AMD GPUs, and Intel Altera FPGAs. To add a new accelerator, one should implement an new accelerator thread that will contain the following functions: `accelAlloc()` and `accelFree()` that are responsible for memory allocations and de-allocations respectively; `accelSyncTo()` and `accelSyncFrom()` that transfer data to and from the accelerator; `accelMemset()` that sets device memory to a particular value and `accelDevcpy()` that performs a transfer within an accelerator. These functions are implemented once for each accelerator type using the native accelerator API.

Accelerator APIs offer function calls that query specific device information, such as `cudaGetDeviceProperties()`, and `cudaGetDeviceCount()`. The design of Arax hides the number and type of the underlying accelerators, so it cannot provide such information. Instead, the Arax server returns some "synthesized" information, ensuring that calls depending on such information will run correctly. This information is based on the specifications of the accelerator with minimal resources; by doing so, we ensure that an application will execute to at least one accelerator. We note that this approach is acceptable for the applications used in our experimental evaluation; however, other applications may require advanced policies, which is left as future work.

Existing applications can use library handles or generators, such as `cuBLAS handles` or `cuRAND generators`. Typically, library handles and generators are opaque structures that store the context required from a library. However, these handles do not have the same semantics in all accelerator libraries. For instance, `CBLAS` (the `BLAS` library for CPUs) does not have the notion of handles. Such cases are managed by Arax before issuing a task to an accelerator: The accelerator threads that are implemented using the native accelerator API prepare handles and generators according to the semantics of each accelerator and use them during the kernel invocation.

2.1.6 Implementing new applications using Arax

In this section, we describe how we port a gaussian from Rodinia benchmark suite to Arax API as an example to write new applications to Arax. To port gaussian to Arax we use the CUDA version (Listing 2.1) and replace all CUDA calls (i.e. allocations, transfers, and kernel calls) with the relevant Arax API calls, as shown in Listing 2.2. In particular a *cudaMalloc()* is replaced with *a_allocate()*, *cudaMemcpy(HostToDevice)*, *cudaMem-*

cpyAsync(HostToDevice), and *cudaMemcpy(DeviceToHost)* with *a_sync_to()* and *a_sync_from()* respectively. Regarding calls that set or copy data to the device, such as *cudaMemset* and *cudaMemcpy(DeviceToDevice)* we issue a task to the server that will call the appropriate function. Moreover, we replace *cudaLaunchKernel()* with *a_issue()*. When the kernel call is synchronous we use *a_task_wait()*.

```
void ForwardSub() {
        int t;
        float *m_cuda, *a_cuda, *b_cuda;

        // allocate memory on GPU
        cudaMalloc((void **)&m_cuda, Size * Size * sizeof(float));
        cudaMalloc((void **)&a_cuda, Size * Size * sizeof(float));
        cudaMalloc((void **)&b_cuda, Size * sizeof(float));

        // copy memory to GPU
        cudaMemcpy(m_cuda, m, Size * Size * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(a_cuda, a, Size * Size * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(b_cuda, b, Size * sizeof(float), cudaMemcpyHostToDevice);

        int block_size, grid_size;
        block_size = MAXBLOCKSIZE;
        grid_size = (Size / block_size) + (!(Size % block_size) ? 0 : 1);

        dim3 dimBlock(block_size);
        dim3 dimGrid(grid_size);

        int blockSize2d, gridSize2d;
        blockSize2d = BLOCK_SIZE_XY;
        gridSize2d = (Size / blockSize2d) + (!(Size % blockSize2d ? 0 : 1));

        dim3 dimBlockXY(blockSize2d, blockSize2d);
        dim3 dimGridXY(gridSize2d, gridSize2d);

        for (t = 0; t < (Size - 1); t++) {
                Fan1<<<dimGrid, dimBlock>>>(m_cuda, a_cuda, Size, t);
                Fan2<<<dimGridXY, dimBlockXY>>>(m_cuda, a_cuda, b_cuda, Size, Size - t, t);

        }
        // copy memory back to CPU
        cudaMemcpy(m, m_cuda, Size * Size * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(a, a_cuda, Size * Size * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(b, b_cuda, Size * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(m_cuda);
        cudaFree(a_cuda);
        cudaFree(b_cuda);
}
```

Listing 2.1: guassian implementation with CUDA API.

```
void ForwardSub() {
        a_accel *accel;
        gaussianArgs gArgs;

        a_task *t_fan1, *t_fan2;
        std::vector<a_task *> free_v;
        free_v.reserve(128);
        a_buffer_s a_fan1;         // input
        a_buffer_s m_fan1;         // output
        a_buffer_s inputs_fan2[2]; // inputs fan2
        a_buffer_s b_fan2;         // input and output for fan2
        // Set the accelerator type.
        a_accel_type_e accelType = ANY;

        // Create a task for each kernel.
        a_proc *fan1 = a_proc_get("fan1");
        a_proc *fan2 = a_proc_get("fan2");

        accel = a_accel_acquire_type(accelType);

        // Allocate memory
        m_fan1 = (a_buffer_s *)a_allocate(
        vpipe, Size * Size * sizeof(float), 64); // output fan1
        a_fan1 = (a_buffer_s *)a_allocate(
        vpipe, Size * Size * sizeof(float), 64); // input fan1
        b_fan2 =
        (a_buffer_s *)a_allocate(vpipe, Size * sizeof(float), 64);

        // Transfer data
        a_sync_to(m_fan1, accel, m);
        a_sync_to(a_fan1, accel, a);
        a_sync_to(b_fan2, accel, b);

        inputs_fan2[0] = m_fan1;
        inputs_fan2[1] = a_fan1; // in to fan2
        gArgs.size = Size;
        // Issue task to accelerator (block, grid size are calculated in the server).
        for (int t = 0; t < (Size - 1); t++) {
                gArgs.t = t;
                t_fan1 = a_issue(accel, fan1, &gArgs, sizeof(gaussianArgs), 1,
                &a_fan1, 1, &m_fan1);
                free_v.push_back(t_fan1);

                t_fan2 = a_issue(accel, fan2, &gArgs, sizeof(gaussianArgs), 2,
                inputs_fan2, 1, &b_fan2);
                free_v.push_back(t_fan2);
        }

        a_sync_from(a_fan1, a);
        a_sync_from(m_fan1, m);
        a_sync_from(b_fan2, b);

        for (auto &i : free_v) {
                a_task_free(i);
        }
        a_free(m_fan1);
        a_free(a_fan1);
        a_free(b_fan2);
        a_accel_release(&accel);
}
```

Listing 2.2: guassian implementation with Arax API.

2.2 Experimental Methodology

For our evaluation, we use two servers with different accelerator types, as shown in Table 2.2. The first server (S1) is equipped with one FPGA and two different GPUs, while the second (S2) with two identical NVIDIA GPUs. The NVIDIA RTX 4000 is equipped with 8 GB of GDDR6, has 2304 CUDA cores, and is connected over PCIe v3 x16. The NVIDIA RTX 2080 Ti has 11 GB GDDR6, consists of 4352 CUDA cores, and uses a PCIe v3 x8 port in our server. For the NVIDIA GPUs, we use CUDA v10.1. The Intel Arria 10 FPGA (de5a_net_ddr4) has 4 GB of DDR4 and uses PCIe v3 x8. We use OpenCL 1.2 and Quartus 20.1 to implement and compile the bitstreams and the server accelerator threads. AMD RX550X GPU has 512 compute cores, has 4 GB of GDDR5 VRAM, and uses PCIe v3 x16. For the AMD GPU, we

ID	CPU	RAM (GB)	PCIe Gen	Accelerators
1	AMD EPYC 7551P 64-Core @ 3.0GHz	128	3.0	(i) NVIDIA RTX 4000, (ii) Intel Altera Arria 10 FPGA (iii) AMD RX550X GPU
2	Intel Xeon CPU E5-2620 16-Core @ 2.10GHz	256	3.0	2x NVIDIA Geforce RTX 2080

Table 2.2: Servers configurations.

Suite / Framework	Application	Input Data (MB)	Output Data (MB)	Kernel code
Rodinia	BFS	40	4	CUDA ROCm OpenCL
	Gaussian (2k)	32	32	
	Gaussian (1k)	8	8	
	Hotspot	8	4	
	Hotspot3D	16	8	
	LavaMD	60	25	
	NN	16	8	
	NW	512	256	
	Particle	1.5	0.25	
	Pathfinder	1024	0.6	
Caffe	Mnist	284	279	CUDA ROCm
	Siamese	566	556	
	Cifar	1052	1050	
	Googlenet	3416	3400	
	Alexnet	5472	5470	
	Caffenet	4274	4274	
TF	Mnist	5460	5460	CUDA
Keras+ TF	Computer Vision (CV)	3316	3216	CUDA
	Generative Deep Learning (GDL)	3974	3871	
	Graph NN (GNN)	2784	2780	
	Recommendation Systems (RS)	5310	5310	

Table 2.3: Applications and their memory footprint.

use ROCm v4.1.0.

In our evaluation, we use a set of micro-benchmarks and real-world applications. We use micro-benchmarks to evaluate the overhead Arax introduces compared to native kernel execution and data transfers. For kernel execution, we use an empty kernel, without computation and data. Regarding data transfers, we copy varying amounts of data from the application to the accelerator via the Arax primitives.

Table 2.3 shows the real-world applications and their inputs/outputs used for our evaluation. Similar to AvA [113], we use applications from Rodinia [17] as well as model training and inference from Caffe [41] and TensorFlow [1] version 2.3.2. The last column of Table 2.3 indicates the accelerator environment for which each kernel is available. We use CUDA for NVIDIA, ROCm for AMD, and OpenCL for FPGA. Using optimized accelerator kernels is orthogonal to our work.

For Caffe Mnist, Siamese, and Cifar, we use the datasets downloaded by the scripts provided in the Caffe repository. For Caffe Googlenet, Alexnet, and Caffenet, we use the ImageNet dataset [87]. For TensorFlow Mnist [50] we use the dataset in LeCun et. all [49]. For Keras, we use Computer Vision (CV), Generative Deep Learning (GDL), Graph Neural Networks (GNN), and Recommendation System (RS) applications, with the code and datasets provided in the Keras repository [34]. Regarding Rodinia datasets, we increase their size by 10× and the kernel execution time by 8×, compared to previous works [113] because the default values are small for executing on a real system (as opposed to simulation).

In all native application runs used as baselines, we add a warm-up phase that initiates the accelerator and moves its power state from idle to maximum. With this warm-up, we avoid the latency implied to the first accelerator call. The FPGA warm-up phase includes the creation of the context, the command queue, the program, and kernel creation, while it excludes the bitstream loading time. In runs with Arax, this warm-up phase is performed by our server. We exclude this warm-up time from all our comparisons.

Finally, to evaluate accelerator sharing, we create a set of workloads with concurrently running applications. These workloads are listed in Table 5.4 and contain a mix of compute- and data-intensive applications. Workloads A-H use multiple instances of the same application, while I-P include different applications.

2.3 Experimental Evaluation

Our evaluation tries to answer the following questions:

- What is the overhead of Arax for decoupling applications from accelerators (§2.3.1)?

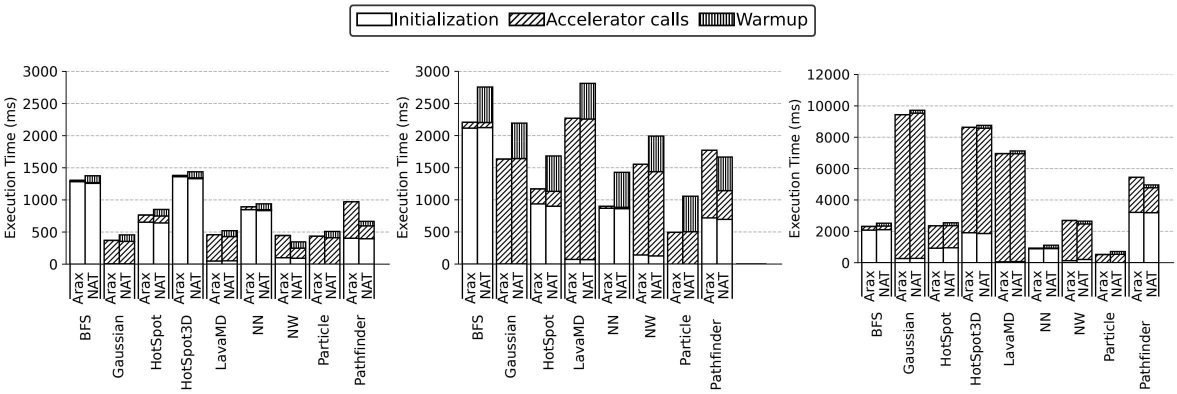
Workload id	Description	Iterations per instance (k)	Epochs per instance
A	2xMnist	10	500
B	4xMnist	10	500
C	2xCifar	9	100
D	4xCifar	9	100
E	2xGaussian	-	-
F	4xGaussian	-	-
G	2xLavaMD	-	-
H	4xLavaMD	-	-
I	Mnist-Siamese	100-50	5000-50
J	Siamese-Cifar	12-9	30-100
K	2xMnist-Siamese-2xCifar	100-12-9	5000-30-100
L	3xMnist-Siamese-2xCifar	100-12-9	5000-30-100
M	Hotspot-Guassian	-	-
N	Gaussian-LavaMD	-	-
O	Particle-Hotspot	-	-
P	Gaussian-Hotspot-LavaMD-Particle	-	-

Table 2.4: Workloads for spatial sharing.

- How effective is accelerator sharing in Arax (§2.3.2)?
- What is the performance improvement of elasticity (§2.3.3)?
- What is the overhead of application migration (§2.3.4)?
- What is the overhead introduced by Arax in real-life ML frameworks (§2.3.5)?

2.3.1 Overhead of accelerator decoupling

In this section, we evaluate the performance of Arax with heterogeneous accelerators. We use Rodinia [17], which offers OpenCL, ROCm, and CUDA kernels. To execute Rodinia in Arax, we port the host code of its CUDA version. Figure 2.5 shows a breakdown of the total execution time achieved for Arax and native execution. The breakdown consists of: (i) the initialization phase, i.e., generation of application inputs, (ii) the accelerator calls, i.e., memory allocations, memory transfers, and the actual kernel execution, and (iii) the accelerator warm-up, i.e., an accelerator call that changes the accelerator power state. We note that the warm-up time is not considered in our comparisons.



(a) CUDA-NVIDIA GPU (b) ROCm-AMD GPU (c) OpenCL-FPGA

Figure 2.5: Overhead of Arax compared to native (NAT) using Rodinia benchmarks over heterogeneous accelerators.

Figure 2.5(a) shows the execution time of Rodinia when running on an NVIDIA GPU. The relative performance of Arax is between 1% and 5% for all benchmarks, except NW (78%) and Pathfinder (62%). The reason for that is the low computation-to-communication ratio NW and Pathfinder exhibit. In particular, the computation-to-communication ratio for NW is 0.3: 0.9 ms for computation over 3 ms for transferring data. Pathfinder is 0.12: 21 ms for computation over 179 ms for transferring data. The other Rodinia applications have more significant computation-to-communication ratios than Pathfinder. For instance, Gaussian's computation-to-communication ratio is 30: 330 ms for computation over 11 ms for transferring data. We run some Rodinia applications with varying computation-to-communication ratios to validate our findings. For instance, Hotspot3D transfers input data to the accelerator and performs a configurable number of passes upon this data. The relative performance of Arax compared to native CUDA for ten iterations is 1.13×. As we increase the number of iterations to 100 and 1000, the relative performance compared to native is 1.03× and 1.01×, respectively. The overall overhead of Arax is 5.5% (geometric mean) for Rodinia applications, ranging from 1% up to 78%.

Figure 2.5(b) and Figure 2.5(c) show the total execution time of Rodinia when running on an Intel FPGA and an AMD GPU accordingly. We observe that the relative performance of Arax compared to AMD GPUs is 2% across all applications, except NW and Pathfinder (8% and 55% respectively). Similarly, the performance for FPGA is up to 3% for all applications, except NW and Pathfinder (9% and 14% accordingly).

The difference in relative performance between the NVIDIA GPU and the other two, i.e., FPGA and AMD GPU, is because the kernel execution takes much less time in the NVIDIA GPU. As a result, the computation-to-communication ratio is proportionally smaller in NVIDIA GPUs than in the AMD GPU or the FPGA.

Cost analysis for kernel launch and data transfer To measure the overhead of a kernel launch, we time the execution of an *empty* kernel. Since kernel launch is asynchronous, we also place a barrier to ensure that the kernel has finished its execution. Figure 2.6 shows the corresponding operations for the case of CUDA and Arax. As we can see, a simple *launch kernel* in CUDA costs approximately 9000 CPU cycles, mainly because it involves a system call. The *device barrier* operation, which is required to wait for the kernel to finish, costs about 2300 CPU cycles. On top of that, Arax introduces a constant overhead of approximately 1500 CPU cycles that are always applied before the *launch kernel*. This overhead is small compared to the duration of the actual *launch kernel* call and becomes proportionally negligible as the kernel duration increases. This effect favors kernels running on AMD GPUs and Intel FPGAs since they exhibit a slower execution than NVIDIA GPUs. For example, the NVIDIA GPU can execute Pathfinder 11× faster than the FPGA and 2× faster on the AMD GPU. Thus, the overheads of Arax are less pronounced when it is compared to native OpenCL (FPGA) and ROCm (AMD).

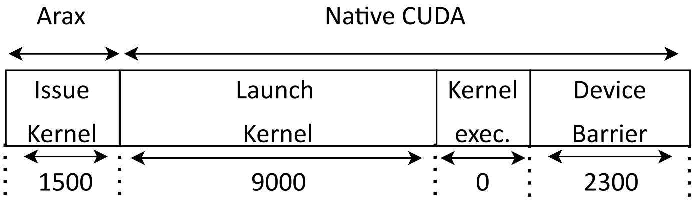


Figure 2.6: Breakdown of overhead for launching an empty kernel with Arax (CPU cycles).

To measure the overhead implied to a data transfer, we create a micro-benchmark that transfers variable size data. On average, Arax is 1.7× slower than native CUDA, due to the extra copy performed to the shared memory segment. In particular, to transfer 1 GB data from an application to the accelerator, Arax requires 180 ms for the CUDA copy and an-

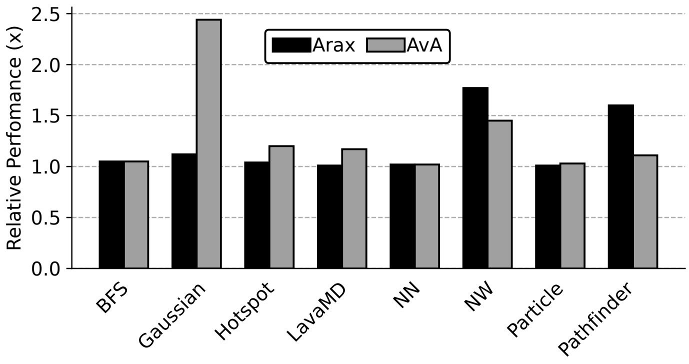


Figure 2.7: Execution time normalized to native for Arax and AvA.

other 135 ms for the copy from the application to the shared memory. The extra copy in the shared memory achieves 8.2 GB/s throughput (measured by the STREAM [64] benchmark, using a single CPU-core). We note that this overhead affects primarily the applications that exhibit a low computation-to-communication ratio. As part of our future work, we plan to use zero-copy between the applications and server address spaces to minimize this overhead.

Arax vs AvA We use Rodinia to compare Arax and AvA [113], which is a state-of-the-art framework for heterogeneous accelerators. Figure 2.7 shows the normalized execution time to native for both Arax and AvA. Arax performs between 10%–32% better than AvA for Gaussian, Hotspot, LavaMD, and Particle. This is because the overhead of *task issue* in Arax is less than AvA. In AvA, every accelerator call goes through the hypervisor, which is not the case for Arax. For NW and Pathfinder, Arax results in 78% and 62% more execution time than native. For these benchmarks, AvA introduces 40% and 3% overhead, respectively, compared to native. These two applications have a low computation-to-communication ratio, and the data copy in Arax across the application and server address spaces becomes more pronounced. This indicates that zero-copy data transfers from the client to server address space are necessary for applications with a low computation-to-communication ratio.

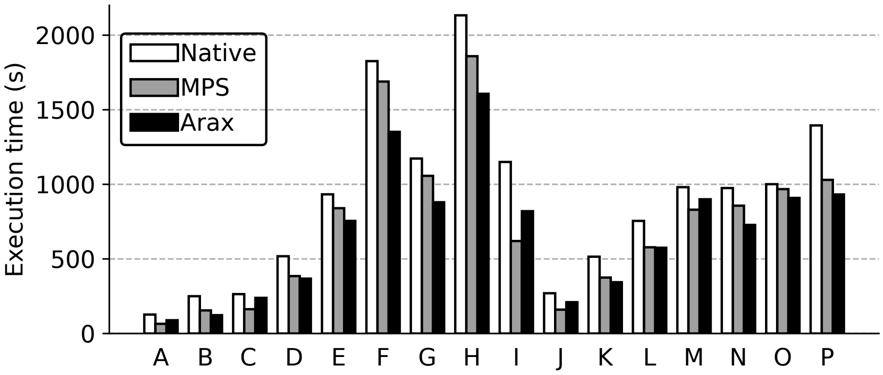


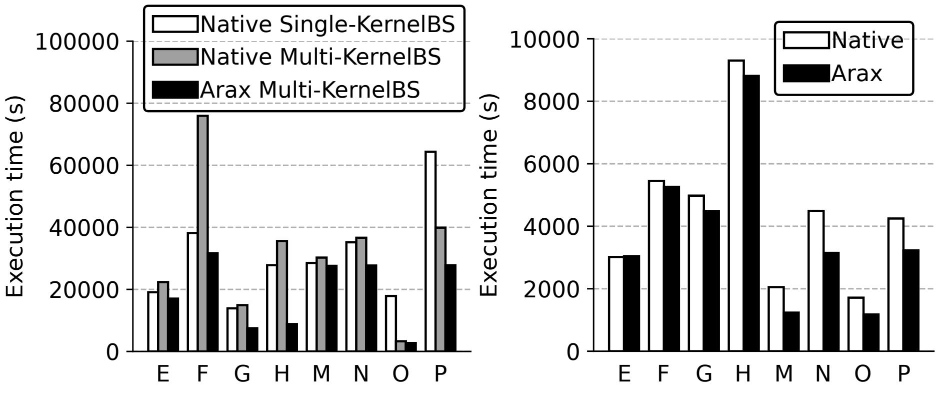
Figure 2.8: Effectiveness of sharing with NVIDIA GPUs for Arax, native (without MPS), and MPS.

2.3.2 Effectiveness of accelerator sharing

We now compare Arax sharing with NVIDIA MPS [20], AMD, and FPGA sharing mechanisms. Even though AMD GPUs do not provide any documentation regarding sharing, our experimentation reveals that they offer spatial sharing by default. Intel Altera FPGAs do not natively support spatial sharing; as a matter of fact, when an application starts, it binds the FPGA, and all subsequent applications fail to start. Instead, with Arax, applications do not have direct access to the FPGA; hence they do not acquire the FPGA exclusively, and they can share its resources.

Figure 2.8 compares sharing mechanisms upon NVIDIA GPUs. We compare Arax (spatial sharing) with MPS (spatial sharing) and native CUDA (time-slice sharing) using the workloads listed in Table 5.4. The x-axis shows the different workloads, while the y-axis shows the total execution time achieved. Overall, the execution time of Arax is comparable to MPS. However, with four concurrent instances, workloads B, D, F, H, K, P, Arax has between 4% and 20% less execution time. Even though we could not investigate the reason behind this, due to the closed-source nature of NVIDIA MPS, we run further microbenchmarks with different GPU models, i.e., RTX 2080, V100, and TITAN V, with a varying number of in-flight kernels and concurrent instances. This evaluation shows the same performance improvement of Arax over MPS. To verify these findings, we disclosed them to NVIDIA, which has confirmed them as two separate issues[1].

[1] ID 3559606, ID 3350973



(a) Intel FPGA (b) AMD GPU

Figure 2.9: Effectiveness of sharing with Intel FPGAs and AMD GPUs for Arax and Native. For FPGAs we compare Arax with a multi-kernel & a single-kernel bitstream.

Comparing Arax with native CUDA (time-slice sharing), we observe that Arax provides 31% (geometric mean) less execution time for all workloads. With four concurrent instances, the performance improvement is more pronounced. In particular, Arax has between 1.32× and 2× less execution time compared to native.

Figure 2.9(a) shows the execution time when multiple applications use the same FPGA for native (time-slice sharing) and Arax (spatial sharing). We examine two versions of native FPGA sharing: (a) The *Single-KernelBS* case in which the bitstream loaded to the FPGA contains one kernel, and (b) the *Multi-KernelBS* case in which the bitstream contains multiple kernels. The drawback of the former is that the FPGA requires reconfiguration to execute a kernel that is not in the current bitstream—an operation that costs about 15 s. In the latter case, i.e., *Multi-KernelBS*, the execution time of an individual kernel, running standalone, increases due to conflicting requirements upon the bitstream compilation. For instance, Gaussian execution takes about 9200 s when a single kernel bitstream (*Single-KernelBS*) is used. For the multi-kernel case (*Multi-KernelBS*), the execution time increases by 17% for the two kernel bitstream and by 52% for the four kernel bitstream.

The spatial sharing capability provided by Arax (Figure 2.9(a); Arax *Multi-KernelBS*) decreases execution time from 3% up to 85% compared to the single kernel bitstream (Figure 2.9(a); Native *Single-KernelBS*) and between 9% and 75% compared to the multi-kernel bitstream (Figure 2.9(a); Native *Multi-KernelBS*). This improvement is because Arax allows applications to execute in parallel in the FPGA, while in the native case, the FPGA is time-shared.

Comparing the *native* single kernel bitstream with the multi-kernel one, we observe that the *Single-KernelBS* is between 6% - 50% faster than *Multi-KernelBS* for workloads E-N. This happens because the reconfiguration time is less than the performance degradation implied by the conflicting requirements of *Multi-KernelBS*. For workload O (Particle-Hotspot), *Multi-KernelBS* has 81% less execution time compared to *Single-KernelBS*. These two kernels do not have conflicting requirements, so their performance degradation is minimal compared to the FPGA reconfiguration time. As the number of reconfigurations increases, as in workload P (Gaussian-Hotspot-Lava-Particle), it is worth packing kernels in the same bitstream to avoid the reconfiguration overhead. In workload P, the execution time of *Multi-KernelBS* is 40% less than *Single-KernelBS*.

