Arax: A Runtime Framework for Decoupling Applications from Heterogeneous Accelerators

Manos Pavlidakis[1,2], Stelios Mavridis[1], Antony Chazapis[1], Giorgos Vasiliadis[1], and Angelos Bilas[1,2]

[1]Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), Greece
[2]Computer Science Department, University of Crete, Greece
{manospavl,mavridis,chazapis,gvasil,bilas}@ics.forth.gr

ABSTRACT

Today, using multiple heterogeneous accelerators efficiently from applications and high-level frameworks, such as Tensor-Flow and Caffe, poses significant challenges in three respects: (a) sharing accelerators, (b) allocating available resources elastically during application execution, and (c) reducing the required programming effort.

In this paper, we present Arax, a runtime system that decouples applications from heterogeneous accelerators within a server. First, Arax maps application tasks dynamically to available resources, managing all required task state, memory allocations, and task dependencies. As a result, Arax can share accelerators across applications in a server and adjust the resources used by each application as load fluctuates over time. Additionally, Arax offers a simple API and includes Autotalk, a stub generator that automatically generates stub libraries for applications already written for specific accelerator types, such as NVIDIA GPUs. Consequently, Arax applications are written once without considering physical details, including the number and type of accelerators.

Our results show that applications, such as Caffe, TensorFlow, and Rodinia, can run using Arax with minimum effort and low overhead compared to native execution, about 12% (geometric mean). Arax supports efficient accelerator sharing, by offering up to 20% improved execution times compared to NVIDIA MPS, which supports NVIDIA GPUs only. Arax can transparently provide elasticity, decreasing total application turn-around time by up to 2× compared to native execution without elasticity support.

KEYWORDS

Heterogeneous accelerators, Spatial sharing, Dynamic resource assignment, Live-migration

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 [1, 8, 10, 27, 31, 32]. Recent estimates [1, 2, 27, 33, 36] indicate that by 2030 servers will include a plethora of processing units and specialized accelerators [3, 6, 18, 29]. This trend poses significant challenges in how applications and higher-level frameworks, such as TensorFlow [9] and Caffe [14], 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.* 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 [11] and oneAPI [13], 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 still bound to a specific set of accelerators or accelerator 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.

First, 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. Existing time-sharing approaches [12, 34–36] cannot address this issue effectively, e.g., in cases where an application cannot fully utilize an accelerator during its time slice. Spatial sharing, on the other hand, has the potential to increase resource utilization. However, existing approaches, such as NVIDIA MPS [23], are limited to specific accelerator types and require applications to perform manual task assignment and data placement.

Second, resources assigned to each application remain fixed throughout its execution. However, applications often exhibit dynamic behavior and fluctuating load requirements [12, 34]. Given that it is difficult to estimate the resource demands of applications accurately and statically assign resources to each application, the *lack of elasticity mechanisms* results in application under- or over-provisioning and eventually to poor resource utilization.

In this paper, we present Arax, a runtime system that decouples applications from heterogeneous accelerators *within*

1

Capabilities	MPS [23]	StarPU [1]	Gandiva [34]	DCUDA [12]	AvA [36]	Arax
Heterogeneity	-	✓	-	-	✓	✓
Spatial sharing	✓	-	-	-	-	✓
Dynamic resource assign.	-	-	✓	✓	-	✓
Reducing effort	-	-	-	-	✓	✓

Table 1: Capabilities of Arax vs. state-of-the-art approaches.

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. 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. The core component of Arax is a backend service, the Arax server, that dynamically maps application tasks and data to available accelerators at runtime. This enables spatial accelerator sharing and adjusts resources at runtime. Last but not least, Arax includes a stub generator (Autotalk) that reduces porting effort for existing accelerator-enabled applications. Table 1 summarizes the main capabilities of Arax, compared to state-of-the-art approaches. The whole Arax ecosystem is available at GitHub[1].

The RPC-based approach of Arax allows **decoupling accelerators from applications**. Arax applications do not need to perform accelerator selection, memory allocation, or task assignment operations; all are handled transparently by Arax. This approach allows Arax to perform memory allocations lazily and only when the actual task assignment occurs. 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. 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 [36], result in up to 30% overhead for the same applications. 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 main contributions of this paper 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.
(5) We demonstrate and evaluate Arax in an accelerator-rich server environment, using GPUs, FPGAs, and CPUs, with Caffe, TensorFlow, and Rodinia.

