Arax: A Runtime Framework for Decoupling Applications from Heterogeneous Accelerators

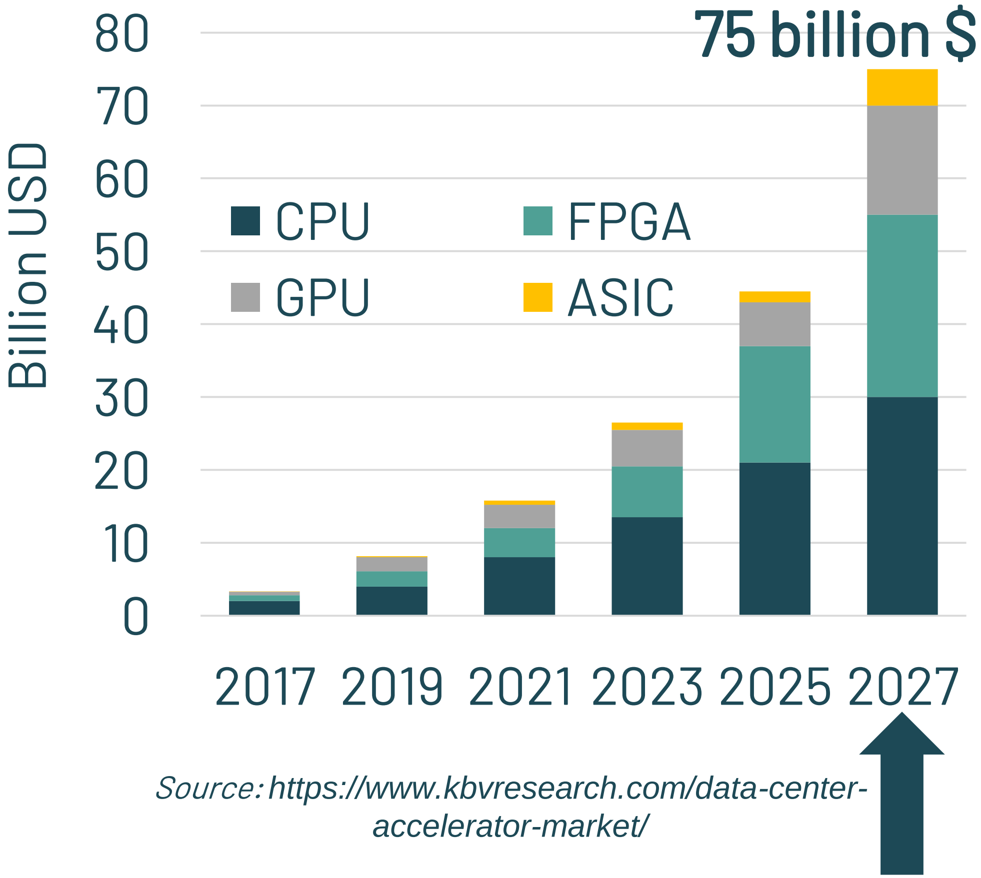
Manos Pavlidakis[1,2], Stelios Mavridis[1], Antony Chazapis[1], Giorgos Vasiliadis[1], and Angelos Bilas[1,2]

[1] Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece
[2] Computer Science Department, University of Crete, Greece

Use of heterogeneous accelerators increases

- The use of <u>**accelerators increases**</u>
  - Need for high performance at low energy consumption
- Accelerator <u>**heterogeneity**</u> increases [1, 2]
  - Different applications have different needs
  - Inference → CPU, ASIC
  - Training → GPU, FPGA

Datacenter accelerator market size



Source: https://www.kbvresearch.com/data-center-accelerator-market/

[1] DOE ASCR Basic Research Needs Workshop 2018, Extreme Heterogeneity
[2] HPCA 2018, Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective

Challenge: Transparent use of multiple/heterogeneous accelerators

- Unified programming models (HIP, SYCL, OpenCL) aim for write-once code
  - They allow compiling the same source code for different accelerators

- Static accelerator selection at app initialization time for the whole execution
  - External schedulers are static in a similar manner

- Static selection leads to accelerator under-utilization due to
  - Reduced accelerator sharing
  - Lack of adaptation during execution (elasticity)

- Dynamically selecting accelerators at runtime requires
  - Significant effort for application writing
  - Global scheduling decisions across applications

Arax

- A runtime for managing **multiple & heterogeneous** accelerators **within a server**
  - RPC-based approach to abstract accelerators
  - Shared runtime for all applications running in a server

- Arax offers **transparent** mechanisms for
  - Dynamic task assignment
  - Lazy data placement
  - Spatial accelerator sharing across applications
  - Automatic stub generation

Why Arax?

Capabilities	MPS (NVIDIA)	StarPU (Europar'09)	Gandiva (OSDI'18)	DCUDA (SoCC'19)	AvA (ASPLOS'20)
Abstract accelerators	–	✓	–	–	✓
Shared runtime	✓	–	✓	✓	–
Dynamic task assignment	–	–	✓ (app)	✓ (app)	–
Live data migration	–	–	✓	✓	–
Spatial sharing	✓	–	–	–	–
Automated porting	N.A.	–	N.A.	N.A.	✓

Why Arax?