Figure 2.9(b) compares Arax with AMD spatial sharing. Arax provides comparable performance to the AMD native execution. In some workloads, such as M and N, Arax provides 45% and 66% performance improvement. Due to the limited information provided by AMD, we extrapolate that there might be performance issues similar to NVIDIA MPS.

2.3.3 Performance gains of elasticity

Arax can opportunistically grow and shrink the number of homogeneous or heterogeneous accelerators provided to an application.

Elasticity with *homogeneous* accelerators To evaluate the performance of elasticity, we modify a representative set of the Arax Rodinia applications to use multiple task queues and, consequently, multiple accelerators. Figure 2.10 depicts the execution time of one application, when increasing the amount of NVIDIA GPUs and the corresponding streams, from one (*1xgpu-1xstr*) to two (*2xgpu-2xstr*). For this experiment, we use the S2 server from Table 2.2, and each application creates eight task queues. The first GPU uses a PCIe v3 ×8, while the second one uses a PCIe v3 ×16. Due to this heterogeneity aspect, we could not see a linear performance improvement when using two GPUs.

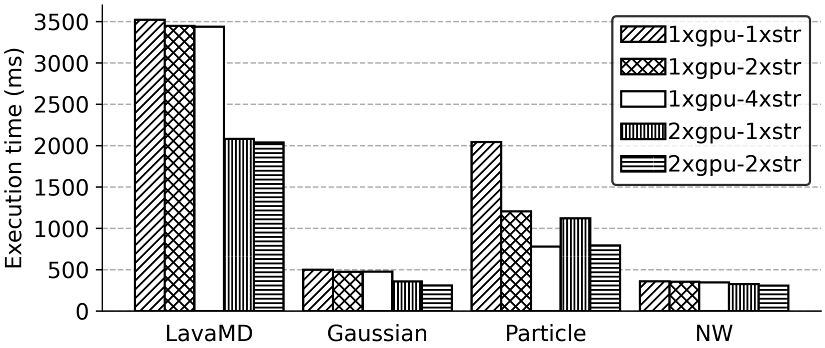


Figure 2.10: Performance improvement of applications when increasing the number of *homogeneous* accelerators or GPU streams.

Gaussian (1k) and LavaMD do not scale as the number of streams in a GPU increases (*1xgpu-1xstr, 1xgpu-2xstr, 1xgpu-4xstr*). This happens because their kernels occupy almost all the GPU threads, so two or more kernels cannot execute in parallel in a GPU. On the contrary, when we provide two GPUs (*2xgpu-1xstr, 2xgpu-2xstr*) to Gaussian, its execution time decreases by 1.35× compared to four streams in a GPU (*1xgpu-4xstr*). LavaMD execution time decreases by 1.7× compared to four streams.

Particle execution time decreases as we increase the number of streams per GPU. In particular, the execution time of two streams (*1xgpu-2xstr*) and four streams (*1xgpu-4xstr*) compared to one stream (*1xgpu-1xstr*) decreases by 1.6× and 2.6×, respectively. This happens because four Particle kernels do not contend for resources in the GPU, and there is not much serialization due to data transfers. The execution time in the two GPU setups (*2xgpu-1xstr*) is comparable to the one GPU configuration with two streams (*1xgpu-2str*), whereas it is 1.4× worst compared to the one GPU with four streams setup (*1xgpu-4xstr*). Finally, NW execution time decreases by up to 16% when increasing the number of GPUs and streams. NW scaling is limited because the computation-to-communication ratio is small.

Elasticity with *heterogeneous* accelerators We now evaluate the elasticity over heterogeneous accelerators using the same applications as in homogeneous elasticity. We note that these applications do not need any modifications due to Arax's accelerator agnostic API. Figure 2.11 shows the execution times of four representative applications using

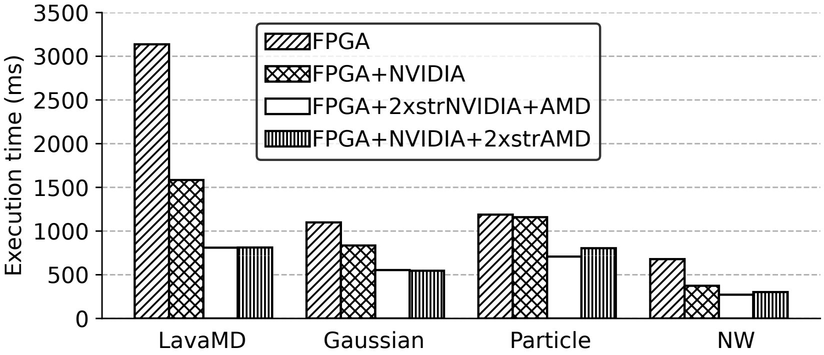


Figure 2.11: Performance improvement of applications when increasing the number of *heterogeneous* accelerators or GPU streams.

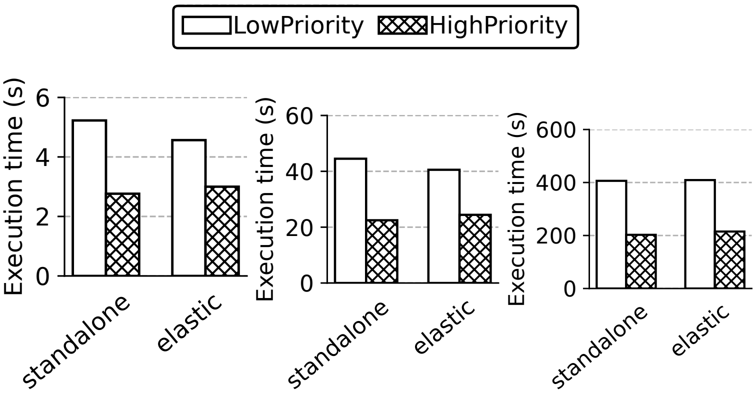
multiple heterogeneous accelerators. Each application is running with the following configurations selected by the Arax server: (a) *1xFPGA*, (b) *1xFPGA and 1xNVIDIA*, (c) *1xFPGA, 1xNVIDIA with two streams and 1xAMD*, (d) *1xFPGA, 1xNVIDIA, and 1xAMD with two streams*, We use the S1 server and four task queues for each application. In this setup, the Arax server at its startup initializes all the accelerators that exist in the server, and then it assigns one or more accelerators in a round-robin fashion to each application according to its parallelism.

As shown in Figure 2.11, the execution time of LavaMD, Gaussian, and NW decreases by 2× when an NVIDIA GPU is used along with an FPGA, shown with the *FPGA* and *FPGA+NVIDIA* bars. As we add more accelerators along with the FPGA, shown with the *FPGA+2strNVIDIA+AMD* and *FPGA+NVIDIA+2strAMD* bars, the execution time of LavaMD, Gaussian, and NW decreases by 1.95×, 1.8×, and 1.3× compared to *FPGA+NVIDIA*, respectively.

Finally, we notice that the performance improvement of Particle between the *FPGA* only setup and the setup with the FPGA and an NVIDIA GPU is only 2%. This is because the execution in RTX 4000 is slower than in the FPGA. When we add more accelerators, shown as *FPGA+2strNVIDIA+AMD* and *FPGA+NVIDIA+2strAMD*, the performance increases by 1.5× compared to the *FPGA+NVIDIA* setup.

2.3.4 Overhead of application migration

Arax's application migration moves application tasks and their data across heterogeneous accelerators. In this section, we evaluate migration overheads using Rodinia and Caffe running over homogeneous and heterogeneous accelerators.



(a) 134 MB (b) 514 MB (c) 2 GB

Figure 2.12: Effectiveness of migration when decreasing the accelerators provided to a low-priority application upon the arrival of a high-priority one. We compare elasticity with the standalone execution in which applications are statically assigned to accelerators. We use datasets from 134 MB up to 2 GB.

Application migration with *homogeneous* accelerators We use the Gaussian application and the S2 server to evaluate our migration mechanism. To increase/decrease the accelerators assigned to an application, we require an assignment policy. We use the elastic assignment policy described in §2.1.2. We run two applications, one with low-priority and one with high-priority. The low-priority application starts first, and the high-priority arrives after a while. In the standalone setup, the low-priority application is statically assigned to an accelerator (A1) while the second accelerator is idle (A2). When the high-priority arrives, it is assigned to A2. With elasticity enabled, the low-priority application initially uses both A1 and A2 since the load is low. Upon the arrival of the high-priority application, the accelerator selector shrinks the resources provided to the low-priority one. The accelerator selector uses the Arax application migration mechanism to move the low-priority application state to A1. Now the low-priority application uses A1, while the A2 is freed for the high-priority one.

Figures 2.12(a), 2.12(b), and 2.12(c) show the execution time for applications with datasets from 134 MB up to 2 GB. We compare elasticity with the standalone execution time. Figure 2.12 shows that the execution time of the high-priority application increases by only 7% compared to standalone execution. The execution time of the low-priority application decreases slightly since it uses more resources at the beginning of its execution. By breaking down the overhead of our migration mechanism, we observed that 80% of the total time is spent in the first data transfer from the accelerator to the server memory. This data transfer must wait for all the issued kernels (approximately 600 in-flight kernels) in the accelerator hardware queue to finish, and then it can start transferring data. The Gaussian kernel execution time increases as we increase the data size from 134 MB to 2 GB. The average kernel duration is 550 μs with 134 MB and 12 ms with 2 GB. As a result, the waiting time of the transfer call increases; for the 134 MB, the transfer has to wait for 0.33 s, i.e., 600 kernels × 550 μs, whereas for the 2 GB, it waits for 9 s, i.e., 600 kernels × 15 ms. We can use kernel preemption [80] to reduce the waiting time of our migration mechanism, but this is beyond the purpose of this chapter.

Application migration for tasks with dependencies and *heterogeneous* accelerators Now we evaluate the effectiveness and overheads of our migration mechanism for applications containing tasks with dependencies. Frameworks, such as Caffe, may not have kernels for all accelerator types. In particular, Caffe cannot run on AMD GPUs or FPGAs since BLAS is not supported for these two accelerators.

To emulate this scenario, we run Mnist, Siamese, and Cifar (with ten epochs) using the NVIDIA GPU as the primary accelerator and executing some kernels in the CPU, AMD GPU, and Intel FPGA, as a “helper accelerator”. We execute `im2col` and `col2im` kernels to the helper accelerator in all setups. Regarding the FPGA, we implement the `im2col` and `col2im` using OpenCL. In all setups, a migration is triggered every time an `im2col` or a `col2im` task is popped by the main accelerator. The Arax server checks for every task if the current accelerator thread has the kernel required from that task. If the required kernel is not in the server stub of an accelerator thread, the accelerator selector sets the

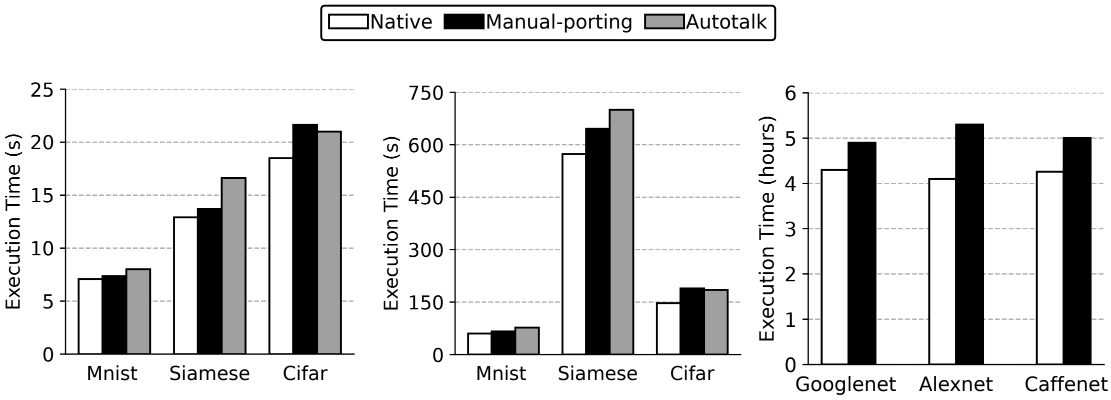
Execution units	Mnist	Siamese	Cifar
NVIDIA-CPU	202	401	520
NVIDIA-AMD	100	213	213
NVIDIA-FPGA	248	N.A.	N.A.
CPU only (single-core)	190	378	490
NVIDIA only	7	13	19

Table 2.5: The execution time (seconds) of Caffe when the execution is migrated from the NVIDIA GPU to another accelerator. *CPU only* and *NVIDIA only* represent the native execution without migrations.

task queue to another accelerator that supports this kernel. The task queue re-assignment triggers data migrations. Consequently, we perform 380k migrations for Mnist (380k times an `im2col` and a `col2im` were not supported), 760k for Siamese, and 890k for Cifar.

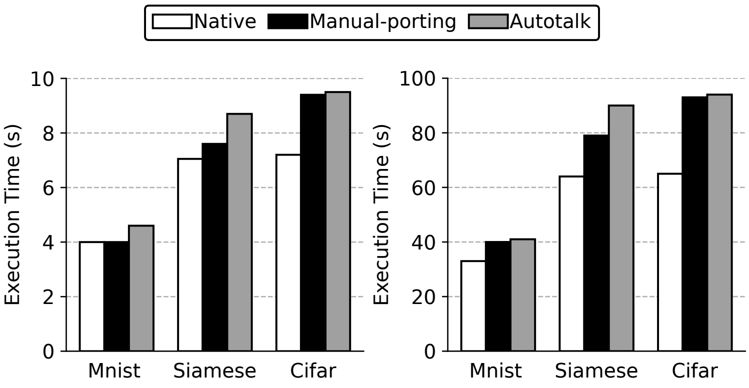
Table 2.5 shows the execution time of Caffe running over heterogeneous accelerators. By comparing the NVIDIA-CPU execution with the native execution using only the CPU, we observe 6% performance degradation due to migrations. On the other hand, by comparing the *NVIDIA-CPU*, *NVIDIA-AMD*, and *NVIDIA-FPGA* with the setup that uses only the NVIDIA GPU (without migrations), the performance is much worse, mainly due to the performance of the kernels to other accelerators. FPGA kernels (`im2col`, `col2im`) run 10× worst than the NVIDIA GPU since they are un-optimized.

2.3.5 Overhead for Caffe and TensorFlow



(a) Train: 10 Epochs (b) Train: 100 Epochs (c) Train: Caffe Models

Figure 2.13: The overheads of Arax using manual-porting and Autotalk (automatic stub generation) compared to native CUDA for Caffe using ML *training*.



(a) Inference: 1000 Iterations (b) Inference: 10000 Iterations

Figure 2.14: The overheads of Arax using manual-porting and Autotalk (automatic stub generation) compared to native CUDA for Caffe using ML *inference*.

In this section, we examine the applicability of our API to complex, real-life ML frameworks and the performance achieved. Arax provides a complete API that can be used directly from new applications (manual-porting) and Autotalk that can be used to auto-port complex frameworks, such as Caffe and TensorFlow. Figures 2.13 and 2.14 show manual-porting, Autotalk, and native CUDA execution time when executing the Caffe framework. We show the training phase with ten epochs of three networks Mnist, Siamese, and Cifar (Figure 2.13(a)). The relative performance of manual-porting compared to native CUDA is between 3% and 17%. With more than ten epochs, as Figure 2.13(b) shows, the execution time increases between 9% and 28%. This slight increase (less than 9%) is because the number of data transfers increases with more epochs. To find the maximum performance degradation regarding training, we run Googlenet, Alexnet, and Caffenet, which perform thousands of epochs and use gigabytes of data. Figure 2.13(c) shows manual-porting and the native CUDA execution time (in hours) for Googlenet, Alexnet, and Caffenet. The performance degradation of manual-porting is between 13% and 28%. The geometric mean of the overhead implied to all Caffe applications is 12.5%.

Figures 2.14(a) and 2.14(b) present the inference phase for manual-porting, Autotalk, and native CUDA. We run inference for Mnist, Siamese, and Cifar with 1k and 10k iterations. The maximum performance degradation for 1k iteration of manual-porting compared to native is 30% with Cifar. For 10k iterations, the degradation is between 24% and

	Mnist	CV	GDL	GNN	RS
Native CUDA	49	190	27	51	235
Autotalk	80	240	28	54	250

Table 2.6: The execution time (seconds) of TensorFlow and Keras for Autotalk and native CUDA.

42%. As explained, the increase in the execution time of manual-porting compared to native CUDA is due to the data transfers. Autotalk adds a minimal overhead compared to manual-porting up to 16%. This happens because with manual-porting we can use fewer barriers and decrease the times that the application blocks. The geometric mean of the overhead implied to all TensorFlow applications is 12.9%.

We use Autotalk to convert TensorFlow and Keras to Arax API. To evaluate the correctness-completeness of Autotalk, we run the unit-tests of TensorFlow, achieving 90% coverage. We also run Mnist and a representative set of Keras applications for the vanilla case, and Arax: some preliminary results are presented in Table 2.6. Our findings suggest that Arax and Autotalk can transparently handle complex, real-life frameworks without significant effort.

2.4 Summary

This chapter, Arax, a runtime that decouples applications from low-level accelerator operations, such as accelerator selection, memory allocation, and task assignment. Arax provides three main capabilities: (a) It assigns application tasks dynamically to different accelerators at runtime and performs all required accelerator memory management internally. (b) It offers fine-grain spatial sharing that improves the utilization of multiple heterogeneous accelerators. (c) It can perform live application migration across heterogeneous accelerators without application modifications or specialized accelerator support. To reduce porting effort, it provides Autotalk, a stub generator that allows linking existing applications, such as TensorFlow and Caffe, to the Arax runtime library with minimal user intervention.

Our evaluation using real-world applications shows that Arax introduces 12% overhead (geometric mean) compared to native execution. Regarding accelerator sharing, Arax improves the execution time up to 20% compared to NVIDIA MPS. Also, its elastic resource assignment reduces total application turn-around time by up to 2× compared to the execution without elasticity support.

Spatial accelerator sharing requires a single accelerator context (common address space) provided by Arax server. However, this common address space exposes application data to other applications that share the same accelerator, leading to security issues. Time-sharing due to context switching resolves this issue at the cost of under-utilization and application latency increase.

Chapter 3

Simplify FPGA Accessing & Sharing

This chapter presents a software layer between FPGAs and applications that reduces communication complexity between application software and accelerator hardware and allows sharing of FPGA across applications. Our approach, named VineTalk –ancestor of Arax, allows applications to access accelerators (i.e., GPUs and FPGAs) transparently, whether they run natively on a server, within virtual machines, or in containers. In this work, we focus on FPGA sharing among different applications that use the same FPGA configuration, and we leave the complete virtualization of FPGAs with dynamic reconfiguration on top of our device-sharing infrastructure as future work. Our system consists of two major components: a Communication Layer (transport layer in Arax) that implements the virtual accelerators (task queues in Arax) and a Software Controller (the Server in Arax) that runs as a user-level process and controls/schedules all accesses to the accelerator. Applications communicate with the virtual accelerators using Software-facing API (client-side API in Arax), while the Software Controller with accelerators, by the Hardware-facing API (used by accelerator threads in Arax), is integrated with the Xilinx SDAccel development environment.

In summary, our work makes the following specific contributions:

1. A Software-facing API, which exposes FPGA accelerators as task queues to applications.

2. A Hardware-facing API, simplifies the porting of kernels for hardware developers.

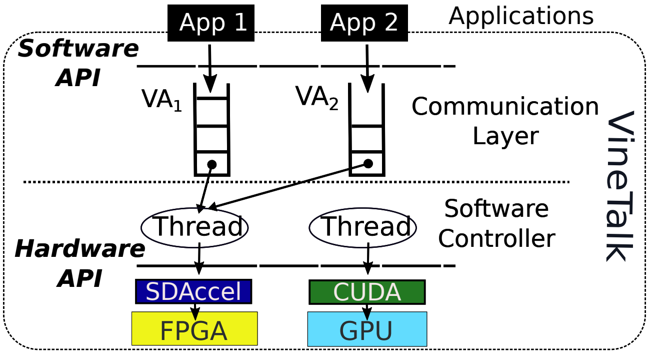


Figure 3.1: Design overview of VineTalk. VA represent *VineAccelerator* (described in Section 3.1.2)

3. A software controller that facilitates sharing of acceleration resources.
4. Integration with the SDAccel framework of Xilinx.

We implement VineTalk for Linux servers using 4700 lines of C code. We demonstrate VineTalk's FPGA abstraction capabilities with a financial application running on a Xilinx ADM-PCIE-KU3 FPGA device. The simplicity of the VineTalk API reduces applications' code by 30% in terms of lines of code. Our results show that applications, that use VineTalk to access the SDAccel framework have a performance overhead from 0.9% up to 4% compared to their native execution. Moreover, for applications that share the same accelerator, the overhead is 0.02 %.

3.1 VineTalk Design

Our design consists of a Software-facing API (§ 3.1.1), Hardware-facing API (§ 3.1.4), a Communication Layer (§ 3.1.2) based on shared memory and a Software Controller (§ 3.1.3). Figure 3.1 presents the design of VineTalk.

3.1.1 Software Facing API

Our Software-facing API replaces the multitude of all platform-specific acceleration APIs, all of which provide functions that handle memory management and data and task trans-

VineTalk Software-facing API	Description	Arax client-side API equivalent
vine_kernel_get()	(Re)Configures an FPGA by loading a kernel from the local repository	N.A.
vine_va_get()	Allocates a *VineAccelerator* that is capable of executing a specific kernel	a_acquire()
vine_buffer_init()	Creates a *VineBuffer* which describes the input and output data for the kernel in the app's address space	a_allocate()
vine_task_issue()	Invokes a kernel to a *VineAccelerator* using one or more *VineBuffer*	a_issue()
vine_task_wait()	Waits for a task to complete. After completion, *VineBuffer*s are updated with computation results	a_wait()

Table 3.1: Main methods of the Software-facing API and comparison with Arax client-side API.

fers between applications and hardware accelerators. The implementation of the API is completely decoupled from accelerator details. VineTalk achieves this by using three main abstractions: *VineAccelerator*s, *VineTask*s, and *VineBuffer*s.

A *VineAccelerator* is a virtual accelerator that can execute kernels to one or more physical accelerators. When a *VineAccelerator* is created by an application, the application specifies the kernel that it needs to execute, which we assume that is preloaded in a repository in VineTalk local node. There is no limit on the number of *VineAccelerator*s that an application can invoke. Although VineTalk can remove unused kernels from the FPGA and instantiate new kernels as requested by applications, we do not explore this further. After a *VineAccelerator* has been allocated, an application can issue *VineTask*s. Different from Arax that tasks and data are separate to support ML frameworks, a *VineTask* represents a kernel with its input/output data. *VineTask*s are not statically mapped to a physical accelerator; the controller assigns them dynamically. Consequently, a single physical accelerator can achieve higher utilization by executing *VineTask*s from multiple *VineAccelerator*s. *VineBuffer*s are used to handle the data transfer between an application address space to the physical accelerator transparently. The Software-facing API enables applications to access these abstractions through a set of methods, presented in Table 3.1.1.

3.1.2 Communication Layer

VineTalk's Communication Layer implements and manages *VineAccelerator*s and *VineBuffer*s. Applications run as separate processes (or VMs) from the Software Controller in a single node. Therefore, *VineBuffer*s and *VineTask*s must be transported across address spaces. To achieve this, we use a shared memory-based transport. VineTalk uses shared memory to store all *VineTask*s and *VineBuffer*s. The advantage of shared memory within a server is that after the setup phase, there is no need to use system calls. This transport approach relies on shared segments that can be mapped across native processes, containers, and VMs. Our shared memory approach currently introduces two additional copies to the shared memory segment when sending/receiving data to/from the accelerator memory, as shown in Figure 3.2.

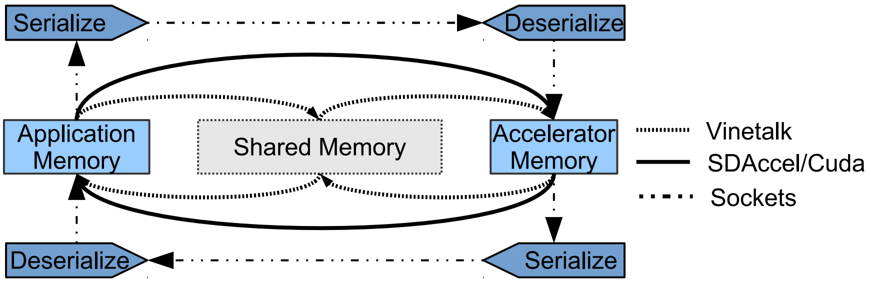


Figure 3.2: Buffer transfers necessary for an inout argument over VineTalk, SDAccel/CUDA, sockets protocols/APIs.

3.1.3 Software Controller

The Software Controller is a process that controls all accesses to the underlying hardware. It monitors *VineAccelerator*s for issued *VineTask*s and utilizes *VineBuffer*s to retrieve the inputs and store the outputs. Moreover, it enables accelerator sharing since it offloads multiple *VineAccelerator*s to the same accelerator. The Software Controller assigns a UNIX thread (i.e., accelerator thread) to each physical accelerator in the system. This accelerator thread first selects the *VineAccelerator* that will serve, based on a scheduling policy (currently round-robin). Second, it pops the first *VineTask* from the selected *VineAccelerator*, and executes it to its physical accelerator. Then, it copies the result to the shared memory segment and serves the next *VineAccelerator*.