2 DESIGN

Figure 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.
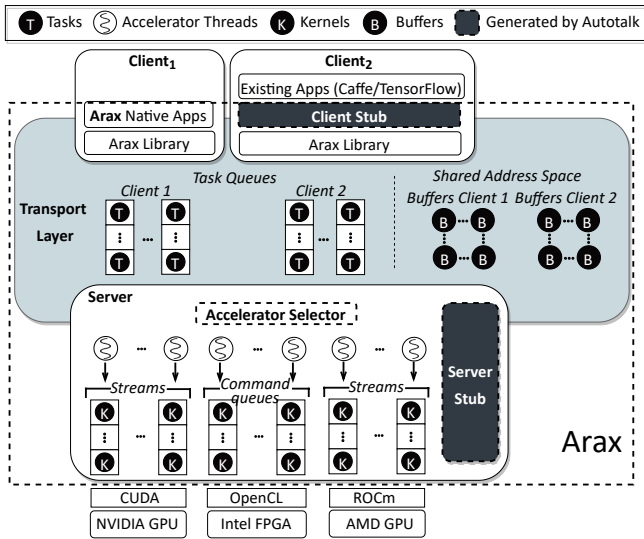
---
[1]https://github.com/CARV-ICS-FORTH/arax

Figure 1: Arax high-level overview. The main components of Arax are: Clients, Server, Transport layer, and Autotalk.

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

Table 2: Methods of Arax API.

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 Client

Arax provides three basic abstractions: (a) *tasks*, (b) *task buffers*, and (c) *task queues*. Table 2 shows an overview of the main Arax API calls.

Tasks. A task can be either a compute or a transfer task. A compute task is an accelerator kernel, while a transfer task is a data transfer between the host and the accelerator. 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 (§2.2). Unlike existing accelerator APIs, task arguments do not include accelerator-specific information, such as thread number or thread block 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 represents the input and output data of a task. 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 (§2.2). 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 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 (§2.2), 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.

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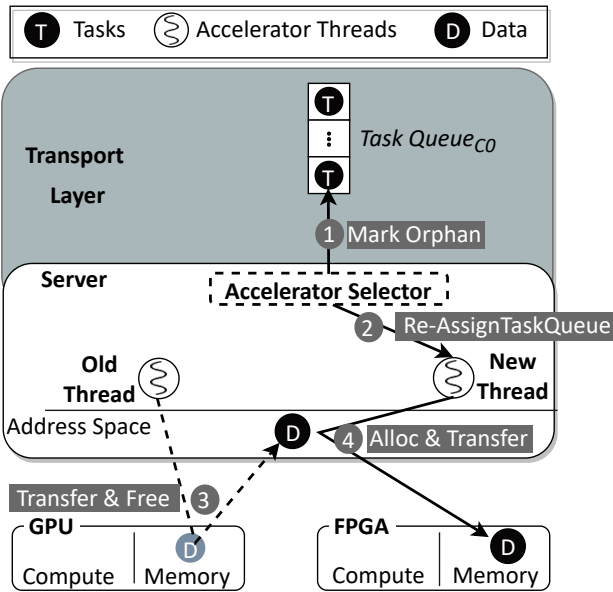
Figure 2: The steps required for an application migration. The task queue is marked orphan (1) and reassigned to a new thread (2). The relevant data are then transferred to the new accelerator via the server memory (3,4).

FPGAs, the Arax server loads a bitstream that contains multiple kernels, similar to Vinetalk [19]. 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; step ❶). 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; 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 [4, 34] 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; step ❸). The server then notifies the new accelerator thread to allocate and copy that data from the server's memory (Figure 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 [24].

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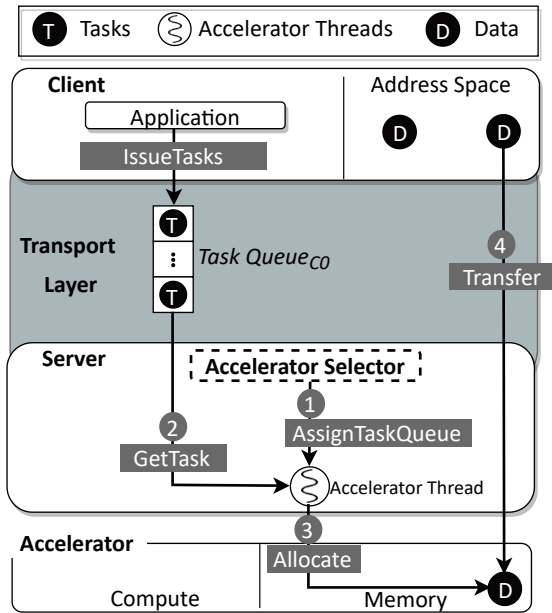
Figure 3: Arax dynamic task assignment. Application issues tasks to a task queue. Initially, the task queue is assigned to an accelerator (1), then the accelerator thread gets a task (2). It allocates accelerator memory for that data (3) and copies the data from the application (4).