Capabilities	MPS (NVIDIA)	StarPU (Europar'09)	Gandiva (OSDI'18)	DCUDA (SoCC'19)	AvA (ASPLOS'20)	Arax (SoCC'22)
Abstract accelerators	-	✓	-	-	✓	✓
Shared runtime	✓	-	✓	✓	-	✓
Dynamic task assignment	-	-	✓ (app)	✓ (app)	-	✓
Live data migration	-	-	✓	✓	-	✓
Spatial sharing	✓	-	-	-	-	✓
Automated porting	N.A.	-	N.A.	N.A.	✓	✓

Outline

- Motivation and overview
- <u>Design</u>
  - Abstraction primitives
  - Global resource management
  - Dynamic task assignment
  - Lazy data placement
  - Spatial accelerator sharing
  - Automatic stub generation
- Evaluation
- Conclusions

Abstraction primitives

✓ Goal: <u>Hide</u> accelerator <u>types</u> from applications

Arax Application

- Arax uses three main primitives

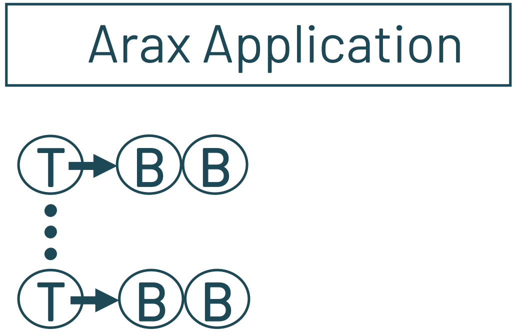
Abstraction primitives

✓ Goal: <u>Hide</u> accelerator <u>types</u> from applications

- Arax uses three main primitives

1. Tasks (Ⓣ): hide accelerator-specific information
   - Represent individual kernels and data transfers
   - Fine-grain in the range of milliseconds

Abstraction primitives



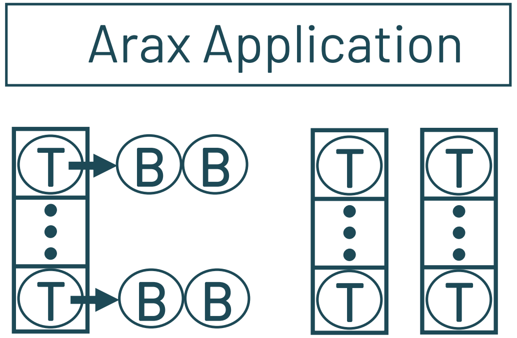
✓ Goal: <u>Hide</u> accelerator <u>types</u> from applications

- Arax uses three main primitives

1. Tasks (Ⓣ): hide accelerator-specific information
   - Represent individual kernels and data transfers
   - Fine-grain in the range of milliseconds
2. Buffers (Ⓑ): hide accelerator memory
   - Opaque identifiers that represent task input/output data
   - Used to keep track of data dependencies in Arax

Abstraction primitives

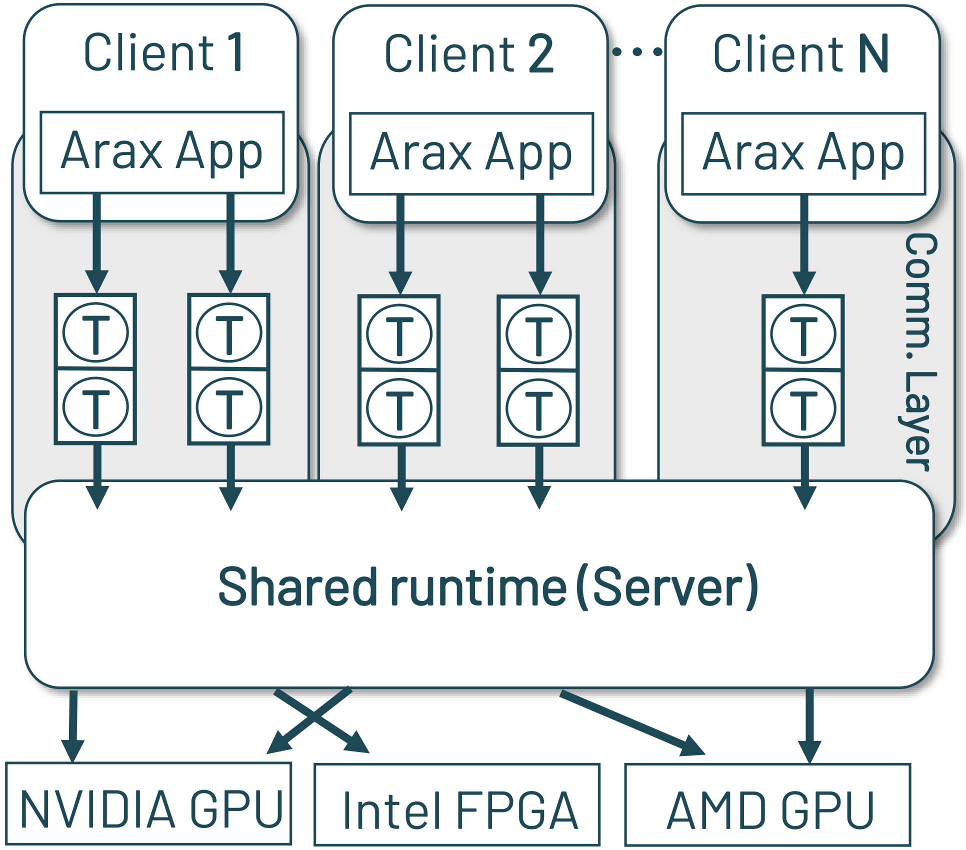
✓ Goal: <u>Hide</u> accelerator <u>types</u> from applications