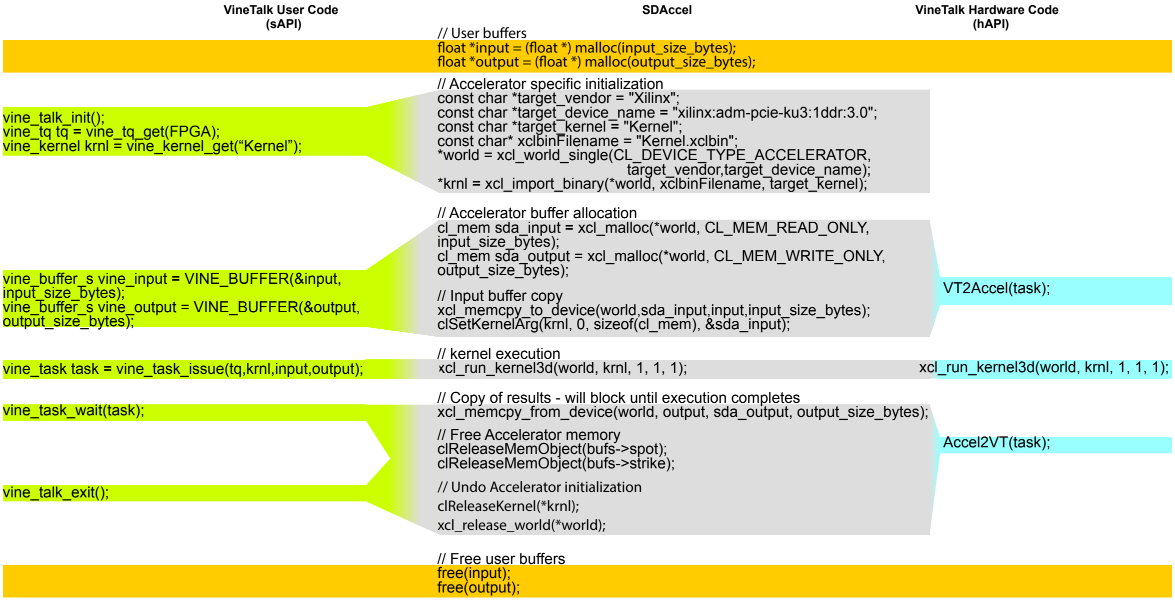
3.1.4 Hardware Facing API

VineTalk allows hardware designers to incorporate new kernels by using mainly two functions: *VT2Accel()* and *Accel2VT()*. VineTalk currently provides a number of different implementations of this simple API to cover kernels for different accelerators, including FPGAs and GPUs. For FPGA devices, VineTalk implements this API in OpenCL and SDAccel, whereas for GPU devices, VineTalk implements this API in OpenCL and CUDA. Porting applications to VineTalk consists of two steps; The first is to create a VineTalk library for each kernel and for each VineTalk library to create a function that contains the kernel invocation. The second is modifying the application to replace all accelerator-related functions with the corresponding methods from the Software-facing API. *VT2Accel()* prepares the input data for an accelerator kernel. A hardware designer is expected to provide this method for each new kernel. The function allocates input *VineBuffer*s on the accelerator memory and copies the contents of *VineBuffer*s of the communication layer. The method will then be used prior to kernel execution by the host controller. Similarly, *Accel2VT()* is called once after the kernel execution finishes. Its goal is to send the output back to *VineBuffer*s and to release the reserved acceleration memory.

3.2 Integration with SDAccel

Xilinx has released the SDAccel framework, a development environment for OpenCL applications that targets Xilinx FPGA-based accelerator cards. It provides an interface between software applications and FPGA devices. The application consists of a host program written in C/C++ and one or more accelerated kernels written in C, C++, or the OpenCL language that run on the underlying FPGA board.

VineTalk intervenes between the application software side and the hardware side of SDAccel and simplifies the development of applications that use FPGA accelerators and incorporate new FPGA kernels into applications. The SDAccel-specific implementation of the Hardware-facing API (§ 3.1.4) allows any SDAccel kernel to be used with applications using VineTalk, with no hardware dependencies. To evaluate the coding effort benefits



SDAccel:

```
// User buffers
float *input = (float *) malloc(input_size_bytes);
float *output = (float *) malloc(output_size_bytes);

// Accelerator specific initialization
const char *target_vendor = "Xilinx";
const char *target_device_name = "xilinx:adm-pcie-ku3:1ddr:3.0";
const char *target_kernel = "Kernel";
const char* xclbinFilename = "Kernel.xclbin";
*world = xcl_world_single(CL_DEVICE_TYPE_ACCELERATOR,
                          target_vendor,target_device_name);
*krnl = xcl_import_binary(*world, xclbinFilename, target_kernel);

// Accelerator buffer allocation
cl_mem sda_input = xcl_malloc(*world, CL_MEM_READ_ONLY,
input_size_bytes);
cl_mem sda_output = xcl_malloc(*world, CL_MEM_WRITE_ONLY,
output_size_bytes);

// Input buffer copy
xcl_memcpy_to_device(world,sda_input,input,input_size_bytes);
clSetKernelArg(krnl, 0, sizeof(cl_mem), &sda_input);

// kernel execution
xcl_run_kernel3d(world, krnl, 1, 1, 1);

// Copy of results - will block until execution completes
xcl_memcpy_from_device(world, output, sda_output, output_size_bytes);

// Free Accelerator memory
clReleaseMemObject(bufs->spot);
clReleaseMemObject(bufs->strike);

// Undo Accelerator initialization
clReleaseKernel(*krnl);
xcl_release_world(*world);

// Free user buffers
free(input);
free(output);
```

VineTalk User Code (sAPI):

```
vine_talk_init();
vine_tq tq = vine_tq_get(FPGA);
vine_kernel krnl = vine_kernel_get("Kernel");

vine_buffer_s vine_input = VINE_BUFFER(&input,
input_size_bytes);
vine_buffer_s vine_output = VINE_BUFFER(&output,
output_size_bytes);

vine_task task = vine_task_issue(tq,krnl,input,output);

vine_task_wait(task);

vine_talk_exit();
```

VineTalk Hardware Code (hAPI):

```
VT2Accel(task);

xcl_run_kernel3d(world, krnl, 1, 1, 1);

Accel2VT(task);
```

Figure 3.3: VineTalk integration with SDAccel.

of VineTalk, we port three financial applications, *Black&Scholes*, *Black-76*, and *Binomial* (described in Section 3.3). We use SDAccel to build three hardware-accelerated variations of the algorithms above. We also wrote and evaluated a simple application, which interfaces with those kernels, submits tasks and data, and reports the results. To port an SDAccel application, we create a VineTalk library for each kernel. We use the Hardware-facing API for each library to simplify the kernel invocation. Moreover, on the application side, we replace all SDAccel-specific functions with the corresponding methods from VineTalk's Software-facing API. The resulting application consists of 30% fewer lines of code and uses semantically much simpler routines. 3.3 shows a snapshot of the two versions of the software side of the *Black&Scholes* application, before and after the use of VineTalk, indicating that the use of VineTalk simplifies application software.

3.3 Performance Evaluation

3.3.1 Experimental Setup

For our experiments, we use one Intel(R) Core(TM) i5-4590 machine running at 3.3GHz, with 16 GBytes of DRAM, and one ADM-PCIE-KU3 FPGA Alpha Data board, with 16 GB DDR3, connected to PCI Express® Gen3 x8. The system runs CentOS 7 with SDAccel version 2016.4.

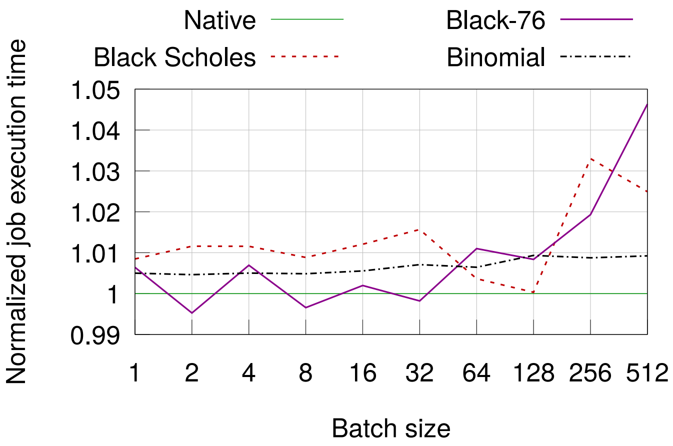


Figure 3.4: Performance comparison between VineTalk-applications, and their standalone execution over SDAccel. The x-axis is the stock batch size, the y-axis is the normalized job execution time in msec.

In our evaluation, we use three financial kernels: *Black&Scholes*, *Black-76*, and *Binomial*. *Black&Scholes* gives a theoretical estimate of the price of European-style options and can also be used for American-style call options. *Black-76* is a variant of the *Black&Scholes* model. *Binomial* option pricing quantizes the time and price of an underlying asset and maps both to a binary tree. We perform each experiment with 2000 options and varying batch sizes between 1 and 512. The batch size represents the number of consecutive options transferred from the application space to the FPGA's memory in a single transfer. We exclude from our results the FPGA reconfiguration overhead, which amounts to 6.15 sec.

Black&Scholes and *Black-76* have four inputs and one output per stock, and with a batch size of one, the input size of a batch is 16 bytes (4 x 4 bytes), and the output size is

4 bytes. On the contrary, *Binomial* uses five inputs and one output, and for batch size of one, the input size of each batch is 20 bytes (5 x 4 bytes), and the output size is 4 bytes.

3.3.2 VineTalk overhead

We compare the execution of the applications above with VineTalk versus the standalone SDAccel execution (Native) to identify VineTalk's overhead. Figure 3.4 presents the normalized job execution time in milliseconds. Job execution times are averages over 20 runs after removing the minimum and maximum values. VineTalk adds negligible overheads for small batch sizes while it adds a slight penalty for larger ones, as shown in Section 3.4. For *Black&Scholes* and *Black-76*, the overhead added from VineTalk is between 0.5% and 4% when the batch size is 512, while for batch sizes between 1 and 32, the overhead is negligible. *Binomial* has an overhead between 0.45% and 0.9% for all batch sizes. In all cases, the main source of the overhead is the two additional data copies (inputs and outputs) required by VineTalk in the shared memory segment. However, although the overhead of those transfers is constant for most experiments, as they transfer similar amounts of data in aggregate, the impact on the execution of each experiment varies.

Runs with larger batch sizes take significantly (by up to two orders of magnitude) less time to execute, and thus, they become more sensitive to the overhead of memory transfers. Table 3.2 demonstrates this difference in the execution time for various batch sizes. The table summarizes the overall application execution time for two batch sizes, 1 and 512, for Native and VineTalk. As the batch size increases, both systems' application performance increases significantly. VineTalk incurs the lowest overhead with the *Binomial* application because it has longer task execution times (and thus a lower communication-to-computation ratio) when compared to the other kernels. *Black&Scholes* and *Black-76* are less compute intensive than the *Binomial* kernel. Which results to greater transfer to compute a ratio of *Black&Scholes* and *Black-76* than the *Binomial*. Consequently, the extra copies have a stronger effect on the execution time of VineTalk.

Benchmark	VineTalk (s)		Native (s)		Overhead (%)	
	Batch 1	Batch 512	Batch 1	Batch 512	Batch 1	Batch 512
Black Scholes	0.39	0.0012	0.38	0.0012	2.56	2.43
Black-76	0.808	0.0018	0.08	0.0014	0.618	4.27
Binomial	256	0.514	254	0.509	0.5	0.97

Table 3.2: Overall application execution time (seconds) and Overhead (%) with 2000 options and batch sizes 1 and 512.

# jobs	Black Scholes			Black-76			Binomial		
	Native (ms)	VineTalk (ms)	Ratio	Native (ms)	VineTalk (ms)	Ratio	Native (ms)	VineTalk (ms)	Ratio
1	9.912	9.88	0.99	18.849	18.47	0.98	5128	5176	1.009
2	9.952	9.68	0.97	18.935	18.39	0.97	5113	5218	1.02

Table 3.3: Comparison of the job execution time of 1 and 2 concurrent VineTalk application(s) with applications running directly on the FPGA (i.e. Native).

3.3.3 Accelerator time-sharing

To evaluate the impact of accelerator sharing, we run up to two instances (jobs) of each application concurrently. Limitations of the current testbed, specifically the number of cores, do not allow us to run more concurrent instances. Concurrent job execution is possible only with VineTalk, which can interleave tasks belonging to different applications. For Native, we execute the two applications sequentially, one after the other. In each experiment, all application instances invoke the same kernel. Each run (i.e., the sum of one or two applications) consists of 2000 options, and the batch size is 50. In the first run (with one job), the job consists of 2000 options, whereas in the second run, the two concurrent jobs, consists of 1000 options. Table 3.3 compares the total serialized execution time (i.e., Native) with VineTalk-powered total execution time. We also present the Native to VineTalk total execution time ratio.

For all applications, the execution time ratio is very close to one. Consequently, the overhead added from VineTalk is negligible. For Black & Scholes and Black-76, the VineTalk to Native job execution time ratio for two concurrent jobs is 0.97, and 1.02 for the Binomial application. Thus multiplexing applications with VineTalk do not introduce overheads.

3.4 Summary

This chapter demonstrates how FPGAs can be used transparently in datacenter servers. In particular, we present how multiple applications can share FPGAs. We design and implement VineTalk, a system that provides a hardware-agnostic abstraction and is designed to be used with different accelerators, including FPGAs and GPUs. VineTalk uses an RPC-like API and a communication channel based on shared memory to allow low-overhead, shared access from applications to accelerators. Our approach is orthogonal to FPGA partitioning and can allow multiple applications to share each partition in an FPGA. Our results show that VineTalk reduces both programmer effort at the application level by reducing lines of code related to kernel invocation by about 30% with significantly simpler semantics and introduces overhead between 0.9% and 4% compared to native application execution over the FPGA. Finally, VineTalk provides the ability of accelerator sharing from consolidated applications, with less than 2% overhead.

Chapter 4

GPU Kernel Revocation

Our goal in this chapter is to guarantee that user-facing tasks with execution time in milliseconds will meet their SLA target in the presence of long-running batch tasks with execution time in the range of seconds. Previous work on GPU preemption [106, 68, 116, 88] does not provide bounded latency and requires the kernel's source code. To tackle this issue, we design and implement TReM as a part of Arax (server). TReM is a revocation mechanism that overcomes the problems of existing preemption approaches. TReM stops a task by aborting its currently executing kernel without saving any state and replays it later.

The first challenge is to stop the executing kernel at any point of its execution, providing bounded latency. To achieve this, TReM stops the kernel from inside the GPU. In particular, we start the actual kernel from a wrapper kernel using CUDA dynamic parallelism [43]. After issuing the actual kernel, the wrapper kernel polls a revocation flag on CUDA unified memory and calls `asm(trap)` to abort the execution of the actual kernel when the host sets the revocation flag.

The second challenge is eliminating the variable latency of moving task state from GPU to host. TReM avoids saving any state of the currently running task to host memory to reduce overhead and latency. Instead, TReM replays revoked tasks. To reduce the replayed work in task granularity, TReM selects to revoke tasks with the latest start time.

We design and implement a runtime scheduler in the VineTalk software controller (i.e., server) that prioritizes user-facing over batch tasks, instructs TReM when to revoke batch tasks, and can manage multiple GPUs in a single node. We design and develop two scheduling policies, Priority and Elastic. Priority tries to allocate a GPU for every user-facing task. As a result, Priority+TReM may revoke as many batch tasks as the number of newly arrived user-facing tasks. Elastic dynamically computes a *minimum* number of accelerators needed to sustain the SLA and devotes the remaining accelerators to batch tasks. Effectively, Elastic+TReM results in fewer revocations, i.e., less work to be re-executed. If there is a need to limit the wasted work for long-running applications, e.g., executing for days, TReM can also be coupled with existing checkpointing mechanisms [52, 16], as discussed in Section 4.5.

Our evaluation shows that TReM revokes an executing kernel in 5ms, while the next kernel requires another 17ms to start (22ms in total). As Section 4.4.1 explains, the 22ms depend on the CUDA runtime. TReM adds negligible overhead to non-revoked tasks and consumes minimal resources in modern GPUs, as discussed in Section 4.4.1. Our experimental results from a real testbed and realistic workloads show that using TReM allows us to meet the SLA for 98% of user-facing tasks in the presence of long-running batch tasks under 89% GPU utilization. Wasted work due to revocations is only 3% of the total user-facing and batch execution time.

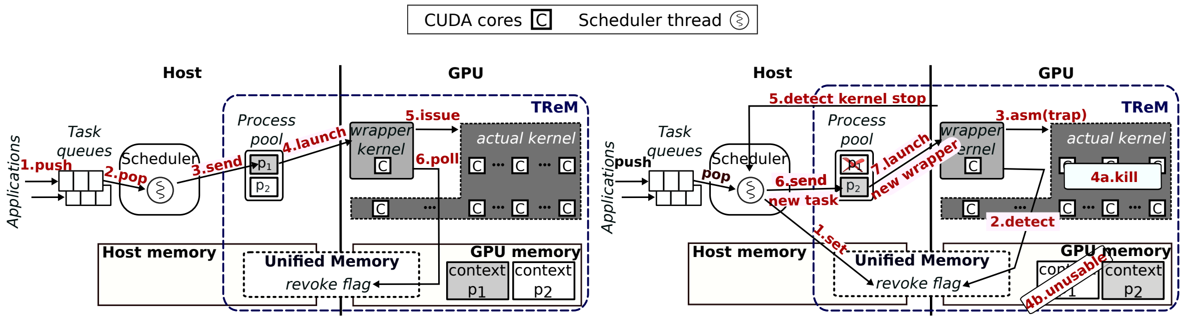
The specific contributions of this chapter are:

1. We design TReM, a task revocation mechanism (§ 4.1) that i) exhibits constant and low overhead, independent of task size and memory footprint, ii) avoids kernel recompilation, iii) incurs zero overhead to non-revoked tasks, and iv) can be deployed to all NVIDIA state-of-the-art GPU architectures.

2. We evaluate TReM using two scheduling policies (§ 4.2), Elastic and Priority, in a *real system* with four (4) GPUs and show the benefits of revoking batch tasks.

3. We use simulation to examine how our approach scales with an increasing number of GPUs and how it behaves under different revocation latencies.

4. We develop a workload generator (§ 4.3) that generates workloads with different characteristics, such as execution times, ratio of batch to user-facing tasks, memory usage, and task inter-arrival time.



(a) Start a CUDA kernel (b) Stop a CUDA kernel

Figure 4.1: TReM overview. The scheduler is part of the Arax server.

4.1 TReM revocation mechanism

TReM revokes the currently executing task by killing its kernel and replays it later using the information maintained at the host. In this work, as in VineTalk, a task consists of kernels with their input and output data.

Table 4.1: Latency of different methods to revoke/preempt a kernel running on a GPU.

Kernel dimensions	**Total threads**	**Latency (ms)**		
		Process kill	**asm(exit)**	**TReM**
kernel<16,16>	256	3000	130	22
kernel<32,32>	1024	3000	195	22
kernel<64,64>	4096	3000	600	22
kernel<128,128>	16384	3000	1430	22

There are three ways to stop the execution of a running kernel; (1) Kill the issuing process on the host, (2) use `asm(exit)`, and (3) use `asm(trap)`. Each host process that performs a CUDA call is associated with a CUDA context. When killing the host process, option (1), the NVIDIA driver clears the killed kernel context and prepares the CUDA environment to be functional. We evaluate this approach while designing TReM, and we find that killing a process introduces a delay of up to 3s to the next kernel (Table 4.1).

Option (2) is to use *asm(exit)*, which is normally called when a kernel uses `return`, terminating the execution of the current thread. We measure the performance of `asm(exit)` for various kernel dimensions, i.e., threads and blocks. Table 4.1 shows that when the number of threads increases, the latency of `asm(exit)` increases significantly. This is because the GPU must wait for all kernel threads to exit. Typical kernels launch thousands of thread blocks with hundreds of threads each [106], which can incur a delay of several seconds. For instance, Table 4.1 shows that for a kernel dimension of <128, 128>, the latency of this mechanism is almost 1.5s. Consequently, `asm(exit)` cannot provide low revocation latency. For Table 4.1, we run each experiment 20 times with the same setup as in Section 4.3.1. To measure the revocation/preemption latency, we start a timer when a kernel is issued and stop it when the next kernel (the kernel after the revoked/preempted kernel) starts its execution in the GPU.

TReM uses `asm(trap)`, option (3), to abort the kernel execution. Commonly, `asm(trap)` is called when a kernel-code assertion fails, and the CUDA context of the issuing process is unusable; thus it cannot be used for subsequent CUDA calls. Calling `asm(trap)` results in immediate termination of the kernel, with constant latency, as we discuss next.

4.1.1 Revoking a kernel with TReM

Figure 4.1 shows an overview of TReM. Applications use *task queues* to issue tasks to the scheduler. This scheduler runs in a different process context from the applications and manages a GPU using a scheduler thread. If multiple GPUs exist in a node, the scheduler spawns multiple scheduler threads, as Section 4.2 explains. A scheduler thread is responsible for dequeuing tasks from a selected queue and issues task kernel to its GPU based on a policy. Moreover, the scheduler thread checks the status of tasks (killed or finished) and reissues them if needed. A GPU is time-shared among applications, serving tasks from multiple task queues. Next, we describe TReM, our revocation mechanism, in detail.

```
void wrapper_kernel(,){
    cudaLaunchKernel(actual_kernel);
    while(1){
        if (revoke_kernel == true)
            asm(trap);
    }
}
```

Listing 4.1: Poll mechanism in the kernel *wrapper*.

As shown in Figure 4.1(a), TReM encapsulates each CUDA kernel in a CUDA wrapper kernel. The wrapper uses one thread and one thread block, i.e., `wrapper <1,1>`. The wrapper then issues the actual kernel, using CUDA dynamic parallelism [43]. With CUDA dynamic parallelism, a parent grid (in our case, the wrapper) launches kernels called child grids (in our case, the actual kernel). Thus, TReM does not require kernel source code. The kernel thread-block dimensions are passed to the wrapper kernel as global variables, and thus they are also available to the actual kernel. Subsequently, the wrapper polls a revoke flag, which is set when the host runtime scheduler decides to kill a kernel running on the GPU. This revoke flag is allocated in *unified memory* since it has to be accessible from both the host and GPU. Using *cudaMemcpy* or *cudaMemcpyAsync* cannot fulfill our purpose because all CUDA calls in the same context are executed in issue order. Algorithm 4.1 presents the code of the wrapper kernel.

Figure 4.1(b) shows the procedure of stopping a kernel. When the revoke flag is set, the wrapper kernel executes `asm(trap)` to stop the running (i.e., actual) kernel. The NVIDIA driver detects that the `asm(trap)` command is called and marks the context of the issuing host process as unusable. As a result, this process can not execute any other CUDA calls. Launching a new kernel requires a new process with a new context (p2 in Fig. 4.1(b)). The process with the unusable context will be removed later from the process pool. To detect that kernel execution has stopped, we check the return value of a CUDA call. If this value is false, the wrapper kernel has called the `asm(trap)` and has stopped its execution.

Figure 4.2 shows the timing of TReM (see Section 4.4.1). To revoke a running kernel, the scheduler sets the revoke flag, and the wrapper executes `asm(trap)`), which requires 5ms. However, the next task will start its execution with an additional delay of 17ms (in total 22ms) because of the first CUDA call. As mentioned in [13], when a kernel is executed after a GPU memory pool is modified, it experiences additional overhead. Due to `asm(trap)`, the CUDA context of the issuing host process becomes unusable, clearing the CUDA context introduces a delay of 60ms. In the case of user-facing tasks, it is important to avoid this extra delay. For this purpose, we defer clearing the unusable GPU context, as discussed below.

Initiating the CUDA runtime, which includes allocating memory and executing a kernel, from a process takes approximately 15ms. We use a *pool of processes* with pre-initialized

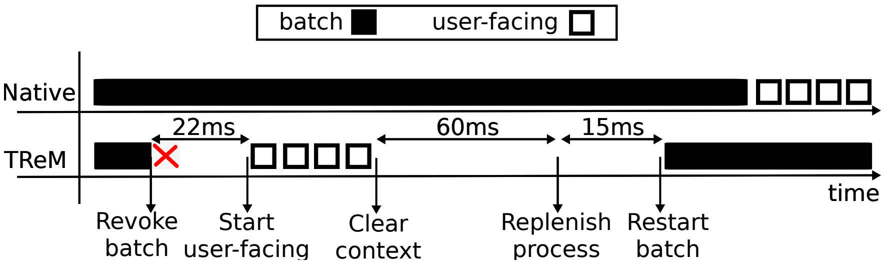
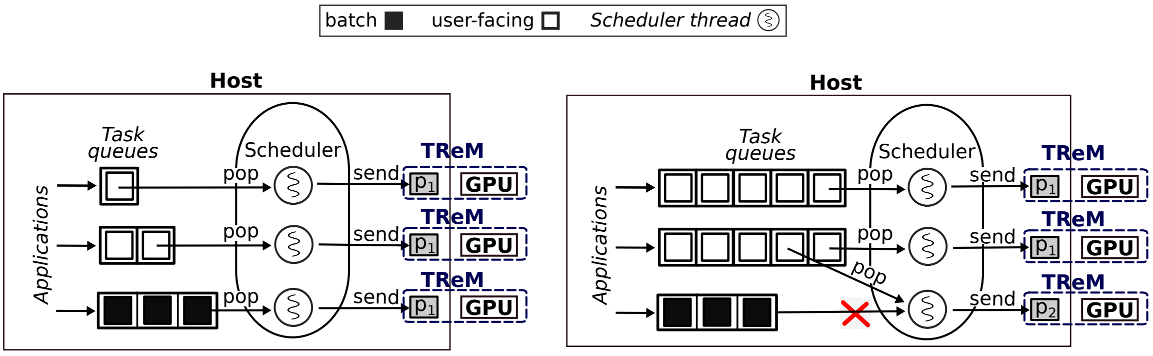


Figure 4.2: The timing of TReM compared to native execution. Batch execution time is in the range of seconds.



(a) Resource estimation is U=2.

(b) Load increases and new estimation for U=3. Scheduler instructs TReM to revoke a batch task and provide this GPU to user-facing tasks.

Figure 4.3: TReM + Elastic in multi-GPU setups.

CUDA environments to avoid this latency. We use only the pool's head process to issue task kernel (p1 in Figure 4.1(a)). When the active process becomes unusable after an `asm(trap)`, it is removed from the pool, and the next process becomes the pool head (p2 in Figure 4.1(b)), which can be used to execute kernels immediately.

After an `asm(trap)`, we have an additional unusable GPU context and one process less in the pool. We need to always have at least two processes in the pool to fulfill the case that a batch is executed and we decide to kill it and immediately serve user-facing tasks. For this purpose, we clear the unusable GPU contexts and replenish the process pool when we spawn a batch task. In this manner, the latency of these operations, 60ms + 15ms in our experiments, affects only a subsequent batch task.