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 3; step ❶). After the selection of the physical accelerator, the thread of that accelerator gets a task from the task queue (Figure 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 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 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 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.

2.3 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 4.1. We believe that future versions of Arax should consider zero-copy mechanisms by using shared pointers between the application and server address spaces.

2.4 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 that creates client and server stubs for each accelerator API offline (Figure 4; Offline). The generated stubs are linked with the applications and the Arax server during their initialization (Figure 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 4; API specifications). The specification file contains for each API call, the number of arguments, their order, and
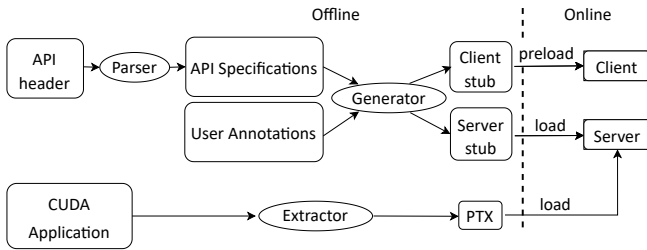
Figure 4: Client and Server stub generation (offline phase) and loading (online phase). The three steps of the offline phase are performed by the parser, the generator, and the extractor.

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 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 [21] to extract kernels from the native CUDA applications that are not included in accelerator libraries (Figure 4; Extractor); these kernels are in PTX format [25] and are dynamically linked with the server executable so they can be invoked at runtime.

2.5 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.

3 EXPERIMENTAL METHODOLOGY

For our evaluation, we use two servers with different accelerator types, as shown in Table 3. 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 use ROCm v4.1.0.

ID	CPU	RAM (GB)	PCIe Gen	Accelerators
S1	AMD EPYC 7551P 32-Core @ 3.0GHz	128	3.0	NVIDIA RTX 4000, Intel Arria 10, AMD RX550X
S2	Intel Xeon CPU E5-2620 8-Core @ 2.10GHz	256	3.0	2x NVIDIA RTX 2080 Ti

Table 3: Servers configurations.

Suite	App.	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	CV	3316	3216	
	GDL	3974	3871	
	GNN	2784	2780	
	RS	5310	5310	

Table 4: Applications and their memory footprint.

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 5: Workloads for spatial sharing.

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 4 shows the real-world applications and their inputs/outputs used for our evaluation. Similar to AvA [36], we use applications from Rodinia [5] as well as model training and inference from Caffe [14] and TensorFlow [9] version 2.3.2. The last column of Table 4 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 [28]. For TensorFlow Mnist [17] we use the dataset in LeCun et. all [16]. For Keras, we use Computer Vision (CV), Generative Deep Learning (GDL), Graph Neural Networks (GNN), and a Recommendation System (RS) applications, with the code and datasets provided in the Keras repository [15]. Regarding Rodinia datasets, we increase their size by 10× and the kernel execution time by 8×, compared to previous works [36] 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 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.

4 EXPERIMENTAL EVALUATION

Our evaluation tries to answer the following questions:

- What is the overhead of Arax for decoupling applications from accelerators (§4.1)?
- How effective is accelerator sharing in Arax (§4.2)?
- What is the performance improvement of elasticity (§4.3)?
- What is the overhead of application migration (§4.4)?
- What is the overhead introduced by Arax in real-life ML frameworks (§4.5)?

4.1 Overhead of accelerator decoupling

In this section, we evaluate the performance of Arax with heterogeneous accelerators. We use Rodinia [5], which offers OpenCL, ROCm, and CUDA kernels. To execute Rodinia in Arax, we port the host code of its CUDA version. Figure 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.

Figure 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 5(b) and Figure 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 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).

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 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 [20] 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 [36], which is a state-of-the-art framework for heterogeneous accelerators. Figure 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
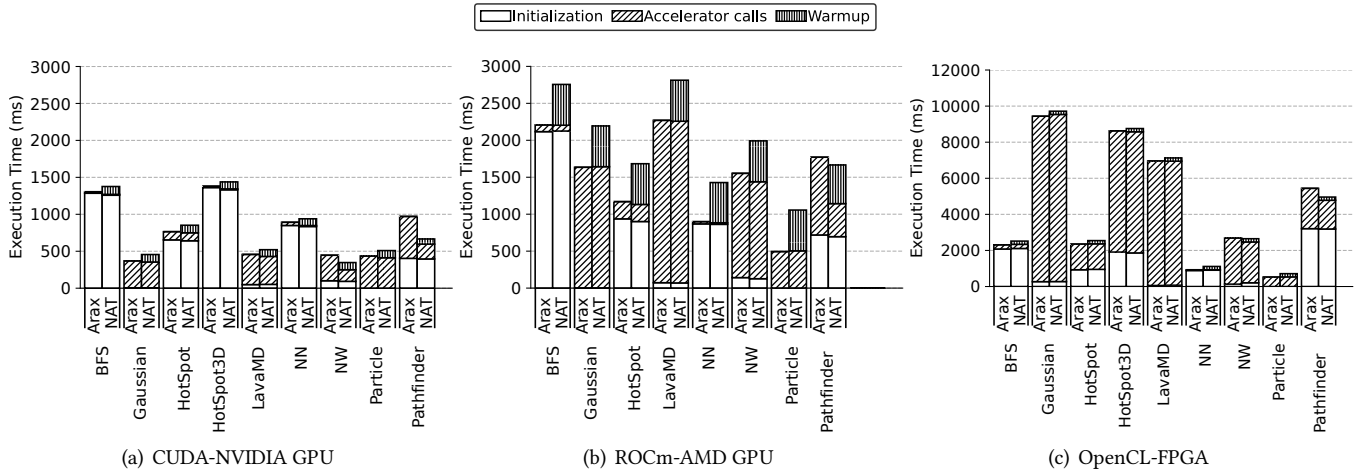
Figure 5: Overhead of Arax compared to native (NAT) using Rodinia benchmarks over heterogeneous accelerators.
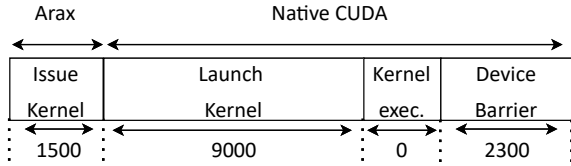

Figure 6: Breakdown of overhead for launching an empty kernel with Arax (CPU cycles).
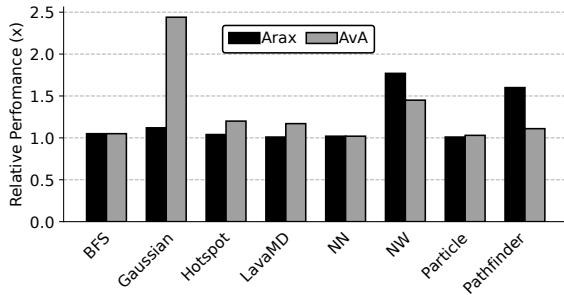

Figure 7: Execution time normalized to native for Arax and AvA.

Figure 8: Effectiveness of sharing with NVIDIA GPUs for Arax, native (without MPS), and MPS.

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.

4.2 Effectiveness of accelerator sharing

We now compare Arax sharing with NVIDIA MPS [23], 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 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. 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
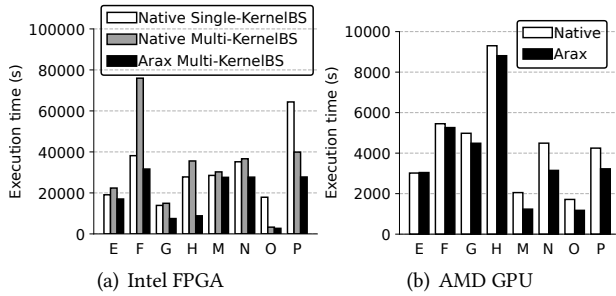
(a) Intel FPGA   (b) AMD GPU

Figure 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.

this, due to the closed-source nature of NVIDIA MPS, we run further micro-benchmarks 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[2].

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 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 9(a); Arax *Multi-KernelBS*) decreases execution time from 3% up to 85% compared to the single kernel bitstream (Figure 9(a); Native *Single-KernelBS*) and between 9% and 75% compared to

the multi-kernel bitstream (Figure 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 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.

4.3 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 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 3, 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.