TReM can handle tasks that utilize all the GPU DRAM since it does not store any state of the task that will be killed, either in the Host or GPU memory. The GPU context of the killed kernel contains useless data and will be removed when we clear the GPU context. If the available GPU DRAM is insufficient to hold the data of a new task, we immediately pay the penalty of 60ms to free all GPU DRAM.

4.2 Reducing SLA violations of user-facing tasks

In multi-GPU setups, our scheduler spawns multiple threads, one for each GPU in the system, as Figure 4.3 shows. Multiple GPUs can simultaneously serve independent tasks from the same queue (application). The mapping of task queues to GPUs is controlled by the scheduler that prioritizes user-facing tasks to reduce SLA violations. TReM runs in each GPU, and the scheduler thread mapped to that particular GPU is responsible for sending tasks to the active process and setting the revoke flag when a task has to be revoked.

Our scheduler prioritizes queues with user-facing tasks over queues with batch tasks. We implement two policies, Priority and Elastic, which differ in multi-GPU setups. In particular, Priority maps all GPUs to all task queues. For instance, when a burst of new user-facing tasks arrives, Priority will spread them to all available GPUs. On the other hand, Elastic exploits the capability to meet the SLA for user-facing tasks using a subset of the GPUs, as described in Section 4.2.1.

4.2.1 Elastic policy

Elastic prioritizes user-facing tasks over batch tasks, similar to Priority. However, it dynamically adjusts the number of GPUs allocated for user-facing tasks such that all outstanding user-facing tasks meet their SLA target. The remaining GPUs are used to serve batch tasks. We recalculate the number of GPUs for user-facing tasks every 100ms or when a task finishes.

We use the following procedure to estimate the number of GPUs needed to satisfy the current user-facing load. At time t, we first compute the maximum latency among all outstanding tasks in the system using Equation 4.1.

$$L^{max}(t) = l^{e}(t) \cdot q(t) \tag{4.1}$$

where $l^{e}(t)$ is our current estimate of the average execution time for user-facing tasks and $q(t)$ denote the number of outstanding user-facing tasks. To compute $l^{e}(t)$, we monitor previously executed user-facing tasks' execution time. Our algorithm then estimates the number of GPUs (U) needed to serve the currently outstanding user-facing tasks without violating their SLA according to the Equation 4.2.

$$U(t) = \left\lceil \frac{L^{max}(t)}{SLA} \right\rceil \tag{4.2}$$

In Figure 4.3(a), Elastic assigns two GPUs to user-facing tasks. According to its resource estimation, two GPUs (U=2) are sufficient to execute all outstanding user-facing tasks under the SLA. On the other hand, Priority will use all the GPUs to serve the three outstanding tasks and postpone the execution of the batch tasks.

4.2.2 Using TReM with Priority and Elastic

In Figure 4.3(b) the load increases, hence Elastic estimates that user-facing tasks require more resources (U=3) to meet their SLA. However, the third GPU executes a batch task. Without TReM, newly arriving user-facing tasks will wait for this batch task to finish its execution and miss their SLA. To overcome this priority inversion problem, we integrate Priority and Elastic with TReM to revoke the batch task executing in the third GPU and assign this GPU to user-facing tasks.

After the arrival of a burst of user-facing tasks, Priority will try to spread the new tasks on multiple GPUs. Being aware of the SLAs, Elastic time-multiplexes user-facing tasks on a reduced number of GPUs, as described in Section 4.2.1. Consequently, Elastic minimizes the number of revocations and, thus, the loss of useful work.

When we need to include more GPUs to serve user-facing tasks, both Priority and Elastic choose to revoke the batch task that *has started most recently*. Additionally, Elastic checks if the remaining time (predicted by the task execution time minus the task elapsed time) of the task that will be revoked is less than the task revocation latency. As a result, Elastic minimizes further the loss of useful work, compared to Priority. We use a revocation counter/threshold to avoid starvation, as explained in the Discussion section.

4.3 Experimental Methodology

In this section, we describe the platform and the workloads that we use to evaluate TReM.

4.3.1 Multi-GPU server configuration and memory affinity

The server in our testbed consists of an Intel(R) Xeon(R) CPU E5-2630 v3 running at 2.40GHz (CentOS 7). The server has four (4) NVIDIA Quadro P1000 GPU cards (Pascal architecture), with 4GB of GDDR5 and 640 CUDA cores each. We use CUDA 9.0 to implement TReM and NVSMI to measure GPU utilization. NVSMI utilization represents the time the GPU is busy and not the amount of GPU resources used.

Every P1000 GPU requires a PCIe gen3 x16 port. Our four GPU setup needs a total of 64 PCIe lanes. Our dual-socket motherboard provides 32 lanes for each socket, therefore we attach two GPUs in each socket.

In a multi-GPU configuration, there are significant memory *affinity issues*: the throughput of memory transfers depends on whether the path connecting the memory with the target GPU pass-through the QPI bus between the two sockets. Using microbenchmarks, we find that the throughput of transfers to different GPUs can interfere with each other, degrading performance from 2x up to 4x. We enforce each host thread (Fig. 4.3) to run in the same socket with its corresponding GPU to eliminate this issue.

4.3.2 Workloads

We evaluate our system using batch and user-facing tasks with execution times reported in Table 4.2. The tasks used in our evaluation originate from the Rodinia3.2 benchmark suite [17] and NVIDIA SDK of CUDA toolkit 9. The execution time is the interval between when a task is issued and the issuer receives the result. Thus it includes the transfers to and from the accelerator and the execution time of its kernel(s) on the GPU. The execution time of batch tasks ranges from tens of seconds up to two minutes, whereas the execution time of user-facing tasks ranges from 1 up to 170ms. We consider the SLA for user-facing tasks 200ms, as in Baymax [18]. The response time contains queuing delay implied from other outstanding tasks in the system.

Tasks	**Average task execution time (ms)**	**Memory footprint (MB)**	**Task type**
Euclid	10	24	user-facing
Particle Filter	23	12	user-facing
NW	38	44	user-facing
BFS	50	48	user-facing
Black&Scholes	60	112	user-facing
Pathfinder	68	74	user-facing
Hot Spot 3D	81	32	user-facing
Monte Carlo	150	68	user-facing
Darkgray	170	200	user-facing
Lava MD	46000	1069	batch
Hot Spot	130696	423	batch
Gaussian	311000	1120	batch

Table 4.2: Average task execution time (ms).

Each user-facing task can have multiple CUDA kernels, as in machine learning inference stages, with the corresponding input and output data. On the other hand, every batch task consists of a single large kernel because this stresses the scheduler more.

In our evaluation we use the following workloads:

- Micro-benchmarks, with a few tasks, to measure the responsiveness and the overheads of our mechanisms.
- A datacenter-inspired synthetic workload, with thousands of tasks, mixing user-facing with batch jobs.

To generate the datacenter-inspired workload mix, we implement a workload generator that mimics traces from Google [83] and Alibaba [59]. Our workload generator takes three parameters; (a) the job duration, (b) the job inter-arrival time, and (c) the user-facing to batch job ratio.

After analyzing Google and Alibaba traces, we found that job duration follows a Pareto distribution. Consequently, we first choose the mean value of the job duration to generate a job. Mean values for batch and user-facing jobs are presented in Table 4.3. The mean values for user-facing and batch job duration is again extracted from the traces above. After selecting the job duration, we choose the task type and number. Each application/job consists of identical tasks, taken randomly from Table 4.2.

Job inter-arrival time follows an exponential distribution, with a base mean value selected to utilize all four GPUs fully. We use a scaling factor on the base inter-arrival mean value ranging from 0.25 to 2.0 to emulate different loads. Effectively, the scaling factor modifies the density of job arrivals affecting the workload's load and burstiness. As a result, we can generate from *low* load, with job inter-arrival 2 (named load 0.25), to *oversubscription*, with job inter-arrival 0.25 (named load 2).

For the ratio between user-facing and batch jobs, we use again values extracted from Alibaba and Google traces. We use 80:20 (according to Google), which means that 80% of jobs are user-facing and the other 20% are batch. Also, we use 50:50 ratio (according to Alibaba), which implies that the user-facing jobs percentage equals batch.

	Workloads	
Workload specification	**W1**	**W2**
User-facing to batch ratio	80-20	50-50
Total Num of Jobs	30	30
Mean user-facing job duration (s)	5	5
Mean batch job duration (s)	600	600
AVG number of task/configuration	1560	1420
AVG experiment duration/configuration (h)	1.1	1.5
Configurations	5	5

Table 4.3: Workload configurations.

We limit the number of outstanding tasks from the same job to eight tasks [18]. We have selected this value empirically to ensure that most tasks in a user-facing job will meet their SLA *if run alone*. Multiple user-facing jobs will be present concurrently at runtime, increasing the system load, as described above. The response time we report and compare with the SLA counts only for outstanding tasks. Table 4.3 summarizes the workloads used for our evaluation. We repeat each experiment five times using different random seeds for each distribution.

4.4 Experimental evaluation

In this section, we first present the overheads of TReM and then evaluate the effectiveness of TReM to improve the QoS of user-facing tasks in the presence of long-running batch tasks.

4.4.1 Overhead of TReM revocation

Duration breakdown of TReM: As shown in Figure 4.2, there are multiple operations involved when deciding to kill a task. Two of them are in the critical path, affecting the latency of waiting for user-facing tasks: kill a task and start a new task using a process from the process pool. Figure 4.4 shows that these two operations require 5ms (kill) and 17ms (new task), for a total of 22ms. By breaking down the ”start new task” operation, we find that 17ms are spent in the first CUDA call, even though we have pre-initialized the CUDA runtime. Note that during normal conditions, i.e., not after a kill operation, this CUDA call typically takes less than 2ms.

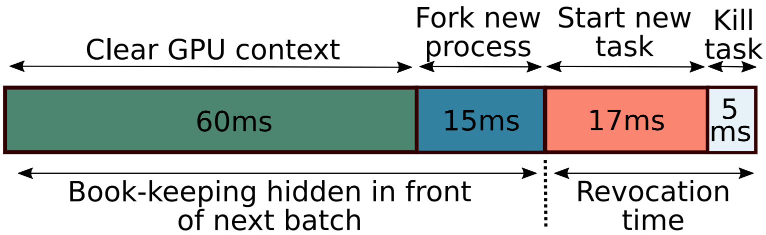


Figure 4.4: TReM overhead breakdown.

Killing the currently executing kernel requires 5ms. In particular, the revoke flag is allocated in the unified memory and is set by the host process and read by the GPU. A page fault occurs when either side accesses a page that is not resident to its memory. The memory system holding the requested page will unmap it from its page table, and the page will be migrated to the faulting process. In our measurements, the process above has almost 1.2 millisecond latency. The remaining time (i.e., 3.8ms) is due to the CUDA call that we use to detect that the wrapper kernel has stopped (i.e., called `asm(trap)`) by checking its return value. As a result, the 22ms revocation overhead is dominated by the CUDA runtime.

TReM needs to clear the GPU context, which costs another 60ms, and replenish the process pool, which requires 15ms. These 15ms are spent creating a new process and executing a CUDA call to warm up the CUDA runtime, creating a context in the GPU for the new process. However, as discussed already, TReM hides the latency of these last two operations by invoking them before a new batch task starts.

TReM does not incur overhead for non-revoked tasks: TReM does not add any new code to executing kernels; thus, it does not introduce overhead during task execution. To verify this, we run each task in Table 4.2 one thousand times with and without TReM. The results are virtually the same with less than 100μs discrepancies, which is at most 0.01% of the execution time.

TReM consumes less than 1% of cores on state-of-the-art GPUs: The wrapper kernel used by TReM to execute the actual kernel is spawned with one thread block using one thread. The NVIDIA runtime starts the wrapper kernel in a warp (32 CUDA cores). In our evaluation, Pascal P1000 has 640 CUDA cores; hence TReM requires 5% of NVIDIA P1000's CUDA cores. However, the percentage of resources consumed by TReM decreases to 0.625% with more recent GPUs, such as Volta that provide thousands of CUDA cores i.e., V100 has 5120 CUDA cores.

Resolving priority inversion: Figure 4.5 depicts the normalized response time of different user-facing tasks over their standalone execution when they time-share a GPU with long-running batch tasks. In this experiment, we first start a batch task, whose duration varies along the x-axis, and we record the latency of a subsequent user-facing task.

Without TReM, the response time of user-facing tasks increases linearly with batch task execution time (Figure 4.5(a)). In particular, it increases two orders of magnitude higher than the task execution time. On the contrary, TReM results in constant response time for all user-facing tasks, independent of the batch task duration (Figure 4.5(b)). Their execution time is at most 3x the execution time of the standalone user-facing task in our experiments.

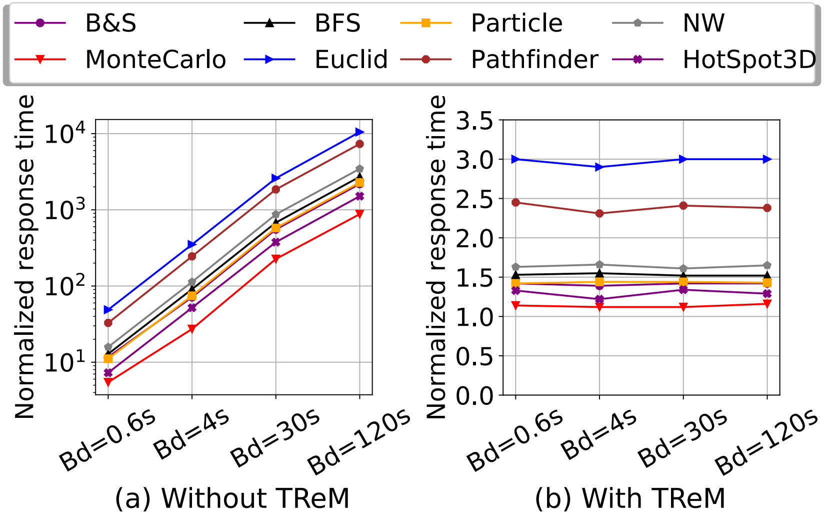


Figure 4.5: Normalized response time of user-facing tasks over their stand-alone execution in the presence of batch tasks with different duration (Bd).

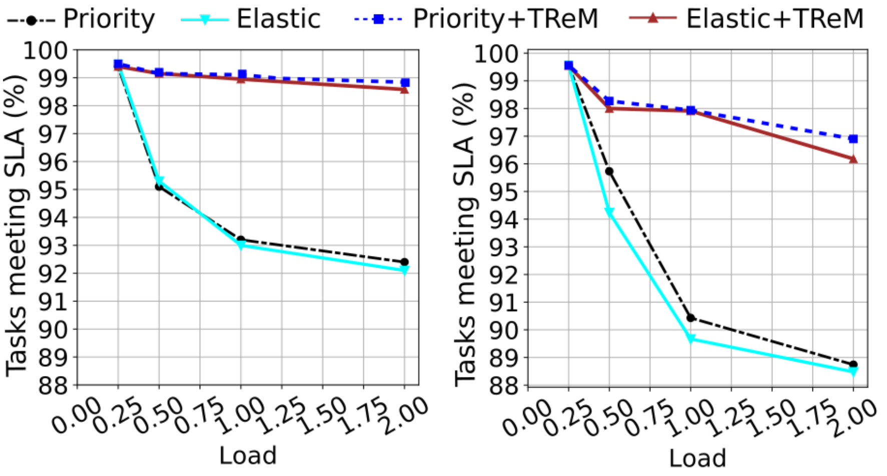
4.4.2 Effectiveness of TReM with long-running batch tasks

SLA violations: We use workloads W1 and W2 to examine how TReM reduces SLA violations. Figure 4.6 shows the percentage of user-facing tasks that meet their SLA (200ms) with increasing load. As mentioned, W1 and W2 differ in the ratio of user-facing to batch jobs.

The x-axis is the incoming load, from low (0.25) to high (1.0) and oversubscribed (2.0). A load of 0.25 suffices to fully utilize one GPU (i.e., job inter-arrival 2). A load of 1.0 can fully utilize all four GPUs (i.e., job inter-arrival 1). A load of 2.0 over-subscribes our system by 2x (i.e. job inter-arrival 0.25). As we see, at low load, more than 99% of the tasks meet their SLA, irrespectively of the policy (Priority, Elastic), and the workload (W1, W2).

At a higher load, we see that W1 (Figure7(a)), with 50:50 ratio of user-facing to batch jobs, incurs more violations, and the efficiency of Priority and Elastic drops to 93% at load 1.0 and 92% at 2.0. On the other hand, using TReM, both policies can tolerate the load increase with a much lower impact on efficiency, meeting the SLA for 99% of tasks at load 1.0 and 98% at load 2.0.

If we increase the number of user-facing tasks, using a ratio of 80:20 in W2 instead of 50:50 in W1, the advantages of TReM are more pronounced (Figure 7(b)). Without TReM,



(a) W1: 50% user-facing – 50% batch (b) W2: 80% user-facing – 20% batch

Figure 4.6: Percentage of tasks that meet their SLA (y-axis) at increasing GPU load (x-axis), for workloads W1 (left) and W2 (right).

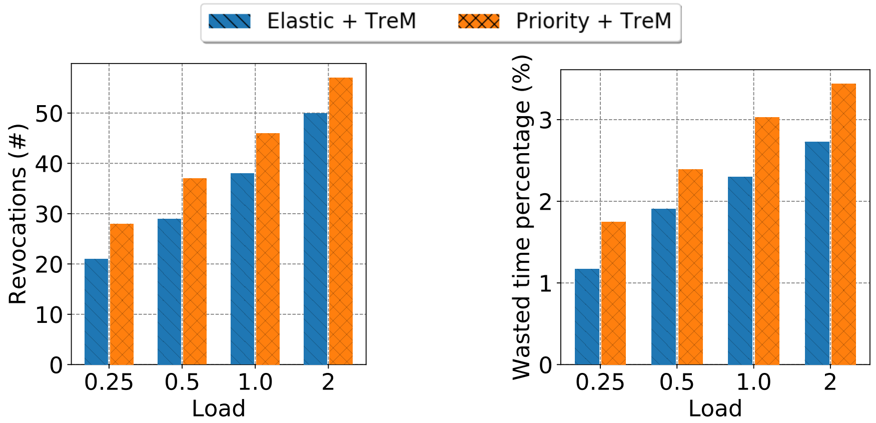
Priority and Elastic meet the SLA for 90% of the tasks at load 1.0, whereas, using TReM, we achieve 98% at load 1.0 and 96% at load 2.0. Therefore, *fast revocations is an effective ingredient to maintain the SLA in the presence of long-running batch tasks.*

At load 1.0, both Priority+TReM and Elastic+TReM achieve 89% GPU utilization. Priority and Elastic achieve 91% GPU utilization but with much more SLA violations.

Comparing Priority with Elastic, we can observe that Elastic leads to less than 0.6% more SLA violations on average compared to Priority. This is expected because Priority utilizes all GPUs to handle the user-facing load, resulting in significantly more revocations, as discussed next. On the other hand, Elastic aims to decrease the number of revocations without increasing SLA violations.

Lost work due to revocations: Figure 4.7(a) depicts the overhead of task revocations using TReM. We see that Elastic+TReM performs fewer revocations compared to Priority+TReM. Elastic uses a minimum number of GPUs to satisfy the SLA, packing when possible multiple user-facing task on the same GPU. Effectively, it triggers considerably fewer revocations. Under low load i.e., 0.25, Priority performs 33% more revocations than Elastic, 27% at load 0.5, and 14% at load 2.0. As the load increases, Elastic requires the same number of GPUs as Priority, thus the difference in revocations diminishes.

Figure 4.7(b) shows the percentage of lost work due to revocations. We measure total



(a) Number of revocations

(b) Wasted compute time

Figure 4.7: Revocations overhead: (a) Number of task revocations; (b) Wasted compute time due to revocations.

lost work *as the elapsed time between the start and kill for each task*. The percentage of lost work is computed as the ratio of the total work discarded over the total useful computation time in the workload. At low load (0.25), the wasted time percentage for both Elastic and Priority is below 2%, while at load 2.0 it reaches 3%. Both policies minimize wasted work by preferring to revoke the most recently started batch tasks. As discussed previously, Elastic outperforms Priority because it packs user-facing tasks in the same GPU and also because it will revoke a batch task only if its remaining time is less than the revocation overhead. Priority does not have the notion of SLA; it just prioritizes user-facing over batch tasks, hence it cannot measure the task remaining time.

Batch job duration percentiles: To examine in more detail the effect of our policies on batch jobs, Figure 4.8 depicts time to completion for batch jobs. As expected, time to completion of batch jobs increases with TReM because tasks are revoked and replayed. We should note, however, that without TReM, these batch jobs would typically wait for all user-facing tasks to complete or would need to execute on additional GPUs. Elastic reduces the impact on completion time from 4% up to 50% compared to Priority. In particular, for 50% of the jobs, Priority+TReM has 1.6x higher time to completion than Priority, whereas Elastic+TReM exhibits only 1.3x increase compared to Elastic. The effect of TReM becomes more pronounced on higher percentiles of batch jobs, and reaches 3.2x for Priority and 2.1x for Elastic.

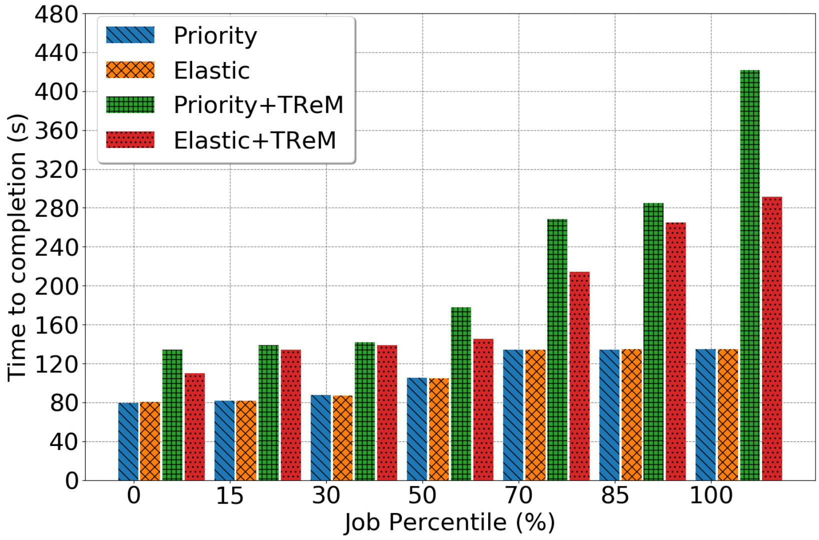


Figure 4.8: Time to completion for batch jobs under different scheduling policies, for load 2.0.

Dynamic GPU partitioning with Elastic: Figure 4.9 shows how Elastic partitions the number of GPUs between user-facing and batch tasks over time. The upper part of the figure shows the number of GPUs allocated to user-facing tasks and the number of GPUs used to run batch tasks. The lower part of the figure depicts the actual latency of user-facing tasks over time. When user-facing tasks arrive (red crosses in the lower part), Elastic allocates more GPUs to user-facing tasks to avoid SLA violations. We see that Elastic accurately estimates the GPU requirements of user-facing tasks.

4.4.3 Scalability of TReM

We implement a simulator using the observed latencies to evaluate our system with more than four GPUs and different revocation latencies. Our simulator models our policies (Priority, Elastic) and TReM without modeling the GPU internals. It takes as parameters (1) the task execution time, reported in Table 4.2, (2) the workloads, described in Table 4.3, and (3) the revocation latency. The simulator runs W1 and W2 with the timings provided from Table 4.2.

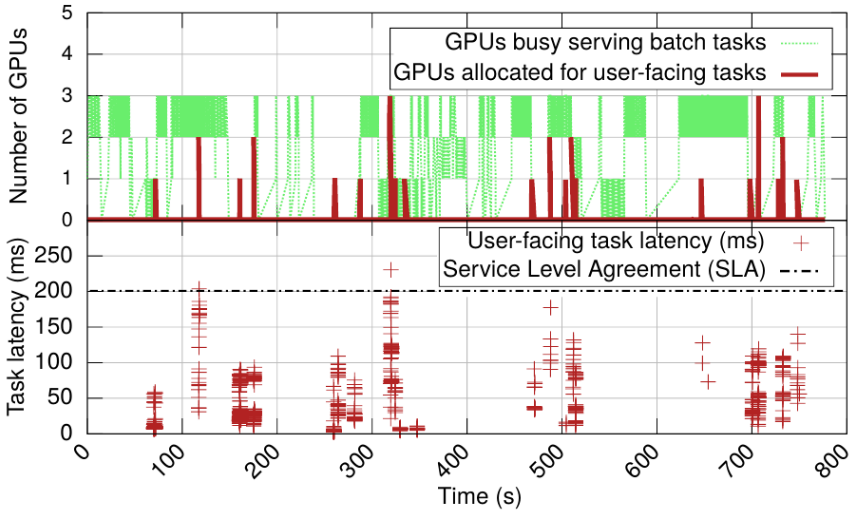
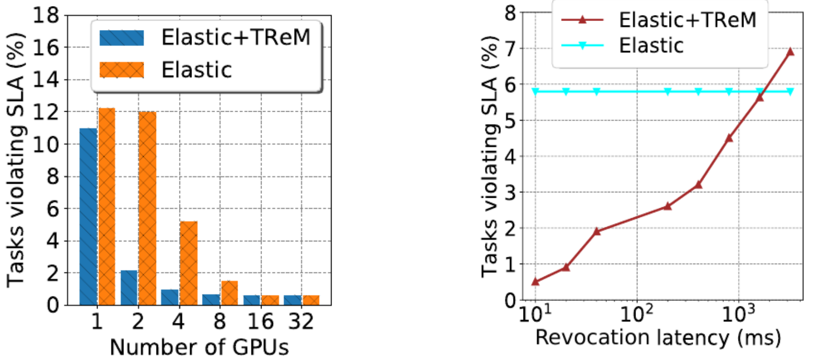


Figure 4.9: Dynamic GPU allocation in Elastic and impact on SLA violations.

Simulation results are a superset of our testbed results in terms of; (a) the number of GPUs and (b) revocation latency. Moreover, the simulator and the testbed results are very close. In particular, the violations of our simulator results for four GPUs and load 1.0 are 1% (Figure 4.10), while for the testbed, the violations are 1.3% (Figure 4.6). Consequently, the difference between the simulator and the testbed results is 0.3%, when trends between policies (in Figure 4.6 with and without TReM) are in the order of 10%.