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.

---

[2]ID 3559606, ID 3350973

Figure 10: Performance improvement of applications when increasing the number of *homogeneous* accelerators or GPU streams.
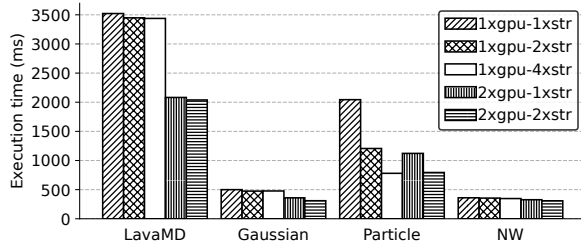

Figure 11: Performance improvement of applications when increasing the number of *heterogeneous* accelerators or GPU streams.

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 11 shows the execution times of four representative applications using multiple heterogeneous accelerators. Each application is running with the following configurations: (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.

As shown in Figure 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+2xstrNVIDIA+AMD* and *FPGA+NVIDIA+2xstrAMD* 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
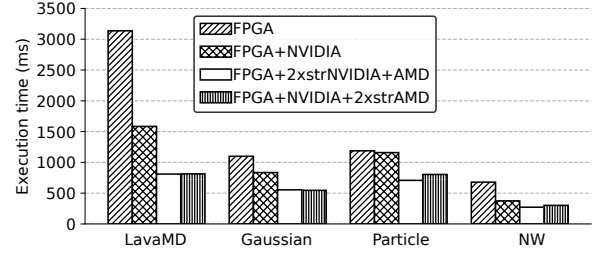
add more accelerators, shown as *FPGA+2strNVIDIA+AMD* and *FPGA+NVIDIA+2strAMD*, the performance increases by 1.5× compared to the *FPGA+NVIDIA* setup.

4.4 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.

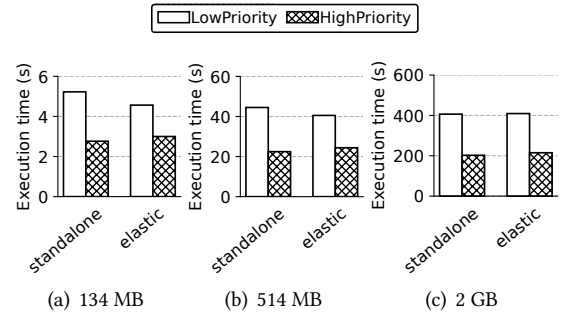(a) 134 MB   (b) 514 MB   (c) 2 GB

Figure 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 §2.2. 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 12(a), 12(b), and 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 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 [26] to reduce the waiting time of our migration mechanism, but this is beyond the purpose of this paper.

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 task queue to another accelerator that supports this kernel. The task queue

	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 6: 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.

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 6 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.

4.5 Overhead for Caffe and TensorFlow

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. Figure 13 shows 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 13(a)). The relative performance of manual-porting compared to native CUDA is between 3% and 17%. With more than ten epochs, as Figure 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 13(e) 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%.
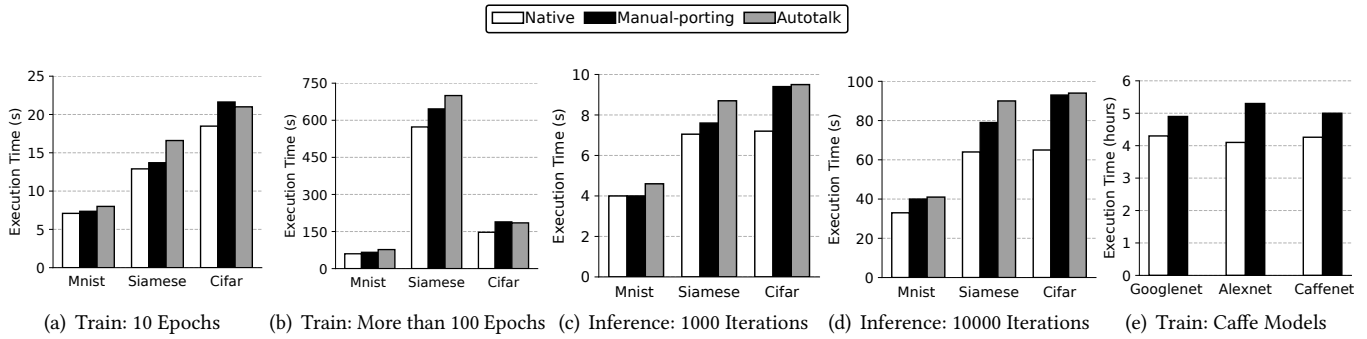
Figure 13: The overheads of Arax using manual-porting and Autotalk (automatic stub generation) compared to native CUDA for Caffe with varying epochs and iterations.