In Figure 4.10, the simulator runs the datacenter workloads, W1 and W2, for load 1.0. and reports their average results. Figure 4.10(a) shows task violations for Elastic and Elastic+TReM with load 1.0 and *a varying number of GPUs*. The positive effects of TReM are more pronounced with 2-8 GPUs. We see that using TReM, we achieve less than 1% violations with 4 GPUs, whereas *without TReM we need 16 (4x) GPUs to achieve the same target.*

Figure 4.10(b) evaluates the percentage of violations with varying revocation latency, between 10 and 1000ms. The percentage of violations increases proportionally with revocation latency. Consequently, other mechanisms such as `asm(exit)` and process kill (Section 4.1) that introduce more latency can meet SLA for only 94% of user-facing tasks. To ensure SLA for more than 99% of user-facing tasks, we require a revocation mechanism with 10ms latency.



(a) Varying number of GPUs

(b) Varying revocation latency

Figure 4.10: SLA violations for W1 and W2 under load 1.0; (a) varying the number of GPUs (revocation latency 22ms); (b) varying the revocation latency (4 GPUs).

4.5 Discussion

Bounding the amount of wasted work: Figure 4.7(b) shows that TReM wastes only 3% of the total work due to revocations. However, it is possible that TReM kills a batch job (i.e., training in Machine Learning) after minutes or even hours of computation. In this case, the percentage of lost work can increase significantly. By incorporating a checkpoint mechanism, as GPU Snapshot [52] or Gandiva [16], to TReM, we can bound the wasted work at a single checkpoint interval. Both checkpoint mechanisms do not require changes in the kernel code; hence they can be integrated with TReM. Additionally, the checkpointing overhead for both approaches is less than 100ms; hence the increase in SLA violations will be minimal.

Repeated revocations & starvation: TReM revokes the batch task that has performed the least work so far. If, however, user-facing tasks arrive periodically (e.g., a user-facing arrives every time a batch task has just started), TReM may starve a batch task by killing it repeatedly. To avoid this, TReM can maintain a revocation counter and inform the cluster scheduler to reassign the load across servers when the revocation counter exceeds a certain value.

CUDA streams: TReM uses `asm(trap)` to revoke a kernel. Effectively, TReM destroys the CUDA context of the issuing process. With CUDA streams, all concurrently running kernels belong to a single CUDA context. In case user-facing kernels run concurrently with batch, user-facing kernels will be revoked as well. TReM+Elastic addresses this by allowing only kernels of the same type, i.e., only batch or user-facing, to run concurrently in a GPU.

4.6 Summary

This chapter presents TReM, a mechanism that revokes batch tasks running in a GPU and starts the next task within 22ms. TReM, in contrast to previous approaches, can revoke a task at any point of its execution using CUDA dynamic parallelism. TReM does not require kernel source code and is supported by almost all NVIDIA GPUs, except the outdated Fermi architecture that does not support dynamic parallelism. We implement two scheduling policies, Priority and Elastic, that aim to meet SLA for user-facing tasks when sharing a GPU with long-running batch tasks. We then use TReM to enhance both policies and reduce SLA violations.

We evaluate TReM with two workloads derived from real datacenter traces. We show that Priority+TReM and Elastic+TReM ensure SLAs for 98% of user-facing tasks when collocated with long-running batch tasks at high load with 89% GPU utilization. On the contrary, scheduling policies Priority and Elastic (without TReM) ensure SLA for 88.8% of tasks under high load. Additionally, Elastic+TReM and Priority+TReM bounds the percentage of lost work to 2.5% and 3.5%, respectively. Finally, Elastic+TReM reduces the number of revocations and the corresponding loss of useful work compared to Priority+TReM by 1%.

Chapter 5

Secure GPU Spatial Sharing

5.1 Introduction

Arax and other approaches [110, 20, 115, 79, 31, 104, 112, 103, 57] share a GPU spatially to applications from different users to improve GPU utilization. However, all these spatial sharing mechanisms introduce a significant concern: GPU kernels execute in the same GPU address space, hence they can modify (either inadvertently or deliberately). This chapter describes Guardian, a PTX-based bounds checking approach that provides transparent memory protection for spatial GPU sharing. Guardian prohibits applications from directly making GPU calls. Instead, their calls are dynamically intercepted and forwarded to a trusted process that has exclusive access to the GPU and performs any necessary checks. Guardian is completely transparent to ML frameworks [78, 41, 1] without requiring any source code modifications, compilation, or extra hardware. Internally, Guardian divides the GPU memory into partitions, which are assigned to different applications. At runtime, the instrumented GPU kernels at the PTX level check every load and store instructions using address fencing or address checking mechanisms. Guardian effectively addresses three main challenges, as follows.

Intercept GPU calls from closed-source libraries. Guardian intercepts all GPU calls transparently at the CUDA runtime and driver library level by dynamically preloading the execution of the applications. Previous API remoting approaches intercept only the top of the CUDA runtime, driver, and accelerated libraries stack [98, 24, 23, 28, 113, 79]. This is not sufficient for Guardian, though, mainly because implicit CUDA calls performed from high-level CUDA library functions (e.g., `cublasIsamax()`) will go unprotected. Guardian forwards the intercepted GPU calls to the GPU manager, which runs as a separate process and is the only entity with GPU access. This allows the GPU manager to securely manage and execute the GPU calls of different applications on a shared GPU.

Fence illegal host and device accesses. Application host memory is inherently protected because Guardian applications run as different processes. This is not the case for the device code, which run on the same GPU context. To isolate the GPU address spaces of co-running applications, Guardian uses a custom allocator that divides the GPU memory into logical partitions assigned to applications. The allocation calls of each application are served from its assigned partition, and every host-initiated transfer is checked at run-time to verify that it falls in a valid range. In addition, Guardian extracts and instruments the virtual assembly version of GPU kernels (PTX), which are available even in closed-source libraries. The instrumentation includes the insertion of run-time checks to ensure that each pointer always falls in the valid range upon dereference.

Lightweight bounds checking. Address checking is a popular method for memory bounds protection, but it is costly because of the metadata management and the run-time checks. Reading the bounds from memory and inspecting if the pointer falls within these bounds incurs significant overheads [47]. To overcome these costs, Guardian follows a twofold approach. *First*, it uses contiguous partitions for each application, different from previous works [115]. This eliminates the need to store metadata for each allocation; instead, Guardian keeps only the start offset and the size of each partition, which can be stored in registers to avoid excessive memory fetches. These registers can be reused without adding significant register pressure to the execution of the GPU kernel. *Second*, it aligns the partitions in power-of-two sizes. This allows Guardian to optimize math operations (i.e., modulo) using fast bitwise operations. In fact, Guardian adds only two bitwise instructions per load and store, to isolate the memory partitions of different applications.

We implement Guardian for NVIDIA GPUs and evaluate it with several micro-benchmarks and real-life ML applications, such as Caffe and PyTorch, that link with closed-source GPU libraries. For Caffe and PyTorch, which invoke billions of GPU kernels, Guardian address fencing has on average 9% overhead compared to native unprotected execution, whereas address checking is 1.7× worse than native. Guardian (protected) spatial sharing is 4.84% slower than MPS (unprotected). At the same time, it improves the total execution time of co-located applications by 37% compared to time-sharing, which is the alternative sharing and protection mechanism used from other systems [108, 48, 113]. Finally, Guardian imposes minimal increase in register usage, and thus register spilling occurs only in 0.9% of PyTorch kernels.

The main contributions of this work are:

- We design, implement, and evaluate Guardian, a novel system that offers *transparent* memory protection for applications executing concurrently on a GPU without relying on hardware support nor the existence of source code. We demonstrate its effectiveness using a broad range of kernels and complex, real-life ML frameworks [41, 78] that extensively use closed-source GPU libraries.

- We present a mechanism to intercept all GPU-related calls only at the CUDA runtime and driver library level. This allows transparently tracing and monitoring any GPU application or closed-source GPU library.

- We evaluate different bounds checking mechanisms implemented at the PTX-level and conclude that address fencing with bitwise operations is highly efficient and practical for protected GPU sharing. Compared to CPUs, GPU bounds checking has lower overhead because kernels have simpler access patterns.

5.2 Background

In this section, we discuss three aspects of GPUs that are related to our work: (i) The programming interfaces and context of GPUs, (ii) the compilation workflow, (iii) the GPU memory-sharing scope, and (iv) the supported addressing modes. Although we discuss these issues for NVIDIA GPUs, we believe that AMD and Intel GPUs have similar architectural characteristics [51].

5.2.1 GPU Programming Interfaces and Context

CUDA includes two programming interfaces for accessing and managing GPUs: the driver and the runtime, provided by two separate libraries the `libcuda.so, libcudart.so`. The runtime library is built on top of the driver interface and is complementary to it. CUDA applications use both domain-specific GPU libraries (CUDA-accelerated libraries), such as cuBLAS and cuDNN, and directly access the GPU via the runtime and driver libraries. Domain specific libraries, typically use the GPU via the CUDA runtime library. Applications use the runtime interface since it offers similar abstractions to the driver interface but at a somewhat higher level, requiring less effort from the programmer. In particular, the driver interface requires explicit management of contexts, modules, and functions loaded in the GPU, while these operations are performed implicitly by the runtime library.

All actions performed from the CUDA runtime and driver interfaces, including memory allocation, data transfers, and kernel invocation (launch), are encapsulated in a CUDA context on the GPU itself. A GPU context is similar to a CPU process. As such, each CUDA application creates its own context during the first CUDA runtime call. The context contains all the information regarding the resources used by an application, such as GPU memory, streams, cores, page table, and the GPU kernels to be used. An application that executes in a context cannot access memory locations used by an application in a different context. At any point in time, the GPU can execute multiple kernels on different streams, however, all kernels must belong to a single context. The GPU allows different contexts to time-share resources using a context switching mechanism. GPUs support preemption, in which case the state of a GPU context is swapped to GPU DRAM so that another context can be swapped in and run. However, context switching does not allow different applications to spatially share the same GPU.

5.2.2 GPU Compilation Workflow

CUDA applications consist of host-level (`.cpp`) and device-level source code (`.cu`) [76]. The host source code is compiled with `clang` or `gcc`, while the device code is with `nvcc` [107, 33]. As Figure 5.1 shows, the device source code is converted to the compiler Intermediate

Representation (IR) format, which is then compiled, via cicc, to Parallel Thread eXecution (PTX) [71] assembly.

PTX is a virtual assembly specification supported by the NVIDIA toolchain in all NVIDIA GPU architectures, past and future [81]. If necessary, the CUDA driver compiles the PTX code at runtime using just-in-time compilation and sends it to the new target device for execution [101]. This allows the generation of forward-compatible optimized machine code that runs on the target device. CUDA provides CUDA_FORCE_PTX_JIT environment variable, which enforces the PTX code to be JIT compiled. Thus, the driver ignores any cuBIN files embedded in an application or CUDA library.

Nvcc always embeds the PTX representation of the device code in the target applications or libraries. Additionally, nvcc also generates machine code for specific GPU architectures (using the ptxas assembler) and embeds this binary code to the application executable in the form of cuBIN files [101]. The developer can specify during compilation (sm flag) the target architectures for which cuBIN files should be generated and included in the target application. The generated PTX code and the cuBIN files are merged in a fatBIN file.

CUDA closed-source libraries contain all GPU kernels in PTX and cuBIN files. As shown in Table 5.1, a CUDA library of a particular CUDA version contains the kernels in PTX format for the most recent GPU architecture and for all previous architectures the kernel code in cuBIN files. For instance, CUDA 11.7.1 is the most recent CUDA SDK version for Ampere architecture, hence, CUDA libraries of this CUDA version contain the cuBIN files for all previous architectures (Turing) and PTX to support Ampere and Hopper.

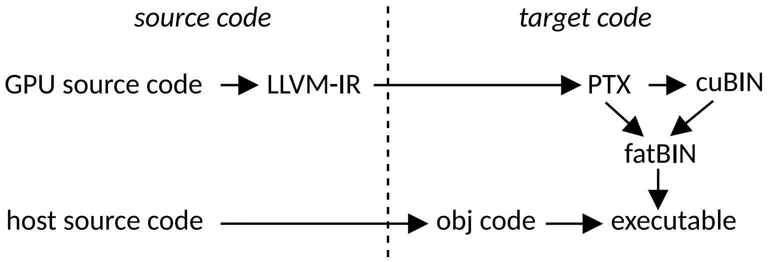


Figure 5.1: Compilation flow of CUDA applications.

CUDA	NVIDIA GPU architecture		
version	Turing (7.5)	Ampere (8.0-8.7)	Hopper (9.0)
10.0-10.2	PTX		
11.0-11.7.1	cuBIN	PTX	
11.8-12.0	cuBIN	cuBIN	PTX

Table 5.1: cuBIN and PTX kernel code included in CUDA-accelerated libraries for different CUDA versions and GPUs.

5.2.3 GPU Memory Sharing Scope

Figure 5.2 shows an overview of the GPU memory hierarchy and the sharing scope. GPU memory is divided into *on-chip* and *off-chip*. The only memories that can be accessed from co-running kernels are located off-chip [51, 115, 21].

On-chip memory consists of the register files and the scratchpad memory, called shared memory. The registers are privately allocated for each thread and cannot be accessed from other threads, even if located in the same Streaming Multiprocessor (SM). Shared memory is accessible only from threads of the same SM.

Off-chip DRAM is divided in local, heap, global, constant, and texture memory for NVIDIA GPUs. The local memory (stack) is located off-chip only if the variables used from a kernel exceed the number of available registers (i.e., register spilling); compilers aim to avoid register spilling since it degrades performance.

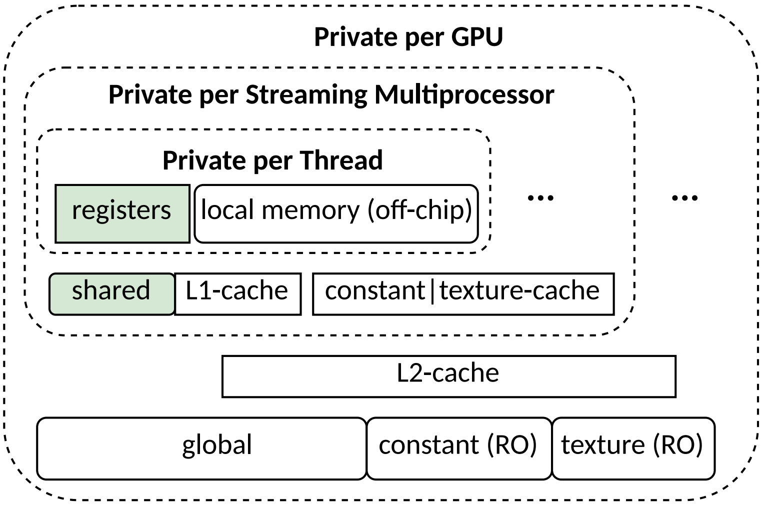


Figure 5.2: NVIDIA GPU memory hierarchy and sharing scope.

Heap memory is allocated and deallocated by kernels using `malloc` and `free` and is not accessible through host-side CUDA calls, e.g., `cudaMemcpy`. However, heap memory is rarely used because in-kernel allocations imply large overheads, up to 63× [51] compared to allocations in global memory using `cudaMalloc`.

Global memory can be accessed from any kernel of the same GPU context, either for read or write operations. Global memory is managed dynamically from the host (e.g., using `cudaMalloc` and `cudaFree` functions), or statically from the device (using the `.global` keyword). A CUDA kernel uses load and store instructions to access data in global memory.

Unified memory is introduced to reduce the programmer's burden to transfer data from/to the host memory explicitly. Instead, data is transferred automatically from the host memory in page granularity by the page fault handler in the GPU driver [51] and the IOMMU [8]. CUDA kernels access such data using the same load and store instructions as if data were in global memory.

Constant memory is a predefined read-only part of the global memory space and uses a separate cache inside each SM. Constant memory is allocated statically using `__constant__`. Texture memory is also cached, read-only, off-chip memory. Texture memory is accessible via objects created and destroyed from the host, using `cudaCreateTextureObject` and `cudaDestroyTextureObject`, respectively. However, we find that the use of these memories is extremely rare in ML applications, hence we can ignore it for protection purposes.

```
// Method A. Full virtual address
ld.global/shared/local %val, [base_addr];
st.global/shared/local [%base_addr], %val;

//Method B. Base address + offset
ld.global/shared/local %val, [base_addr+offset];
st.global/shared/local [%base_addr+offset], %val;
```

Listing 5.1: GPU memory addressing modes.

5.2.4 Addressing Modes

Modern GPUs provide two addressing modes for loading or storing data to memory [71], as shown in Listing 5.1. In the first case, the base address is loaded into the destination register, while in the second, an offset is first added to the base address, and the result is loaded into the destination register. The same modes apply to stores and to all off-chip memories. Similar to CUDA [71], AMD HIP [73] supports these two addressing modes. Constant objects are statically defined outside the kernel code and are loaded to registers using move instructions. Finally, texture objects are allocated from host-level code and are accessed using the `texref` instruction.

5.3 Threat Model

Our work considers memory safety across kernels from different applications that share a GPU spatially in the cloud or other shared environments. Guardian prohibits applications from different users to read or modify each other's data in the host or device memory. Within the realm of GPU security concerns, our primary emphasis is on memory safety, as it represents a prominent threat to spatial GPU sharing.

We consider all GPU kernels unsafe, provided by individual users or from GPU libraries. As a result, any instruction that performs loads or stores from a base address fetched from a destination register is considered unsafe and should be protected via bounds checking.

Regarding control flow, direct branch instructions are safe because they jump to labels defined inside a PTX file. The assembler will report errors if the labels are absent from the PTX file or are incorrect. On the contrary, *indirect* branch instructions (`brx.idx`) are unsafe because they use a register to index a statically defined array of labels. The register employed for indexing cannot be validated at compile time, potentially leading to out-of-bounds accesses.

Our threat model assumes that the GPU driver and the GPU device are trustworthy and reliable. Consequently, security issues related to exploiting GPU resource contention or side-channels [111], denial-of-service [66], or physical access attacks [102] are outside

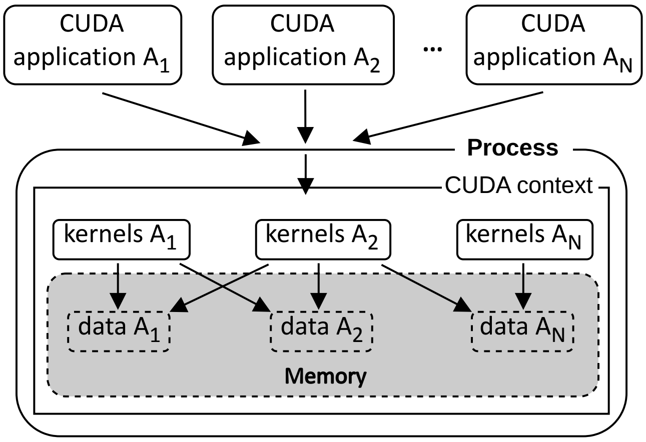


Figure 5.3: Multi-tenant spatial GPU sharing, without Guardian. The common GPU context required for spatial sharing allows applications to access each others memory.

the scope of this work.

5.4 Guardian Design

The goal of Guardian is to prevent applications of different users from reading or modifying each other's data when executing concurrently on the same GPU. Spatial GPU sharing requires a common CUDA context to execute kernels from different applications concurrently. Previous work [70, 79, 115, 28] uses a separate process that creates that single context. Applications issue all their GPU tasks to this process, which enqueues to different streams, thus kernels can be executed concurrently. However, without proper protections, this approach allows GPU kernels to modify memory locations belonging to other applications, as shown in Figure 5.3.

Guardian uses three mechanisms shown in Figure 5.4 and described in more detail below. The dynamically loadable library (§5.4.1) intercepts CUDA calls and forwards them to a trusted process, the gManager (§5.4.2) that executes GPU calls on behalf of the applications. The PTX-patcher (§5.4.3) applies bounds checking instructions (§5.4.4) to GPU kernels.

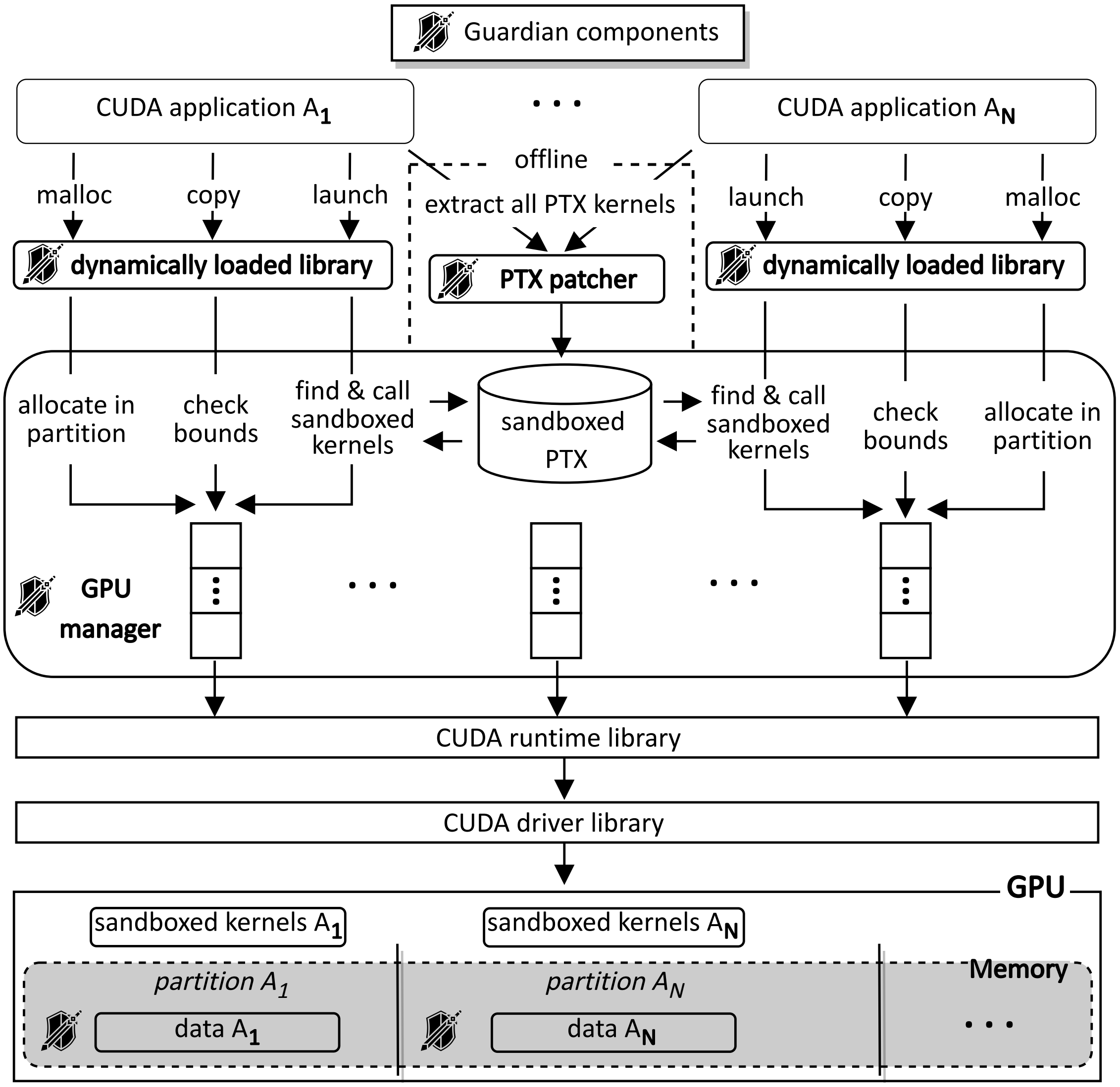


Figure 5.4: Guardian online and offline (dashed annotated) mechanisms to allow protected spatial GPU sharing. Guardian intercepts the CUDA runtime interface used from applications and perform the necessary checks at memory allocations, transfers, and kernel executions. This allows kernels from different applications to execute concurrently on different memory partitions, eliminating illegal accesses.

5.4.1 Dynamically Loadable Library

Guardian uses a dynamically linked library (i.e., gLib) that is preloaded to the applications and transposes the default CUDA runtime and driver library, as shown in Figure 5.5. These two libraries are the lowest public interfaces for providing CUDA calls to manage

GPU resources, such as allocating memory and launching kernels. CUDA applications and CUDA-accelerated libraries use the CUDA runtime interface and, to a lesser extent, the CUDA driver interface [24, 23]. The latter is used by applications only for specific, lower-level operations, such as explicit PTX (un)loading and context management.

The interception of CUDA calls is challenging mainly for two reasons. First, the functions provided by CUDA-accelerated libraries invoke several *implicit* CUDA runtime calls, including memory allocations, transfers, and kernel launches. For example, we have noticed using NVIDIA Nsight profiling tool that a single cuBLAS function, such as `cublas-Isamax()`, can invoke *implicit* several CUDA runtime calls, such as `cudaMalloc()`, `cudaMemcpy()`, and kernel launches via `cudaLaunchKernel()`. Previous work [79, 23, 24, 28] treated such library calls as a black box, which is inadequate for Guardian because implicit CUDA calls can go unprotected. Intercepting implicit calls requires applications to link with the *static* version of CUDA-accelerated libraries (e.g., `libcudblas_static.a`) since only this version uses the shared version of CUDA runtime library (i.e., `libcuda_rt.so`). We also find that CUDA libraries dynamically load the CUDA driver library using `dlopen()` instead of linking with it. To prevent the original CUDA driver library from being loaded, we intercept `dlopen()` and provide the glib.

Second, CUDA libraries use an un-documented API function, namely `cudaGetExport-Table()`, which returns tables of function pointers. The use of these functions depends on the application's need. For instance, we have found that large frameworks, such as PyTorch and Caffe, use about seven export tables containing more than 90 functions. By carefully rewriting these functions, Guardian can adequately intercept the CUDA runtime and driver libraries and run successfully real-world ML applications. The intercepted CUDA calls are forwarded to another process, the gManager (§5.4.2), which is the only entity with GPU access. This enables Guardian to securely perform any runtime checks necessary before executing the GPU operations on behalf of the applications.

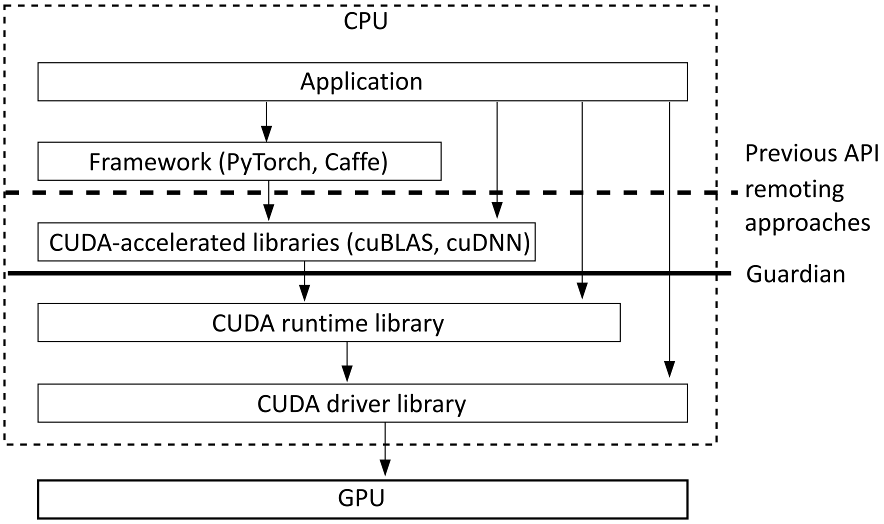

CPU
Application
Framework (PyTorch, Caffe)
CUDA-accelerated libraries (cuBLAS, cuDNN)
CUDA runtime library
CUDA driver library
GPU
Previous API remoting approaches
Guardian

Figure 5.5: Guardian CUDA library interception level versus previous approaches [24, 23, 28, 98]. Guardian intercepts only the CUDA runtime and driver APIs and *not* the high-level calls to CUDA accelerated libraries as in previous works.

5.4.2 GPU manager

The gManager allocates GPU a memory partition per application, fences memory transfers, calls the sandboxed GPU kernels, and allows multi-tenant GPU sharing using the mechanisms described below.

GPU Memory Partitioning

The virtual memory of GPUs is managed through the `cudaMalloc()`-family functions, which return arbitrary addresses upon each call. To perform GPU memory isolation across different applications, Guardian uses a custom allocator that initially reserves all GPU memory and splits it into partitions. Each partition is a contiguous memory block assigned exclusively to an application (or tenant). Contiguous memory partitions enable Guardian to offer isolation by checking that all memory accesses are always within the partition boundaries.

Guardian intercepts `cudaMalloc()` (malloc in Figure 5.4) and allocates memory in the application's partition. Similarly, our allocator marks the region in a partition as free by intercepting the `cudaFree()` function. For each application, we store the application id, the base address, and the partition size in a *partition bounds table* used at runtime. Currently, Guardian partitions GPU memory statically; hence, each application must specify its maximum memory requirements at initialization. Though this is sufficient for the applications and workloads we examine, it is interesting for future work to explore dynamic partition resizing.

Data Transfers

Data transfers include operations that move data between the host and the GPU memory (e.g., `cudaMemcpyH2D()`) or within the GPU memory (e.g., `cudaMemcpyD2D()`). Even though these calls are initiated from the host, they refer to the same GPU address space; hence, applications can still perform memory operations to partitions of other applications. gLib intercepts the memory management CUDA calls (copy in Figure 5.4) and uses the *partition bounds table* to verify that the memory ranges are within the correct partition. Guardian allows a transfer to complete if the destination start and end addresses are within the allocated partition. For `cudaMemcpyH2D()`, we check the destination pointer; for `cudaMemcpyD2H()`, we check the source pointer; and for `cudaMemcpyD2D()` we check both.

GPU Kernel Invocation

The gManager creates a new `CUmodule` for each PTX exported and patched during the offline phase (§5.4.3). A `CUmodule` is a CUDA code (PTX or cuBIN) unit that can be dynamically loaded and executed on the GPU. The `CUmodules` are then loaded into the current context using `cuModuleLoadData()`. A `CUmodule` can contain more than one kernel; hence, we use `cuModuleGetFunction()` to create a `CUfunction` handle for each kernel. The `CUfunction` handles are stored in a map, called *pointerToSymbol*, used to locate the appropriate kernel for execution.

At runtime, Guardian intercepts each kernel invocation via `cudaLaunchKernel()` and executes the corresponding sandboxed kernel instead, as shown in Figure 5.4. Every time a kernel is invoked for execution (through `cudaLaunchKernel()`), Guardian performs a lookup at the *pointerToSymbol* table to find the `CUfunction` handle of the corresponding sandboxed kernel. Then, it adjusts the number of parameters accordingly; for address

```
__global__ void kernel(int *A, int j){
  int tid = threadIdx.x;
  A[i] = j;
}
```

Listing 5.2: Sample CUDA kernel source code.

fencing (bitwise operation), it passes the mask and the base partition address (§5.4.3), whereas for address checking, the partition base and ending addresses. Each partition's information (base address, mask, or end address) is retrieved through the *partition bounds table*. Finally, the gManager issues the sandboxed kernel using `cuLaunchKernel()`. When the gManager detects that an application runs standalone, it issues a native kernel, avoiding the overhead implied by the extra instructions.

Spatial Multiplexing

To enable spatial sharing, GPUs require a single context and CUDA streams provided by the gManager, similar to previous works [115, 79, 70]. As a result, applications do not create their own context; instead, they funnel their work to the GPU through the context of the gManager. All CUDA kernels and data transfers originating from a single application will be executed in-order from the gManager. In contrast, kernels and data transfers from different applications will be executed concurrently using different streams. Applications and the gManager run in different address spaces; thus, we use an IPC channel and a separate shared memory segment to exchange operations and data similar to other API remoting approaches [115, 67, 79, 28]. Although we implement our GPU manager, Guardian can be integrated into others [115, 31] if their source code is available.

5.4.3 Offline Kernel Sandboxing

The PTX-patcher is a python script that uses `cuobjdump` [39] to extract any embedded PTX kernel from the application executable and the CUDA libraries (offline in Figure 5.4). The extracted PTX kernels are then sandboxed to ensure they do not access data outside the correct partition boundaries. Listing 5.3 shows the sandboxed PTX code of the original kernel shown in Listing 5.2. The original PTX (without sandboxing) consists of a kernel

```
1   .visible .entry kernel(
2   .param .u64 kernel_param_0,
3   .param .u32 kernel_param_1,
4   // Base address
5   .param .u64 kernel_base,
6   // Mask parameter
7   .param .u64 kernel_mask)
8   {
9     .reg .b32       %r<3>;
10    .reg .b64       %rd<5>;
11    ld.param.u64    %rd1, [kernel_param_0];
12    ld.param.u32    %r1, [kernel_param_1];
13
14    // Extra registers for base and mask
15    .reg .b64       %grdreg<3>;
16    // Load extra parameters to registers
17    ld.param.u64    %grdreg1, [kernel_base];
18    ld.param.u64    %grdreg2, [kernel_mask];
19
20    cvta.to.global.u64      %rd2, %rd1;
21    mov.u32         %r2, %tid.x;
22    mul.wide.s32    %rd3, %r1, 4;
23    add.s64         %rd4, %rd2, %rd3;
24
25    // Bit-wise And with mask
26    and.b64         %rd4, %rd4, %grdreg2;
27    // Bit-wise OR with base addr.
28    or.b64         %rd4, %rd4, %grdreg1;
29
30    st.global.u32   [%rd4], %r2;
31    ret;
32  }
```

Listing 5.3: Sample sandboxed PTX CUDA kernel. Guardian address fencing (bitwise operations) implementation is explained with comments.

function definition that includes a list of parameters –lines 2 and 3. These parameters are addressable, read-only variables declared in the `.param` state space. Parameters are loaded to registers using `ld.param` instructions – lines 11 and 12. Each kernel allocates the minimum number of registers used throughout the execution –lines 9 and 10. Then, the kernel uses these registers to load and store the values generated in each execution step –lines 20-23 and 30-31.

Our patcher (1) adds two extra parameters in each kernel –lines 5 and 7, (2) defines two extra registers to load the mask and the base partition address parameter –line 15, (3) loads the extra parameters in the registers –lines 17-18, and (4) appends two `bitwise` instructions –lines 26 and 28– before every load/store. The `bitwise AND` operation is performed between the load/store address and the mask. The mask for each partition is

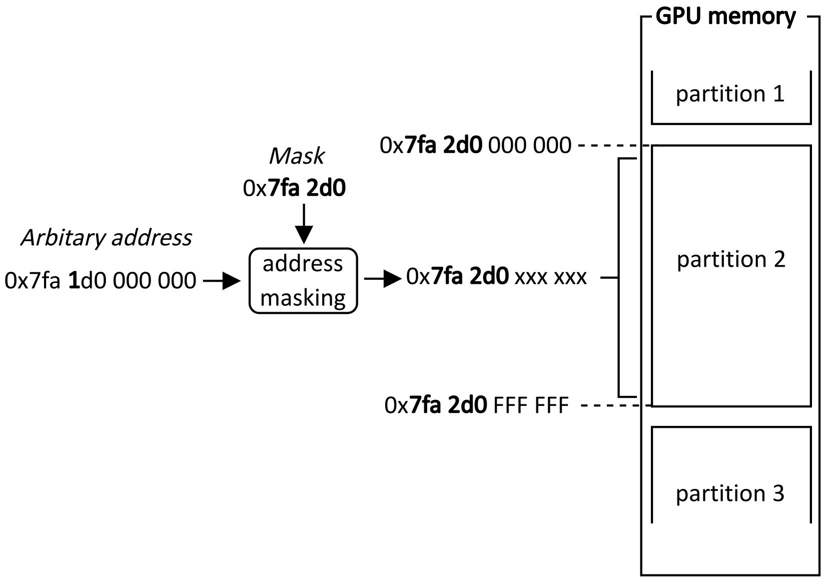


Figure 5.6: Bitwise instructions mask addresses that fall outside a partition.

calculated using the highest address and the partition size. For instance, if the partition starting address is `0x7fa2d0000000` and the partition size is 16 MB: the ending address is `0x7fa2d0FFFFFF` and the mask is `0x000000FFFFFF` (partition 2 in Figure 5.6). In any case, the number of zeros in the mask depends on the partition size. Then we use a `bitwise OR` between the address and the base address of a partition. The `bitwise AND` with the mask and the `bitwise OR` with the partition base address make an address outside the partition to start from the begging of the partition, i.e., wrap around, as shown in Figure 5.6. The illegal address that points to partition 1 (assigned to another application) due to the bitwise operations with the masking address, will finally point to partition 2. With this approach, only invalid or malicious kernels will wrap around and potentially corrupt their data. If memory corruption results to other execution issues (e.g., no convergence in ML applications) for invalid or malicious applications, the gManager can utilize existing techniques [80] to detect and terminate the endless kernel. Alternatively, Guardian can use address checking (§5.4.4) to detect invalid accesses and return from the kernel, but at a higher cost (§5.6.2).

Intel, AMD, and NVIDIA GPUs have two addressing modes for loading or storing data to memory [71, 51]. In the first case, the base address is loaded into the destination register (line 30 in Listing 5.3), while in the second, an offset is first added to the base address, and the result is loaded into the destination register (i.e., $ld.global\ \%val, [\%base\_addr + offset]$). The same modes apply to stores and all off-chip memories. The PTX-patcher applies the bit-masking instructions directly to the base address for the first mode. For the second mode, the patcher calculates the new address by adding the offset to the base address and stores this in a new temporary register. Then, it applies masking instructions to this new address. Our patcher instruments `.func` in the same way as kernels (`.entry`). The `.func` directive denotes a function callable from both host and kernel code.

Indirect branch instructions are unsafe, but we find that, they do not exist in PyTorch kernels. However, Guardian can protect these as well by applying a mask to the index relative to the array size, causing the index to wrap around.

5.4.4 Bounds Checking Tradeoffs

Guardian currently supports three bounds-checking methods: One address checking and two address fencing approaches. Each approach has different requirements and can be dynamically utilized at runtime by Guardian to serve different purposes. First, address checking uses *conditional checks* to verify that the addresses used in load and store operations are in the correct partition. This approach detects out-of-bounds accesses and is more suitable for debugging purposes. Unlike address fencing, address checking can be used for partitions of arbitrary size but at a higher cost (80 cycles) because the Address Divergence Unit executes conditional checks.

Address fencing does not provide out-of-bounds detection, but it is more efficient and practical, making it sufficient for isolating concurrent applications. Address fencing with *modulo* applies the following instructions before every load and store: $fenced\_addr = partition\_base + ((arbitary\_addr - partition\_base)\%partition\_size)$. CUDA ISA implements the 64-bit modulo operation via a function call that requires 2× more cycles than the 32-bit modulo implemented inline by NVIDIA. We implement the 64-bit modulo inline with three instructions and an extra parameter holding the $\frac{1}{parttition\_size}$. The extra parameter avoids the division's high overhead since it is also implemented via a function call. This approach

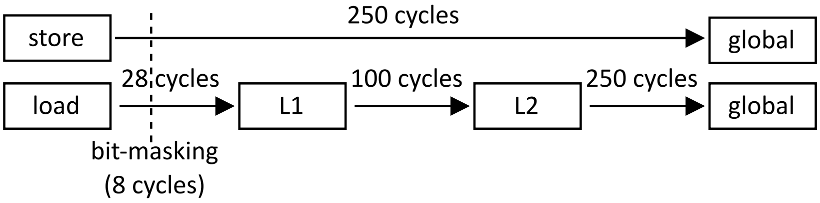


Figure 5.7: Bit-masking latency (8-cycles) compared to latency of different memories.

implies less overhead (28 cycles) than address checking and still does not require partition alignment.

Finally, address fencing with *bitwise* operations is the most lightweight compared to previous approaches because it requires almost 8 cycles –4 cycles per bitwise operation [2]. As shown in Figure 5.7, a load/store instruction requires 28 cycles if the data reside in L1-cache, whereas if the data is in global memory, it requires 220-350 cycles [10, 42]. In the rare case that all the data are in L1-cache (100% cache hit ratio), our approach implies 30% overhead, whereas in the typical case (data in global memory), we add on average 3.5% (§5.6.4). Our approach requires the memory size of the partitions to be in the power of two to cut down the extra instructions required to check a partition's upper and lower bounds. The power-of-two block size allocators restrict the number of concurrent applications, however PyTorch and TensorFlow use this type of allocator as default. Consequently, we choose to optimize the common case and leave the allocation issue as future work.

Guardian passes the mask and the base partition address to every kernel using two extra parameters. Using these parameters inside the kernel requires two extra registers. This does not lead to register spilling, as we show in §5.6.3, because GPU kernels use the minimum number [30, 81] of registers, and the nvcc compiler optimizes further register usage [55]. We have also examined two other possible solutions: The first is a global map stored in GPU memory, but updating the map is prohibitively expensive. The second is to generate a different kernel binary for every partition with the mask hard-coded. This approach does not scale when multiple applications use thousands of kernels. Using JIT to avoid pre-compiling kernels induces considerable overhead. Therefore, the gManager compiles at its initialization the sandboxed PTX with the extra parameters avoiding JIT.

Specifications	RTX A4000	RTX 3080 Ti
Compute Capability	8.6	8.6
#SMs	48	80
#CUDA cores	6144	10240
L1 (KB)	128	128
L2 (KB)	4096	6144
Global memory (GB)	16	12
#Registers / Thread	255	255
PCIe	v4 x16	v4 x16
L1 hit latency (cycles)	28 [10, 42]	28 [10, 42]
L2 hit latency (cycles)	193 [10, 42]	193 [10, 42]
Global memory BW (GB/s)	448	912
Error Correction Code	Yes	No

Table 5.2: GPU specifications we use for the evaluation.

5.5 Experimental Methodology

Our evaluation tries to answer the following questions:

1. What is the impact of Guardian on GPU spatial sharing compared to unprotected NVIDIA MPS (§5.6.1)?
2. What is the overhead of Guardian on real-life applications –running standalone– compared to native execution and other protection approaches (§5.6.2)?
3. What is the impact of address fencing (bitwise operation) on GPU register usage (§5.6.3)?
4. What is the performance of address fencing (bitwise operation) at high cache hit ratios (§5.6.4) using different GPUs and access patterns (§5.6.5)?
5. What is the cost of CUDA runtime and driver API interception (§5.6.6)?

Server platforms: To evaluate Guardian we use two GPU models (Table 5.2), that are installed on two different servers. The first server is equipped with a Quadro RTX A4000 GPU, four AMD EPYC 7551P NUMA CPUs with 8 physical cores each (running at 3.0 GHz, hyper-threaded), and 128 GB of DRAM. To avoid passes over QPI/UPI, we pin applications

Libraries/ Frameworks	#kernels	#functions	#total loads	#total stores
cuBlas (v11)	4115	0	341249	106399
cuFFT (v10)	5173	4	175256	371932
cuRAND (v10)	204	0	4949	3610
cuSPARSE (v11)	4335	0	334694	101792
cuDNN (v7)	11713	5	1032688	551610
Rodinia	23	7	544	285
Caffe	1294	4	87267	32946
PyTorch	27987	319	2083978	857987

Table 5.3: Load and store instructions in CUDA-accelerated libraries and frameworks we use.

to the cores closer to the GPU. The second server contains a GeForce RTX 3080 Ti, an Intel(R) Core i7-8700K CPU with 6 cores running at 3.70 GHz, and 32 GB of DRAM. Both servers have NVIDIA CUDA v11.7 with NVIDIA driver v.515 installed. All the experiments, except §5.6.5, are performed in the Quadro RTX A4000. Regarding GPU kernel scheduling, we use the default NVIDIA policy, namely leftover [32, 74]).

Applications and datasets: To evaluate the overheads of Guardian under *real-world* scenarios, we use multiple neural networks from Caffe [41] and PyTorch [78] frameworks with large data sets that invoke billions of kernels and execute for hours and applications from the Rodinia benchmark suite [17]. Regarding ML applications, we run lenet, siamese, computer vision, and rnn neural networks with the mnist dataset [50], while for cifar10, the cifar dataset [46]. Both mnist and cifar datasets contain hundreds MBs of images. All the above neural networks are executed with 100 up to 500 epochs and invoke up to 142 million CUDA kernels. We also run experiments with imagenet dataset [87], which consists of 256 GB of images, using googlenet, alexnet, caffenet, vgg11, mobilenetv2, and resnet50 as neural networks. These networks invoke billions of kernels, and we run them for ten epochs leading to 99% accuracy. Table 5.3 shows the total number of kernels, functions, and the load/store (`ld/st`) instructions contained in the libraries and frameworks that we use in our evaluation. Regarding Rodinia, we increase the default dataset size by 10× and kernel execution time by 8×, compared to previous work, because the default values are small for executing on real systems.

Workloads with **same** apps			Workloads with **different** apps		
ID	**Name**	**Epochs per app**	**ID**	**Name**	**Epochs per app**
A	2xlenet	500	I	lenet-siamese	500-50
B	4xlenet	500	J	siamese-cifar10	30-100
C	2xcifar10	100	K	2xlenet-siamese-2xcifar10	500-30-100
D	4xcifar10	100	L	3xlenet-siamese-2xcifar10	500-30-100
E	2xgaussian	-	M	hotspot-guassian	-
F	4xgaussian	-	N	gaussian-lavamd	-
G	2xlavamd	-	O	particle-hotspot	-
H	4xlavamd	-	P	gaussian-hotspot-lavamd-particle	-

Table 5.4: Mixes of workloads used for assessing the performance of Guardian under GPU sharing.

Workloads: From Caffe, PyTorch, and Rodinia, we create a set of workloads shown in Table 5.4, to evaluate Guardian under concurrently running applications. Each workload is a mix of compute- and data-intensive applications and covers scenarios in which applications compete and stress the GPU resources. As in previous works [20, 54, 70, 94], we create workloads with 2-6 concurrent clients. The workloads A-H use multiple instances of the same application, while I-P includes different applications. To ensure that application executions overlap, we appropriately modify the number of epochs of each application, affecting the total execution time. We also vary the batch size to increase memory usage in each application from 500 MB to 2 GBs. To assess the applicability and coverage of Guardian, we use CUDA library samples that contain applications that use `cuBLAS`, `cuFFT`, and `cuSPARSE` libraries. These examples include more than 30 library calls that are not in the real-world frameworks we use.

Performance measurements: We use Nsight to profile GPU kernel execution and collect metrics, such as cache hits, GPU calls latencies, and kernel invocations. We use the `Xptxas=-v` compiler flag to measure register and constant memory use by the sandboxed kernels in Guardian. For measuring the duration of host calls, we use the `rdtsc` instruction and we set the CPU frequency to its maximum value.

Baseline and Guardian Deployments: Regarding ***GPU sharing*** we use four deployments. *Native* uses the default CUDA runtime environment, which offers time sharing with protection and represents the baseline performance. The other three setups provide GPU spatial sharing: NVIDIA Multi-Process Service (MPS) [70] allows concurrent execution of multiple kernels but without strong protection guarantees. Guardian without protection the spatial sharing of Arax, which is analogous to MPS. Guardian with address fencing (bitwise operation) is our main approach for protection using bit-masking. Regarding G-NET [115] that uses network functions for its evaluation, we extrapolate its protection mechanism for ML applications using address checking. Mask [8] uses the Mosaic simulator [7] for its evaluation, and we omit to compare directly with Guardian. Finally, MIG [69] statically partitions high-end NVIDIA GPUs, leaving GPU resources underutilized, making a comparison less relevant.

We run ***standalone neural networks*** to isolate Guardian protection overheads. This is essential for two reasons, leading to Guardian overheads amortization. First, the gManager and applications operate in separate address spaces, necessitating IPC mechanisms to exchange data and tasks. This, in turn, increases the execution time of GPU calls. Second, spatial sharing intensifies resource contention, which may increase the latency of GPU loads and stores. Regarding this scenario, we use: (a) *Native* CUDA as a baseline. (b) Guardian *without protection* that just intercepts GPU calls but does not perform any checks nor instrumentation. This setup models the overhead of intercepting and forwarding CUDA calls to the gManager. (c) Guardian with *address checking*, to evaluate control flow instructions. (d) Guardian with *address fencing modulo operation* to measure the overhead of our inline modulo instruction. (f) Guardian with *address fencing bitwise operation* to appraise bitwise instructions.

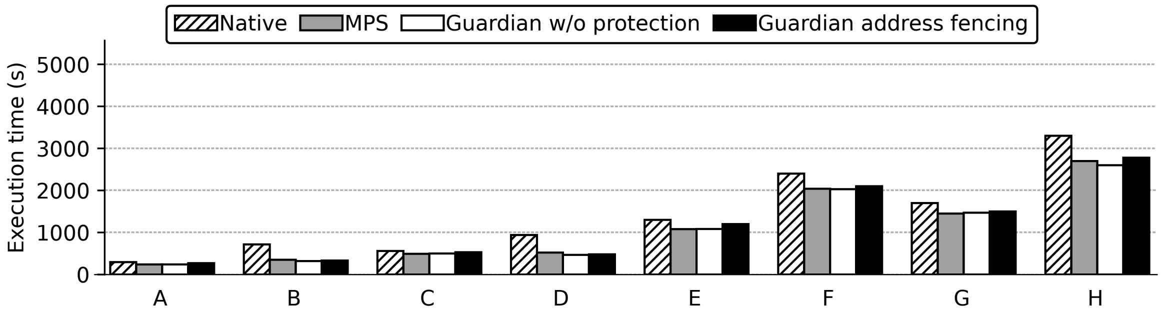


Figure 5.8: GPU sharing using native CUDA time sharing (protected), MPS spatial sharing (unprotected), Guardian spatial sharing without protection, and Guardian spatial sharing with address fencing under workloads with the **same** applications.

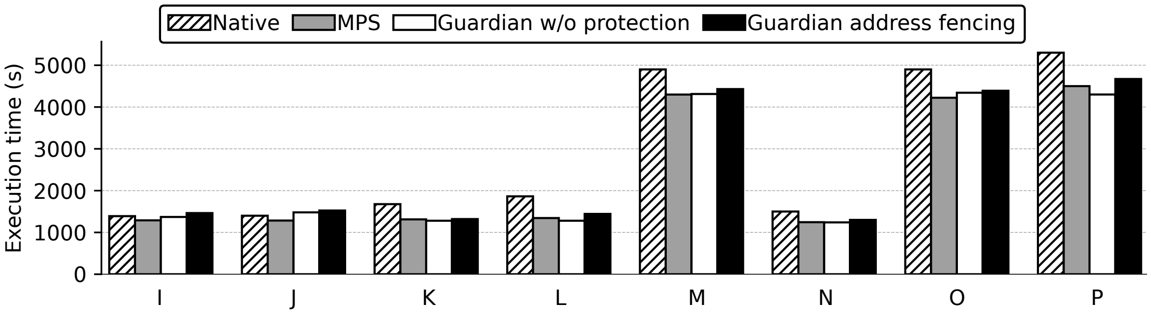
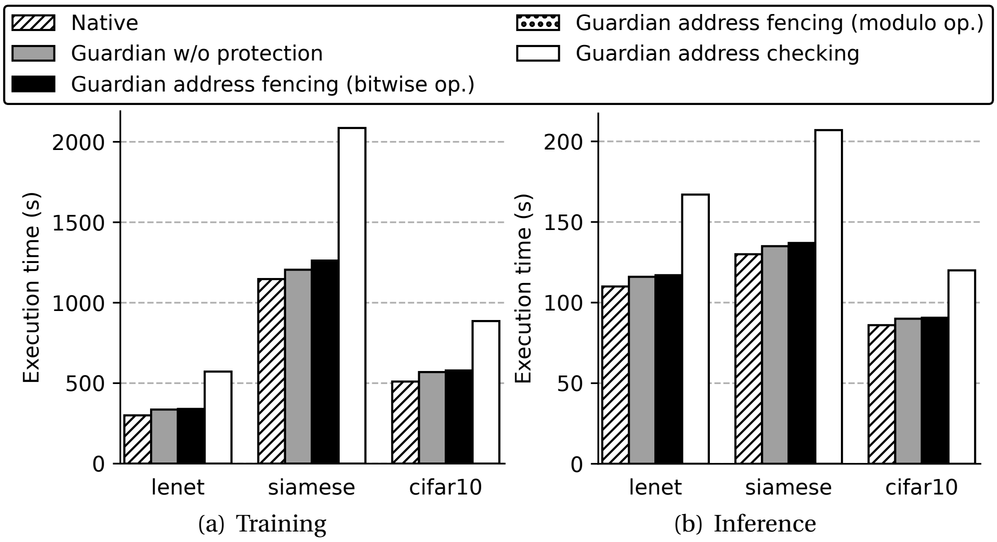


Figure 5.9: GPU sharing using native CUDA time sharing (protected), MPS spatial sharing (unprotected), Guardian spatial sharing without protection, and Guardian spatial sharing with address fencing under workloads with the **different** applications.