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

Table 7: The execution time (seconds) of TensorFlow and Keras for Autotalk and native CUDA.

Figures 13(c) and 13(d) 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 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 7. Our findings suggest that Arax and Autotalk can transparently handle complex, real-life frameworks without significant effort.

5 RELATED WORK

We categorize related work in four areas: (a) static accelerator assignment, (b) dynamic accelerator assignment, (c) accelerator virtualization, and (d) accelerator spatial sharing.

Existing programming models, such as CUDA [22], SYCL [11], and oneAPI [13], 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 [1] 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 applications. Gandiva [34] is a cluster-level scheduler for ML training applications that dynamically assigns GPUs to applications. DCUDA [12] 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 [4]. 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 [7, 30, 36]. API remoting [7, 30] is an I/O virtualization technique in which API calls are forwarded to a user-level computing framework [30] or to a remote server [7]. The main disadvantage of API remoting is the inability to support multiple APIs, which is not the case for Arax. AvA [36] 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 we show in our evaluation. Arax frees applications from accelerator selection, allowing dynamic task assignment. By creating a single GPU context, our server enables spatial sharing.

Finally, GPUs support spatial sharing through NVIDIA MPS [23], 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., in the case of 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 of 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 that can 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   CONCLUSIONS

In this paper, we present 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.

The extra data copy in the Arax transport layer introduces 80% overhead for applications with low computation to communication ratio. Consequently, future work should examine optimizations for zero-copy data transfers across application, server, and accelerator address spaces. In addition, mechanisms for low-overhead, on-demand data transfer across accelerators when using arbitrary pointers as task arguments can further reduce data transfers during task migrations.

ACKNOWLEDGMENTS

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par '09*.

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

[3] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. 2015. Origami: A Convolutional Network Accelerator. In *GLSVLSI '15*.

[4] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, N. Kwatra, and S. Viswanatha. 2020. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys '20*.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC '09*.

[6] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning. In *MICRO '16*.

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

[8] Jouppi Norman et. al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA '17*.

[9] Martín Abadi et. al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[10] 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. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *ISCA '18*.

[11] Kronos Group. 2022. SYCL2020. Retrieved September 2022 from https://www.khronos.org/sycl/

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

[13] Intel. 2020. oneAPI. Retrieved September 2022 from https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.4ac4fz

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *arXiv*.

[15] Keras. 2014. Keras Code Examples. Retrieved September 2022 from https://keras.io/examples/

[16] Cortes Lecun. 2022. The mnist database of handwritten digits. Retrieved September 2022 from http://yann.lecun.com/exdb/mnist

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE*.

[18] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, X. Zhou, and Y. Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *ASPLOS '15*.

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

[20] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. In *TCCA '95*.

[21] NVIDIA. 2021. CUDA Binary Utilities. Retrieved September 2022 from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[22] NVIDIA. 2022. CUDA: Compute Unified Device Architecture. Retrieved Sep. 2022 from https://developer.nvidia.com/cuda-toolkit

[23] NVIDIA. 2022. Multi-Process Service. Retrieved September 2022 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[24] NVIDIA. 2022. NVIDIA GPUDirect. Retrieved September 2022 from https://developer.nvidia.com/gpudirect

[25] NVIDIA. 2022. Parallel Thread Execution ISA. Retrieved September 2022 from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[26] Manos Pavlidakis, Stelios Mavridis, Nikos Chrysos, and Angelos Bilas. 2020. TReM: A Task Revocation Mechanism for GPUs. In *HPCC'20*.

[27] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. 2019. Transparent acceleration for heterogeneous platforms with compilation to OpenCL. *TACO '19* (2019).

[28] 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. 2015. ImageNet Large Scale Visual Recognition Challenge. *IJCV '15*.

[29] 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. 2021. Simba: Scaling Deep-Learning Inference with Chiplet-Based Architecture. In *MICRO '21*.

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

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

[32] Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *ISFPGA '19*.

[33] 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. 2018. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity. In *ASCR Workshop on Extreme Heterogeneity*.

[34] 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. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI '18*.

[35] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *OSDI '20*.

[36] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *ASPLOS '20*.