5.6 Experimental Evaluation

5.6.1 Impact of Guardian at GPU Sharing

In this section, we compare Guardian address fencing with MPS and Guardian without protection when multiple applications share the same GPU spatially. Figures 5.8, 5.9 shows the execution times of *Native*, *MPS*, *Guardian without protection* and *Guardian address fencing* for the workloads of Table 5.4. Comparing Guardian address fencing to MPS, our approach is, on average, 4.84% slower due to the extra checks enforced to prevent out-of-bound accesses. When we turn off these checks in Guardian (no protection), the execution times achieved are 0.05% worst than MPS. In high resource contention, as in workloads I-P, the overheads of Guardian address fencing are lower on average 3.2% since our overheads



(a) Training (b) Inference

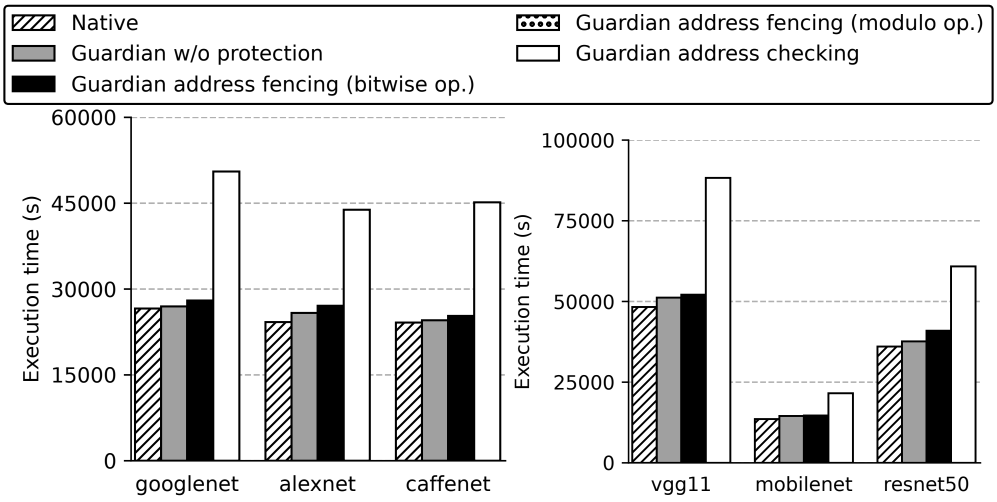
Figure 5.10: Comparison of address fencing (bitwise) with other approaches, using Caffe with *mnist and cifar* dataset.

are amortized. It is worth noticing that Guardian without protection performs better than MPS in workload with thousands of pending kernels (D, P, K, H). This is because the MPS server becomes a bottleneck according to previous works [79].

Finally, we compare spatial and temporal sharing, which is the default mechanism used from many previous works [113, 108, 16, 48] because it ensures protection. Guardian address fencing is, on average 23% faster than native, while in some cases, it is up to 2× faster due to parallel kernel execution. We note that the performance improvements of spatial sharing are primarily affected by the resources required by the concurrently executing workloads. In cases where the resources needed are low, as in workloads B and D, the benefits are more prominent, i.e., 2×, while the performance gap is reduced for more resource-intensive workloads.

5.6.2 Guardian Overheads Compared to Other Approaches Without Sharing

Figures 5.10 and 5.11 plot the times of the individual execution for several ML frameworks and CUDA-accelerated libraries (Table 5.3), using *Native, Guardian without protection, Guardian address fencing (bitwise and modulo),* and *Guardian address checking.* We note



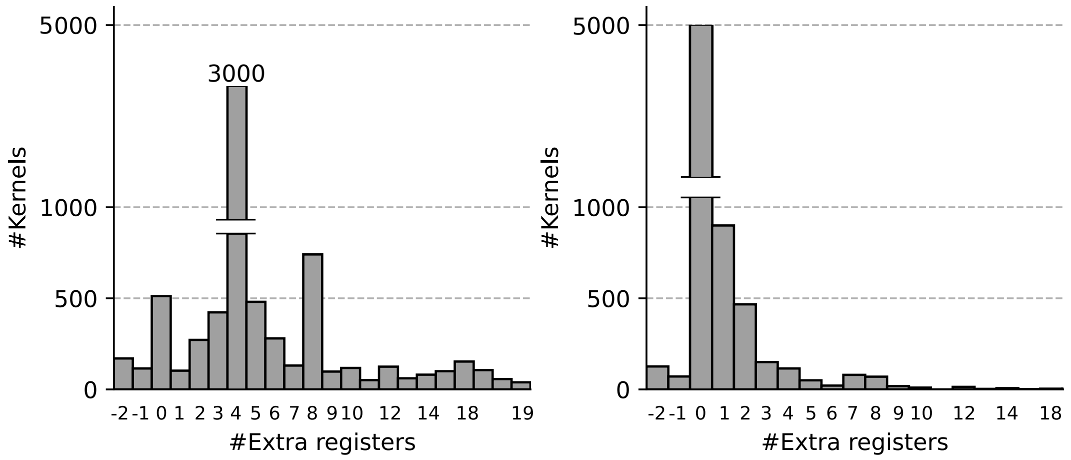
(a) Caffe Training (b) PyTorch Training and Inference

Figure 5.11: Comparison of address fencing (bitwise) with other approaches, using Caffe and PyTorch with the *imagenet* dataset.

that the training phase for lenet, siamese, cifar10 issues up to 142 million kernels, whereas googlenet, alexenet, caffenet, vgg11, mobilenetv2, and resnet50 issue billions of kernels. The inference phase issues up to 8 million kernels.

Figure 5.10(a) shows lenet, siamese, and cifar10 training, while Figure 5.10(b) shows the inference phase of the same neural networks. Guardian has between 5.9% up to 12% overhead compared to the unprotected native CUDA. The Guardian without protection approach includes the interception of CUDA calls and the search in the *pointerToSymbol* table to find the appropriate sandboxed kernel. The kernel issued in the GPU does not contain the bit-masking instructions, while transfer instructions do not contain the out-of-bounds checks added from Guardian address fencing. The Guardian without protection approach has an overhead from 3.7% to 10% compared to native. By comparing Guardian address fencing (bitwise operation) and Guardian without protection, the overhead of Guardian address fencing is between 1.05% up to 4.3%. As a result, the overhead added by bounds checking (in transfers and PTX kernels) is 2.9% on average.

Figure 5.11(a) shows googlenet, alexenet, and caffenet training. Guardian address fencing (bitwise operation) has between 4.5% up to 10% overhead compared to the unprotected native CUDA. The Guardian without protection approach has an overhead from



(a) No Optimizations (-G)

(b) Optimization level 3 (-O3)

Figure 5.12: Guardian's per thread register usage vs native.

1.36% to 6% compared to native. By comparing Guardian address fencing (bitwise operation) and Guardian without protection, the overhead of Guardian is between 2.9% up to 4.3%. Figure 5.11(b) shows vgg11, mobilenet, and resnet50 training and inference using PyTorch. The overhead of Guardian for call interception is, on average 5.5% (native vs. Guardian without protection). The overhead of Guardian address fencing compared to Guardian without protection is on average 7.6%.

Our optimized modulo –without function call– approach, namely address fencing modulo operation, increases the execution time by 29% on average compared to native, due to the addition of seven extra instructions. Conditional checks increase execution time by 1.7× on average compared to native. This is because the branch instructions are more expensive compared to bitwise operations. In the addressing mode that uses `address+offset` we add up to eight instructions (32cycles) to check the bounds of each memory partition.

5.6.3 Impact of Address Fencing on Register Usage

Figure 5.12 shows the number of registers that are eventually used for storing the address mask and the base address in the address fencing bitwise approach. Figure 5.12(a) shows the additional registers used from our approach when compiling the PTX without any optimization flag, whereas Figure 5.12(b) shows full optimizations. The lack of optimization

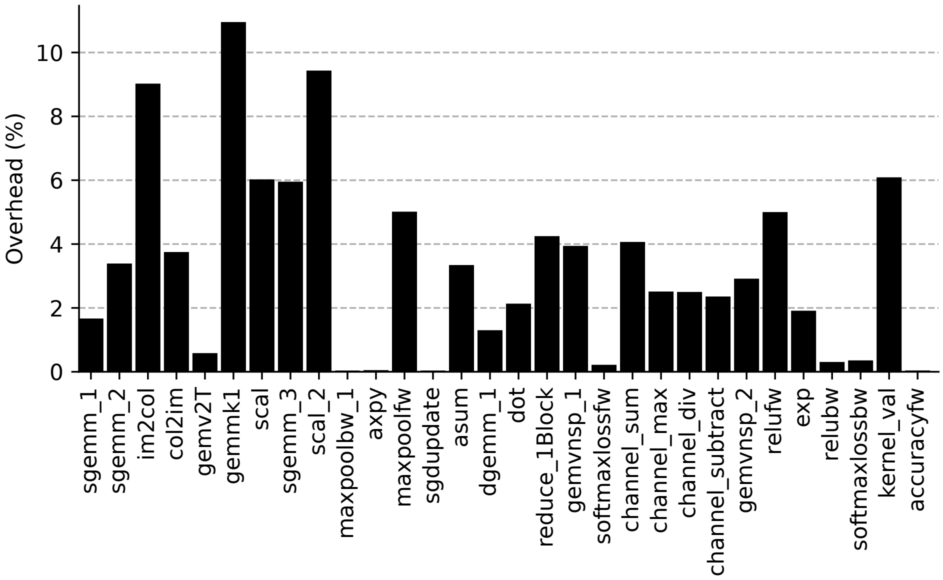


Figure 5.13: Performance overhead of sandboxed kernels against native execution.

flag results in kernels (from cuBINs) using up to 4 additional registers in 62% of the total kernels. However, when we use full optimizations in the compilation (O3), 71% of kernels use no extra registers, 13% use up to one extra register, and 7% use up to two extra registers. In some rare cases, the number of registers is smaller than the default because the compiler spills some registers in the global memory. Regarding the constant memory affected by the extra parameters Guardian adds in 99% of kernels 16 bytes.

5.6.4 Performance of Address Fencing at High Cache Hit Ratio

Figure 5.13 shows the overhead of Guardian address fencing (bitwise operation) normalized to native for 890000 kernels used in lenet. The overhead of Guardian is, on average 3.2%. We have performed the same breakdown for computer vision and observed similar results. The overheads of Guardian bit-masking instructions depend on the latency of the load and store instructions. A load instruction that retrieves data from global memory is 220-350 cycles [10, 42], while if data are in L1-cache is 28 cycles. Our approach adds two (bitwise AND, OR) up to four instructions (for cases that include address+offset) per load and store instruction. Each of these instructions is executed in almost 4 cycles. As a result,

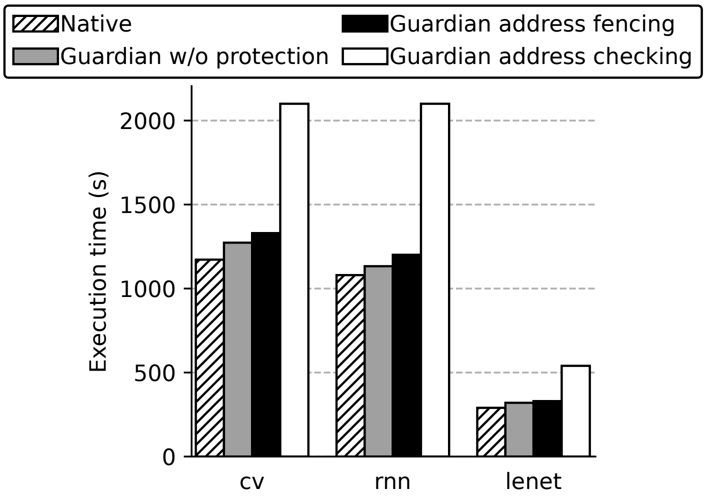


Figure 5.14: Guardian overhead with PyTorch and Caffe on GeForce GPU, compared to native execution.

if all data are in the L1 cache –not common–, our overhead is from 28% up to 57%. If all data are only in global memory, our overhead (with global memory latency 285 cycles) is from 2% up to 5%. We have profiled all kernels of lenet and observed that the average L1 cache hit rate is 37%, while for L2 is 72%. L2 latency is 180 cycles, only 1.4× better than global's. Overall, address fencing (bitwise operation) in Guardian incurs small additional overhead due to two main reasons: (1) As we show, ML kernels exhibit a low cache hit ratio. (2) As shown from previous works [6], cache hits result in a lower load/store instruction latency in the rare case that every thread in the warp hits in the cache.

5.6.5 Performance of Guardian on Different GPUs and Access Patterns

Figure 5.14 shows the results with three neural networks from PyTorch and Caffe executed in the GeForce GPU. In computer vision (cv) and rnn Guardian address fencing (bitwise operation) incurs 12% and 10% overhead compared to native, respectively. Lenet with Guardian incurs 13% overhead compared to native. Conditional checks exhibit on average 1.8× worst execution time compared to native. Overall, we note that Guardian has similar overhead across different GPU types.

Figure 5.15 shows the performance of Guardian over CUDA-accelerated library calls

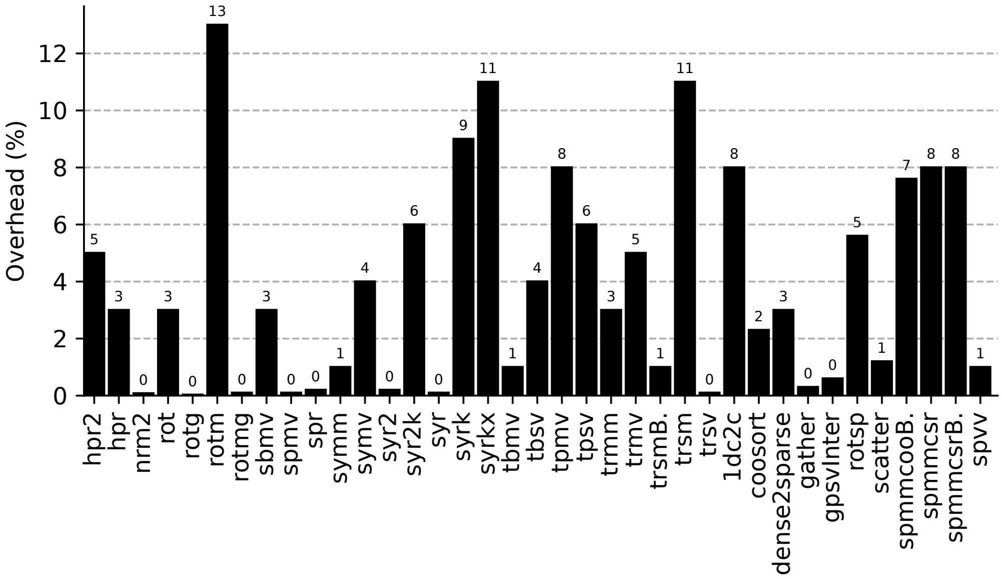


Figure 5.15: Guardian overhead (%) for 37 kernels from CUDA-accelerated libraries compared to native execution of each kernel on the GeForce GPU.

that are not contained in the ML frameworks used previously. Guardian successfully intercepts these calls and adds 4% overhead, on average, which is similar to the results observed with the Quadro GPU.

5.6.6 Cost of CUDA calls Interception

The interception of kernel invocations in Guardian requires between 214 and 900 CPU cycles ("Lookup GPU kernel" in Table 5.5) for the lookup operation to locate the sandboxed kernel (stored in a c++ `unordered map`). Regarding the extra arguments passed in the kernel, we require between 300 and 600 CPU cycles to allocate a new parameter array and copy the new and old parameters in this array ("Augment kernel params" in Table 5.5). Guardian adds on average 957 CPU cycles per `cudaLaunchKernel`. We perform each experiment ten times, excluding the minimum and maximum values.

The `cudaLaunchKernel` NVIDIA system call is measured using the Nsight profiler. The average execution time (for more than one thousand kernels) in CPU cycles is approximately 9000 CPU cycles ("Launch kernel to GPU" in Table 5.5). So our overhead *without* the kernel execution is 10% on average. We have profiled lenet and cv applications (exe-

	Lookup GPU kernel	**Augment kernel params**	**Launch kernel to GPU**
Native	0	0	~9000
Guardian	557	400	~9000

Table 5.5: Guardian average cost in CPU cycles for the main operations performed when a kernel launch is intercepted and replaced with a sandboxed kernel.

cuting millions of kernels) and found out that the kernel execution time *without the cudaLaunchKernel* is, on average 18000 CPU cycles. Consequently, the overhead of Guardian, including the kernel execution time, is 3% per kernel, on average.

Memory allocation and data transfer We use a micro-benchmark that uses memory allocations and data transfers of different sizes to evaluate Guardian's memory management operations. The results suggest that (a) our allocator does not imply overhead compared to native CUDA, and (b) the protection checks used on every data transfer over the PCIe bus imply negligible overhead.

5.7 Summary

This chapter presents Guardian, a GPU memory sandboxing approach that allows real-life applications from different users to share a GPU safely. Guardian makes it practical to share GPUs spatially in multi-tenant environments. The benefits of Guardian are three-fold: (1) It is transparent to applications, even when applications use closed-source GPU-accelerated libraries that include host-level and GPU kernel code. (2) It fences all memory accesses of GPU kernels, including closed-source kernels, by instrumenting kernels at the PTX level. (3) It incurs low overhead using bit-masking to fence addresses without performing address range checks. Our evaluation using real-world ML frameworks shows that Guardian can support all required GPU-accelerated libraries transparently and introduces on average 9% overhead compared to native unprotected execution. Additionally, Guardian incurs up to 2.4× less overhead compared to other software-based approaches, while it is comparable to hardware-based approaches.

Chapter 6

Related Work

The related work of this dissertation falls into four different research areas: (i) Decoupling application from accelerators, (ii) Simplifying FPGA access and providing sharing, (iii) GPU kernel revocation with SLA guarantees, and (iv) GPU memory protection under spatial sharing.

6.1 Decoupling applications from accelerators

We categorize related work in four areas: (a) static accelerator assignment, (b) dynamic accelerator assignment, (c) accelerator virtualization, and (d) accelerator spatial sharing. Table 6.1 compares Arax with previous state-of-the-art approaches.

Existing programming models, such as CUDA [40], SYCL [84], and oneAPI [3], enforce applications to select the desired accelerator types either at compile time or at the beginning of application execution, resulting in static binding of applications to accelerators. StarPU [5] performs finer-grain assignment of a graph of tasks to multiple and heterogeneous processing units; however, still in a static manner. Arax assigns tasks dynamically to the available accelerators. It also provides spatial sharing across heterogeneous accelerators and a stub generator to reduce application porting effort. We note that Arax and StarPU offer a similar approach for defining independent sets of work. StarPU indicates a set of dependent tasks with labels, whereas Arax uses task queues.

Arax shares similar goals with recent work in dynamically assigning GPUs to appli-

Approach	Heterogeneity	Spatial Sharing	Dynamic resource assignment	Reducing effort
MPS [20]	-	✓	-	-
StarPU [5]	✓	-	-	-
Gandiva [108]	-	-	✓	-
DCUDA [35]	-	-	✓	-
AvA [113]	✓	-	-	✓
Arax [79]	✓	✓	✓	✓

Table 6.1: Capabilities of Arax vs. state-of-the-art approaches.

cations. Gandiva [108] is a cluster-level scheduler for ML training applications that dynamically assigns GPUs to applications. DCUDA [35] is a runtime system that provides dynamic assignment of applications to GPUs. The main limitation of these works is that they are either based on domain-specific application features or vendor-specific accelerator mechanisms. Gandiva migration uses TensorFlow checkpoints, which however, are not provided by all applications and frameworks [16]. DCUDA provides support only for NVIDIA GPUs. In contrast, Arax is accelerator-agnostic and does rely on application- or accelerator- specific mechanisms.

Previous work has also explored the concept of accelerator virtualization [113, 93, 23]. API remoting [93, 23] is an I/O virtualization technique in which API calls are forwarded to a user-level computing framework [93] or to a remote server [23]. The main disadvantage of API remoting is the inability to support multiple APIs, which is not the case for Arax. AvA [113] is a framework that virtualizes heterogeneous accelerators. However, with AvA, all accelerator calls, including kernels with microsecond execution time, go through the hypervisor, increasing response time. Additionally, AvA requires applications to select the accelerators in advance, leading to static application to accelerator assignment. AvA creates a server for each application to execute tasks to accelerators. This design decision does not allow GPU spatial sharing due to the lack of a single context. Arax is a user-space approach resulting in less overhead, as our evaluation shows. Arax frees applications from accelerator selection, allowing dynamic task assignment. By creating a single GPU context, our server enables spatial sharing.

NVIDIA GPUs support by default time sharing, according to which only one application uses the GPU at any time. The CUDA runtime executes the kernels from different applications back-to-back [8], which, however, does not improve the GPU utilization for applications that do not have enough parallelism to utilize all the resources of beefy GPUs. Kernel preemption introduced in Pascal architecture allows the time-sharing scheduler to share the GPU across applications fairly. However, due to the fine-grain context switching, the execution time of applications increases.

Finally, GPUs support spatial sharing through NVIDIA MPS [20], while AMD GPUs support it by default. On the other hand, FPGAs require partial reconfiguration that divides the FPGA into fixed areas; these areas can then accommodate different compute kernels. Even though each of these mechanisms provides spatial sharing primitives for each accelerator type, they still require low-level knowledge of each accelerator API and its runtime to implement task assignment policies. Moreover, it may require coordination across different applications, e.g., FPGAs, which is not always possible in modern servers. Finally, existing sharing mechanisms rely on applications to select the accelerator they will use, leading to inefficiencies. Arax's advantage is that it can handle sharing heterogeneous accelerators while abstracting the related complexity away from applications. For instance, with FPGAs, the Arax server performs any required partial reconfiguration, loading the appropriate bitstream to serve a task. Finally, Arax makes it easy to apply new task assignment policies transparently to all applications facilitating further research in the area.

6.2 FPGA Software Access and Sharing

The use of heterogeneous systems comes at a significant cost: the increase in programming complexity at different levels. To overcome issues related to programming the FPGA itself, developers can employ *high-level Languages,* such as OpenCL or System C [105, 9, 90, 89]. However, little has been done to reduce the effort in incorporating FPGAs in applications and services. Dynamic reconfiguration [12] of FPGAs has traditionally been used to improve flexibility and usability. More recently, research has examined various techniques to partition FPGAs so multiple applications can use them. Our work is orthogonal to these efforts since we aim to allow applications to share each partition simultaneously.

Next, we discuss recent progress in incorporating FPGAs in the data and cloud applications.

Fahmy et al. [27] presented a framework that integrates reconfigurable accelerators in a standard server with virtualized resource management and communication. The proposed framework integrates a PCIe-based FPGA board into a standard datacenter server. The FPGA is partitioned into separate accelerator slots. Accelerator functions are either stored in a library on the host machine as partial bitstreams or can be uploaded by the user. In this framework, a hypervisor is implemented to configure and schedule the user logic in the FPGA resources. Unlike this approach, VineTalk places the accelerator controller in the host, avoiding dependencies with a hypervisor.

In [4], a runtime system has been proposed to simplify the FPGA application development process by providing a high-level API and a simple execution model that supports software and hardware execution. The proposed framework allows the design and dynamic mapping of accelerators onto FPGAs for cloud applications. The runtime system utilizes a MicroBlaze soft core running FreeRTOS tasks responsible for low-level management of hardware resources (memory, PCIe, partial reconfiguration, and accelerators). Host applications interact with the accelerator using a DyRact API. In contrast, VineTalk uses a host core to schedule tasks/accelerators, leaving all FPGA/accelerator resources available to applications. Additionally, VineTalk uses a higher-level, hardware-agnostic API that decouples application developers from accelerator-specific knowledge, such as a number of DMA channels.

6.3 GPU kernel revocation and scheduling

Sharing a GPU across applications introduces difficulties in providing latency guarantees to user-facing tasks. We categorize previous approaches in scheduling and preemption.

6.3.1 SLA-based scheduling

Timegraph [44], Baymax [18], gVirt [97], and VGRIS [114] implement sophisticated SLA-aware task scheduling policies. These approaches consider only batch tasks with execution time comparable to the SLA, in the order of tens or a few hundred milliseconds. Thus, they alleviate the *priority inversion problem* when a critical task waits for a batch task to finish, only with short batch tasks.

With modern workloads, long-running batch tasks are becoming more common as the complexity of the algorithms and the amount of data they use increases. Long-running tasks can monopolize the GPU [68], requiring a GPU preemption or revocation mechanism. TReM is a task revocation mechanism that can be coupled with suitable scheduling policies. A scheduling policy will determine when a task has to be killed and instruct TReM to kill it.

6.3.2 State-saving preemption mechanisms

Operating systems can preempt a running process and give the CPU to another process within a few microseconds. Such low overhead preemption mechanisms are not available in modern GPUs.

Chimera [75] is an effective preemption approach that provides block context switching, draining, and flushing. However, Chimera is only implemented in a simulation environment and is not supported by existing GPUs. PEP [56] and GPU snapshot [52] add incremental checkpoints to decrease the overhead of saving the full context of a revoked or failed kernel. These approaches are orthogonal to ours. TReM can be integrated with Kyushick's [52] approach to bind the wasted work to a single checkpoint interval at the cost of additional memory usage at the GPU.

Sajjapongse et al. [88] rely on existing synchronization points to preempt kernels. Their approach cannot provide low response time to user-facing tasks when synchronization points are rare. Moreover, they transfer the state and data of the preempted kernel to host memory, resulting in long preemption latency for tasks with a large memory footprint. GPES [116] splits the kernel statically into multiple sub-kernels. Consequently, it avoids the problems implied by synchronization points. However, GPES has to determine the granularity of a slice. If the slices are too small, they introduce runtime overhead; if they are too large, they can not provide low preemption latency. In contrast, TReM provides low revocation latency because it can revoke a kernel at any arbitrary point of its execution without saving any task state.

FLEP [106] uses `asm(exit)` to preempt kernels, as discussed in detail in Section 4.1. To reduce the preemption latency, FLEP splits the initial kernel into multiple smaller kernels using persistent threads [36]. Thus it limits the number of launched thread blocks in each kernel to the maximum number of thread blocks that the GPU can simultaneously execute [36]. However, the preemption latency of FLEP depends on the execution time of thread blocks. Additionally, FLEP does not discuss freeing the GPU DRAM from preempted task data, thus potentially inducing memory monopolization. Finally, FLEP requires kernel source code to make it preemptible. TReM is based on CUDA dynamic parallelism [43], so it does not rely on kernel slicing and does not require kernel source code. Furthermore, it can kill a kernel at any point of its execution, with constant delay. TReM does not save the state of the killed kernel in GPU memory; hence, it does not prevent other high-priority kernels from starting due to memory shortage.

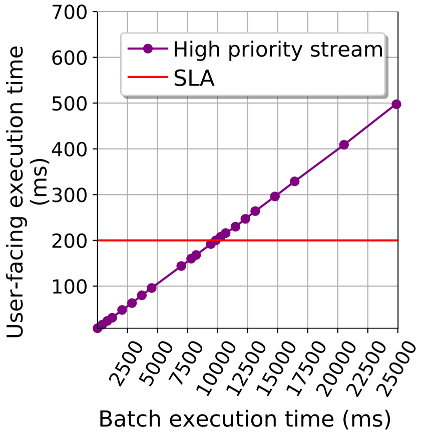


Figure 6.1: Evaluating NVIDIA compute preemption by collocating two tasks in the same GPU using high- and low-priority streams. The user-facing task is assigned to the high-priority stream and a batch to the low-priority one. The high-priority stream preempts the low-priority one. As we increase the batch duration (x-axis), the duration of the user-facing task is affected and increases linearly.

Approach	Preemption/ Revocation	Low & Constant preemption latency	Tasks with large memory footprint	No need for krnl src code	Supports all NVIDIA GPUs (C.C.*>3)	Provides SLA aware policies
FLEP [106]	✓	-	-	-	✓	-
GPES [116]	✓	-	-	-	✓	-
Pascal Preemption [68]	✓	*Not known*	✓	-	-	-
Chimera [75]	✓	✓	✓	-	✓	-
Baymax [18]	-	-	✓	-	✓	✓
TReM + Elastic [80]	✓	✓	✓	✓	✓	✓

Table 6.2: TReM and prior state-of-the-art approaches.
**Compute Capability*

NVIDIA Pascal GPUs, similarly to FLEP, provide a compute preemption mechanism that allows CUDA kernels to be interrupted at block-level granularity. According to CUDA [60] *the higher priority stream will preempt blocks already executing in the low priority stream.* We evaluate the compute preemption of Quadro P1000 (Pascal architecture), using streams with priorities [60]. In our experiment, we use a benchmark with two kernels, one assigned to a low-priority stream (i.e., batch) and the other to a high-priority stream (i.e., user-facing). The two kernels consist of 1024 thread blocks with 512 threads per block to ensure that they will compete for stream multiprocessors (SMs). These two kernels are identical, iteratively copying one array to another. We first start the low-priority kernel, and we measure the latency of the high-priority kernel. Figure 6.1 shows that the execution time of the high-priority kernel *increases linearly* with the execution time of the low-priority kernel, even though the high-priority kernel size stays fixed. Therefore, the preemption latency depends on the low-priority kernel's thread block execution time.

Table 6.2 compares TReM with previous state-of-the-art approaches. To the best of our knowledge, Elastic+TReM is the only SLA-aware scheduling solution that deals with long-running batch tasks, using a revocation mechanism that: (i) provides low and constant latency; (ii) handles tasks with large memory footprint avoiding to store or transfer large amounts of state and merely restarting the task; (iii) does not require GPU kernel source code; and (iv) works with all NVIDIA GPUs (Compute Capability (C.C.) ≥ 3) that exist in today's and future datacenters.

6.4 GPU memory protection

Table 6.3 summarizes the main characteristics of state-of-the-art memory protection mechanisms and Guardian. Next, we discuss the main approaches for protection in shared GPUs.

6.4.1 Protect GPU Memory under GPU sharing

Time-sharing offers memory protection since it allows only one context to be active in the GPU at any time; thus, it is mainly used from previous works [108, 113, 48, 23, 109]. The device driver is responsible for allocating and managing resources belonging to a context. Upon a context switch, its resources are freed, and the translation lookaside buffer (TLB) is invalidated. Consequently, application data are protected at the cost of GPU utilization because context switching is expensive [8, 115, 111]. On the other hand, Guardian eliminates the expensive context switching and improves GPU utilization by offering protected spatial GPU sharing.

Mask [8] is a hardware-based approach that allows applications to share spatially and securely a GPU. Mask extends the GPU TLB to hold information about warps and the memory they can use, and as a result, it supports protected spatial sharing. Mask supports closed-source GPU libraries but with limited applicability due to the special hardware required. Guardian does not need extra or special hardware, protects closed-source libraries, and implies comparable overhead, making it more practical, powerful, and generic.

NVIDIA's Multi-Instance GPU (MIG) [69] partitions statically high-end NVIDIA GPUs (i.e., A100 and H100) in completely isolated parts. Besides the limited applicability (requires special hardware), recent works [54, 8] showed that MIG static partitioning leads to under-utilization and that changing from one partition scheme is not flexible. AMD and Intel GPUs do not offer any protection and, by default, allow applications to share a GPU [79] spatially. Guardian uses a more dynamic GPU sharing scheme, similar to

Approach	No src code mod.	CUDA lib support	No extra /special HW	Spatial sharing
Time-sharing [70]	✓	✓	✓	-
Mask [8]	✓	✓	-	✓
MIG [69]	✓	✓	-	✓
G-NET [115]	-	-	✓	✓
Guardian	✓	✓	✓	✓

Table 6.3: Comparing Guardian with state-of-the-art memory protection approaches for GPU sharing.

MPS [70], with protection guarantees. Regarding compute resource isolation (i.e., CUDA cores), Guardian can use existing approaches [74, 115] or MPS resource provisioning [20].

G-NET [115] is a software-based approach that overcomes the limited applicability of hardware-based ones. G-NET deploys a custom type of pointer [91], namely `isoPointer`, that checks if the accessed memory address belongs to the correct partition. However, leveraging these pointers requires manual effort to port the kernel source code. The source code requirement is a serious limitation leading to weaker protection because most CUDA-accelerated applications rely heavily on closed-source GPU libraries, e.g., cuBLAS and cuDNN. Guardian operates in the kernel code's virtual assembly (PTX) available in closed-source GPU libraries.

6.4.2 Detect Buffer Overflows for a Single Application

clArmor [25] and GMOD [22] protect against overflows by adding canary values around the allocated buffers. However, such approaches have limited security coverage because they cannot capture non-adjacent accesses that jump over canaries. Parravicini et al. [77] add conditional checks inside the kernel LLVM-IR to preserve Java memory safety semantics in NVIDIA GPUs. They use static analysis to minimize the significant overhead implied by conditional checks, which require the application and kernel source code (or LLVM-IR), limiting its applicability. GPUShield [51] overcomes the limitation of source code using an extra hardware unit that performs the address checking. CUDA-MEMCHECK and cuCatch [95] are debugging tools that operate in the PTX [71] level and detect out-of-bounds accesses without requiring extra/specific hardware. All these approaches focus on buffer overflow detection of a single application and are considered orthogonal to Guardian.

6.4.3 Ensure Privacy and Data Confidentiality

Graviton [102] is a trusted execution environment (TEE) providing privacy and data confidentiality guarantees. Graviton requires minimal hardware modifications only in the GPU command processor. Honeycomb [61] relies on address checking to eliminate the necessity for hardware modifications. Furthermore, it depends on source code for static analysis to minimize its overhead. Although, TEEs tackle a significantly different problem [72], Guardian can be combined with Honeycomb to provide a TEE for GPUs with low overhead and support for closed-source GPU libraries.

6.4.4 API Remoting

Cricket [24], DGSF [28], rCUDA [23], Arax [79], and GPUless [98] treat high-level functions to CUDA accelerated libraries as a black box. This is because CUDA libraries use an undocumented function, the `cudaGetExportTable()`, which exports a set of function pointers that implement hidden functionalities. Guardian uses a minimal implementation of these hidden CUDA calls, which is however adequate to run PyTorch and Caffe. We experiment with both interception approaches and determine that the Guardian interception approach is more robust. This is because we only need to intercept 200 relatively straightforward CUDA runtime-driver API calls, as opposed to dealing with 1200 high-level (far more complex) calls to CUDA-accelerated libraries [79].

Chapter 7

Future Work

This dissertation provides solutions to the main issues that exist in the use of multiple and heterogeneous accelerators within a server. Next, we describe some directions for future work.

7.1 Apply zero-copy in shared memory

According to our preliminary evaluation, the main overhead of our runtime depends on the kernel computation- to-communication ratio. For kernels with a high computation-to-communication (C2C) ratio, our overhead is up to 5%. For a low C2C ratio, the overhead is up to 70%. Our overheads are more pronounced for kernels with low C2C ratios because we perform an extra copy in the shared memory. We create a micro-benchmark that transfers variable-size data to measure the overhead implied by a data transfer. On average, the overhead of our runtime is 1.7× slower than native CUDA due to the extra copy performed to the shared memory segment. In particular, to transfer 1 GB of data from an application to the accelerator, our runtime requires 180 ms for the CUDA copy and another 135 ms for the copy from the application to the shared memory. The extra copy in the shared memory achieves 8.2 GB/s throughput (measured by the STREAM [64] benchmark, using a single CPU core). We note that this overhead primarily affects applications with a low computation-to-communication ratio. As part of our future work, one can use zero-copy or double-buffering techniques between the applications and server address spaces to minimize this overhead.

7.2 Use remote heterogeneous accelerators

Access to local accelerators is not always feasible for some reasons: (1) the lack of physical space in the computer to install the accelerators, (2) the lack of available hardware slots, such as PCIe slots, (3) PCIe slots with a version older than the one needed by the accelerators, (4) economic reasons, given that the cost of this kind of devices is not negligible. To address these issues, we can extend Arax to forward calls to remote accelerators in other servers. Existing works [23, 35] support only NVIDIA GPUs, whereas our approach will support heterogeneous accelerators and spatial sharing.

7.3 Batch dependent tasks

The execution time of kernels used from ML frameworks is in the range of microseconds, according to our profiling. As a result, forwarding, either in the same server or, even worst, to a remote server, a single accelerator call, or a kernel implies high overhead. To amortize the overheads of remote calls, we can create a batch of tasks, as we did in our initial version –VineTalk–. The key difference is that the Arax server should not view a task as a black box. Instead, it should access the individual kernels and accelerator-related calls to support mechanisms such as lazy data placement, dynamic task assignment, live migration, sharing, and preemption.

7.4 Compile PTX kernels to other GPUs

Previous works [37, 38, 14, 45] convert CUDA **source code** to other accelerators. However, intercepting high-level calls of domain-specific libraries is not robust, as shown in Guardian. Closed-source CUDA libraries contain only the PTX version of CUDA kernels. Consequently, in order to facilitate heterogeneous accelerator support, two fundamental strategies emerge: the conversion of PTX code to accelerator-specific assembly or the translation of PTX to LLVM Intermediate Representation (LLVM-IR). The first approach, which involves direct PTX-to-assembly conversion, is hindered by performance limitations. Furthermore, for each unique accelerator type, the same optimization and transformation procedures must be repeated, resulting in redundancy and inefficiency. In contrast, the adoption of LLVM-IR presents a more promising solution. The LLVM-IR inherently offers built-in support for code generation across a wide spectrum of accelerator types, making it a compelling choice for achieving greater efficiency and versatility in this context. As a result, Arax will be able to support multiple and heterogeneous accelerators more effectively.

7.5 Extend accelerator memory

Datacenter workloads work with massive data structures up to terabytes. For instance, graphs, data analytics, graph neural networks, and recommender systems access massive datasets organized into array data structures whose sizes range from tens of gigabytes to tens of terabytes and are expected to proliferate in the foreseeable future. Storing these datasets as in-memory objects enables applications to naturally and efficiently process the data. However, even high-end GPUs' memory capacity is insufficient to hold all the datasets (A100 has 80 GB DRAM). The state-of-the-art approaches that extend the GPU memory use: (i) Page-faults (DRAGON [62]), or (ii) direct access to storage devices (BaM [82]). DRAGON extends NVIDIA's unified memory page fault mechanism to serve faults from the storage device. This approach is transparent at the cost of performance due to page faults. BaM requires modifying the CUDA kernels to use *BaM_Arrays*. As a result, it sacrifices transparency for performance. However, kernel modification cannot be done for CUDA closed-source domain-specific libraries (e.g., cuBLAS, cuRAND). Arax server could be extended to use a key-value store. Key-value stores can offer fast random accesses. Our approach can provide the required performance without modifying kernels.

7.6 Integrate Arax to a cluster-level scheduler

In real-world setups, the Arax server is deployed across all nodes within a data center. Effective coordination between resource managers, such as Kubernetes, and the Arax server is of paramount importance to provide Quality of Service (QoS) and prevent resource overcommitment. The resource manager's scheduler plays a crucial role in workload distribution, responsibly assigning tasks across servers while considering the accelerator resource utilization information furnished by the Arax server. This collaborative effort ensures optimal resource utilization while maintaining QoS. Furthermore, within each node, Arax's accelerator selector is responsible for scheduling the tasks of the assigned applications to the underlying accelerators. This multi-tier scheduling approach, involving both server-level and accelerator-level allocation, contributes to the overall efficiency and performance of the system.

Chapter 8

Conclusions

This dissertation proposes a runtime for managing multiple and heterogeneous accelerators within a server. Our approach has four main contributions: (1) Arax removes static application to accelerator assignment using a generic API and a shared runtime process that manages accelerators. In particular, we provide three main abstraction primitives to hide the accelerator type and number from applications. To optimize the accelerator resource management across applications, we design and implement a shared runtime process, the server, for all applications in a node. The server assigns application tasks to accelerators as late as possible to be able to adapt to application load change. Additionally, to allow flexibility in task placement, it moves each task's data just before its execution.

(2) To streamline the process of adapting existing CUDA applications to our API, we analyze the interception of various levels within the CUDA software stack. This analysis encompassed the CUDA driver API, the CUDA runtime API, and high-level function calls to CUDA libraries. Our findings led us to conclude that intercepting the CUDA runtime and driver API is the most robust level of interception. This preference is primarily attributed to the limited number and relative simplicity of calls that need to be managed at this level.

(3) TReM is a revocation mechanism that can be integrated with scheduling policies to ensure the performance of latency critical GPU kernels. Our revocation approach is engineered to halt the execution of a GPU kernel at any given point in its operation, leveraging the capabilities of CUDA dynamic parallelism, CUDA unified memory, and the use of the

`asm(trap)` construct. This combination of technologies enables precise and controlled intervention, allowing for the timely suspension of GPU kernel execution when needed.

(4) To improve accelerator utilization, Arax and previous works offer GPU spatial sharing that allows applications from different users to run concurrently on the same GPU. Spatial sharing is made possible by employing a single GPU context and utilizing streams. However, the use of a single context gives rise to memory protection challenges, as it involves a shared address space that is common to all users and applications. To facilitate protected GPU sharing we introduce Guardian. Guardian implements an address fencing technique inside the PTX kernel code that implies minimal overhead, making it a practical and efficient solution for enabling protected GPU sharing.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI'16*, 2016.

[2] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for nvidia gpgpus. In *HPEC'19*, 2019.

[3] Ben Ashbaugh, Alexey Bader, James Brodman, Jeff Hammond, Michael Kinsner, John Pennycook, Roland Schulz, and Jason Sewall. Data parallel c++ enhancing sycl through extensions for productivity and performance. In *International Workshop on OpenCL*, 2020.

[4] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. Virtualized execution runtime for fpga accelerators in the cloud. In *IEEE Access*, 2017.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par '09*, 2009.

[6] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. Exploiting inter-warp heterogeneity to improve gpgpu performance. In *PACT '15*, 2015.

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: a gpu memory manager with application-transparent support for multiple page sizes. In *MICRO '17*, 2017.

[8] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ASPLOS '18*, 2018.

[9] David F. Bacon, Rodric M. Rabbah, and Sunil Shukla. Fpga programming for the masses. In *Communications ACM*, 2013.

[10] M Bari, L Stoltzfus, P Lin, C Liao, M Emani, and B Chapman. Is data placement optimization still relevant on newer gpus? In *U.S. Department of Energy Office of Scientific and Technical Information*, 2018.

[11] Gaurav Batra, Zach Jacobson, Siddarth Madhav, Andrea Queirolo, and Nick Santhanam. Artificial-intelligence hardware: New opportunities for semiconductor companies. In *McKinsey & Company, New York, NY, USA, Tech. Rep*, 2018.

[12] Jürgen Becker, Michael Hübner, and Michael Ullmann. Run-time fpga reconfiguration for power-/cost-optimized real-time systems. In *VLSI-SOC: From Systems to Chips*, 2006.

[13] Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. Understanding and exploiting the internals of gpu resource allocation for critical systems. In *ICCAD'19*, 2019.

[14] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. Evaluation of intel's dpc++ compatibility tool in heterogeneous computing. In *PDC '22*, 2022.

[15] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *GLSVLSI '15*, 2015.

[16] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, N. Kwatra, and S. Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *EuroSys '20*, 2020.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*, 2009.

[18] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *ASPLOS '16*, 2016.

[19] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: Energy-efficient hardware accelerators for machine learning. In *MICRO '16*, 2016.

[20] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In *SoCC '20*, 2020.

[21] Bang Di, Jianhua Sun, and Hao Chen. A study of overflow vulnerabilities on gpus. In *NPC' 16*, 2016.

[22] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. Gmod: A dynamic gpu memory overflow detector. In *PACT '18*, 2018.

[23] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *HiPC '11*, 2011.

[24] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. Cricket: A virtualization layer for distributed execution of cuda applications with checkpoint/restart support. In *Concurrency and Computation: Practice and Experience*, 2022.

[25] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for gpgpus. In *CGO '17*, 2017.

[26] Jouppi Norman et. al. In-datacenter performance analysis of a tensor processing unit. In *ISCA '17*, 2017.

[27] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *CloudCom '15*, 2015.

[28] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. DGSF: Disaggregated GPUs for Serverless Functions. In *IPDPS '22*, 2022.

[29] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA '18*, 2018.

[30] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *MICRO '12*, 2012.

[31] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO'20*, 2020.

[32] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. Demystifying the placement policies of the nvidia gpu thread block scheduler for concurrent kernels. In *SIGMETRICS '21*, 2021.

[33] Vinod Grover and Yuan Lin. Compiling cuda and other languages for gpus. In *GTC '12*, 2012.

[34] Antonio Gulli and Sujit Pal. Deep learning with keras. Packt Publishing Ltd, 2017.

[35] Fan Guo, Yongkun Li, John C. S. Lui, and Yinlong Xu. DCUDA: Dynamic GPU Scheduling with Live Migration Support. In *SoCC '19*, 2019.

[36] K. Gupta, J. A. Stuart, and J. D. Owens. Gpu programming for gpgpu workloads. In *InPar '12*, 2012.

[37] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. Cox: Exposing cuda warplevel functions to cpus. In *TACO '22*, 2022.

[38] Ruobing Han, Blaise Tine, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. Supporting CUDA for an extended RISC-V GPU architecture. In *CoRR '21*, 2021.

[39] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. Decoding cuda binary. In *CGO '19*, 2019.

[40] Wen-Mei Hwu, Christopher Rodrigues, Shane Ryoo, and John Stratton. Compute unified device architecture application suitability. In *Computing in Science & Engineering*, 2009.

[41] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ArXiv*, 2014.

[42] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. In *ArXiv*, 2018.

[43] Stephen Jones. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation*, 2012.

[44] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX ATC'11*, 2011.

[45] Niklas Kerscher. Investigating the hip programming model with regards to portability and performance portability. In *ArXiv*, 2022.

[46] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Toronto, ON, Canada, 2009.

[47] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *EuroSys '18*, 2018.

[48] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: Compute allocation in hybrid clusters. In *EuroSys '20*, 2020.

[49] Cortes Lecun. The mnist database of handwritten digits, 2022.

[50] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[51] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing gpu via region-based bounds checking. In *ISCA '22*, 2022.

[52] Kyushick Lee, Michael B. Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W. Keckler, and Mattan Erez. Gpu snapshot: Checkpoint offloading for gpu-dense systems. In *ICS '19*, 2019.

[53] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *S&P '14*, 2014.

[54] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: Exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *SoCC '22*, 2022.

[55] Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. Automatic data placement into gpu on-chip memory resources. In *CGO '15*, 2015.

[56] Chen Li, Andrew Zigerelli, Jun Yang, and Yang Guo. Pep: proactive checkpointing for efficient preemption on gpus. In *DAC '18*, 2018.

[57] Teng Li, Vikram K Narayana, and Tarek El-Ghazawi. Symbiotic scheduling of concurrent gpu kernels for performance and energy optimizations. In *CF '14*, 2014.

[58] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS '15*, 2015.

[59] C. Lu, K. Ye, G. Xu, C. Z. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *Big Data '17*, 2017.

[60] Justin Luitjens. Cuda streams: Best practices and common pitfalls. In *GPU Techonology Conference*, 2015.

[61] HaoHui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. Honeycomb: Secure and efficient GPU executions via static validation. In *OSDI 23*, 2023.

[62] Pak Markthub, Mehmet Belviranli, Seyong Lee, Jeffrey Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *SC '18*, 2018.

[63] Stelios Mavridis, Manolis Pavlidakis, Ioannis Stamoulias, Christos Kozanitis, Nikolaos Chrysos, Christoforos Kachris, Dimitrios Soudris, and Angelos Bilas. Vinetalk: Simplifying software access and sharing of fpgas in datacenters. In *FPL' 17*, 2017.

[64] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *TCCA '95*, 1995.

[65] Andrea Miele. Buffer overflow vulnerabilities in cuda: a preliminary analysis. In *Journal of Computer Virology and Hacking Techniques*, 2015.

[66] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in Multi-Core systems. In *USENIX Security '07*, 2007.

[67] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. In *Parallel and Distributed Computing*, 2020.

[68] NVIDIA. Whitepaper pascal compute preemption. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016. Online; accessed 25 July 2016.

[69] NVIDIA. Multi-instance gpu. 2022.

[70] NVIDIA. Multi-process service. 2022.

[71] NVIDIA. Parallel thread execution isa, 2022.

[72] Meni Orenbach and Mark Silberstein. Enclaves as accelerators: learning lessons from gpu computing for designing efficient runtimes for enclaves.

[73] Nathan Otterness and James H. Anderson. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *ECRTS 2020*, 2020.

[74] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ASPLOS '13*, 2013.

[75] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *ASPLOS '15*, 2015.

[76] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with gpu memory exploitation. In *Computers and Security*, 2021.

[77] Alberto Parravicini, Davide B. Bartolini, Lukas Stadler, Arnaud Delamare, Marco Arnaboldi, and Marco Domenico Santambrogio. Automated gpu out-of-bound access detection and prevention in a managed environment. In *ArXiv*, 2015.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NIPs '19*, 2019.

[79] Manos Pavlidakis, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. Arax: A runtime framework for decoupling applications from heterogeneous accelerators. In *SoCC '22*, 2022.

[80] Manos Pavlidakis, Stelios Mavridis, Nikos Chrysos, and Angelos Bilas. Trem: A task revocation mechanism for gpus. In *HPCC '20*, 2020.

[81] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: A detailed hack for cuda and a (partial) fix. In *TECS '16*, 2016.

[82] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *ASPLOS '23*, 2023.

[83] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC '12*, 2012.

[84] Ruyman Reyes and Victor Lomüller. Sycl: Single-source c++ accelerator programming. In *Parallel Computing: On the Road to Exascale*, 2016.

[85] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Transparent acceleration for heterogeneous platforms with compilation to opencl. In *TACO '19*, 2019.

[86] Davide Rossetti and S Team. Gpudirect: Integrating the gpu with a network interface. In *GPU Technology Conference*, 2015.

[87] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. In *IJCV '15*, 2015.

[88] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *HPDC '13*, 2013.

[89] O. Segal, M. Margala, S. R. Chalamalasetti, and M. Wright. High level programming framework for fpgas in the data center. In *FPL '14*, 2014.

[90] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. In *CoRR*, 2015.

[91] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: A case for software address translation on gpus. In *ISCA '16*, 2016.

[92] Yakun Sophia Shao, Jason Cemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with chiplet-based architecture. In *MICRO '21*, 2021.

[93] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *IPDPS '09*, 2009.

[94] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA '13*, 2013.

[95] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. In *PLDI '23*, 2023.

[96] George Teodoro, Rafael Oliveira, Olcay Sertel, Metin Gurcan, Wagner Meira Jr, Umit Catalyurek, and Renato Ferreira. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *CLUSTER '09*, 2009.

[97] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *USENIX ATC'14*, 2014.

[98] Lukas Tobler. Gpuless–serverless gpu functions. In *Master Thesis ETH*, 2022.

[99] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *ISFPGA '19*, 2010.

[100] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity. In *ASCR Workshop on Extreme Heterogeneity*, 2018.

[101] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *MICRO '19*, 2019.

[102] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI '18*, 2018.

[103] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *HPCA '16*, 2016.

[104] Florian Wende, Thomas Steinke, and Frank Cordes. Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture. In *ArXiv*, 2014.

[105] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-level language tools for reconfigurable computing. In *Proceedings of the IEEE*, 2015.

[106] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *ASPLOS '17*, 2017.

[107] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc - an open-source gpgpu compiler. In *CGO '16*, 2016.

[108] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI '18*, 2018.

[109] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI '20*, 2020.

[110] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA '16*, 2016.

[111] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *ICS'19*, 2019.

[112] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *PPoPP '17*, 2017.

[113] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. Ava: Accelerated virtualization of accelerators. In *ASPLOS '20*, 2020.

[114] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. Vgris: virtualized gpu resource isolation and scheduling in cloud gaming. In *HPDC '13*, 2013.

[115] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-net: Effective gpu sharing in nfv systems. In *NSDI'18*, 2018.

[116] Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: A preemptive execution system for gpgpu computing. In *RTAS '15*, 2015.