- Arax uses three main primitives

1. Tasks (Ⓣ): hide accelerator-specific information
   - Represent individual kernels and data transfers
   - Fine-grain in the range of milliseconds
2. Buffers (Ⓑ): hide accelerator memory
   - Opaque identifiers that represent task input/output data
   - Used to keep track of data dependencies in Arax
3. Task Queues (▯): express task order
   - Arax ensures in-order execution in each queue
   - Applications can allocate several queues for concurrency

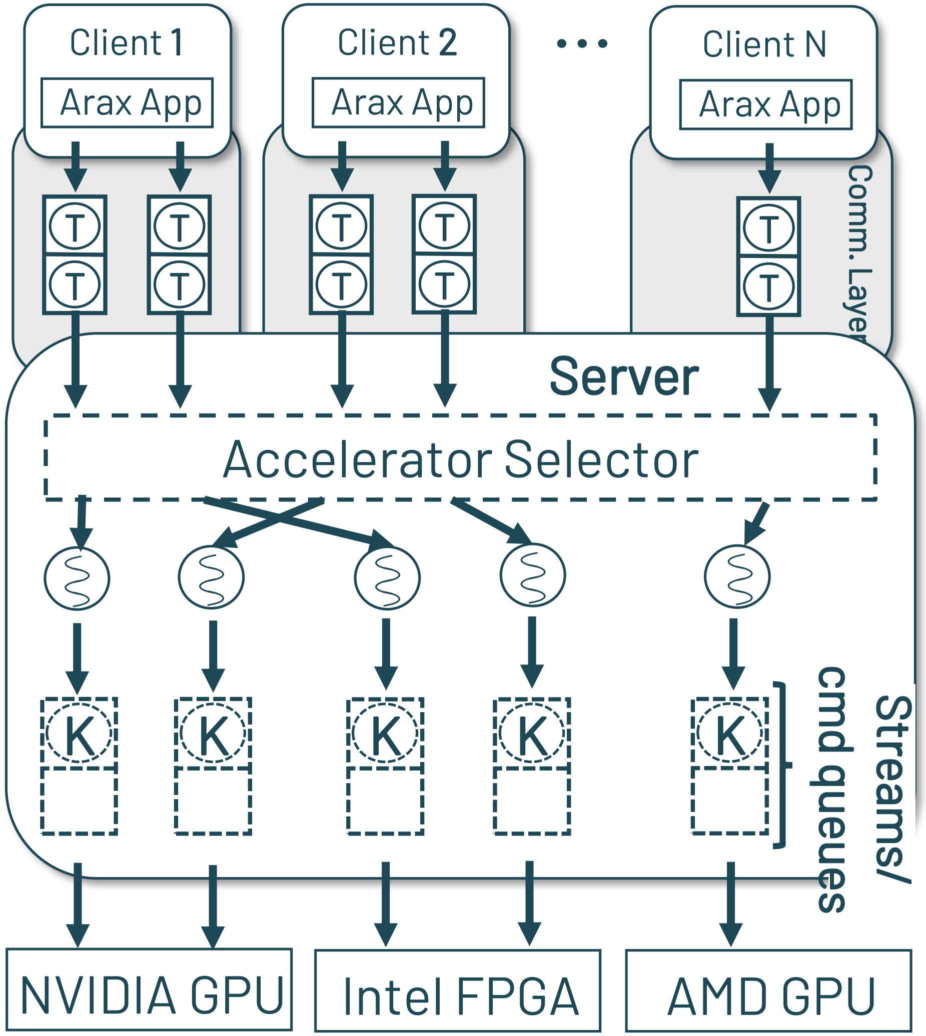
Global resource management across applications



✓ Goal: Optimize accelerator use across applications

- Arax uses a **shared runtime** process for **all apps**
  - Each application runs in a separate address space
  - The runtime (server) has a global view of apps & accelerators
- Arax uses **shared memory** for **communication**
  - Task and buffer synchronization → Mutexes/Spin locks
  - Allocation of in-transit buffers → Reference counters
  - Tracking of data location → Metadata per buffer

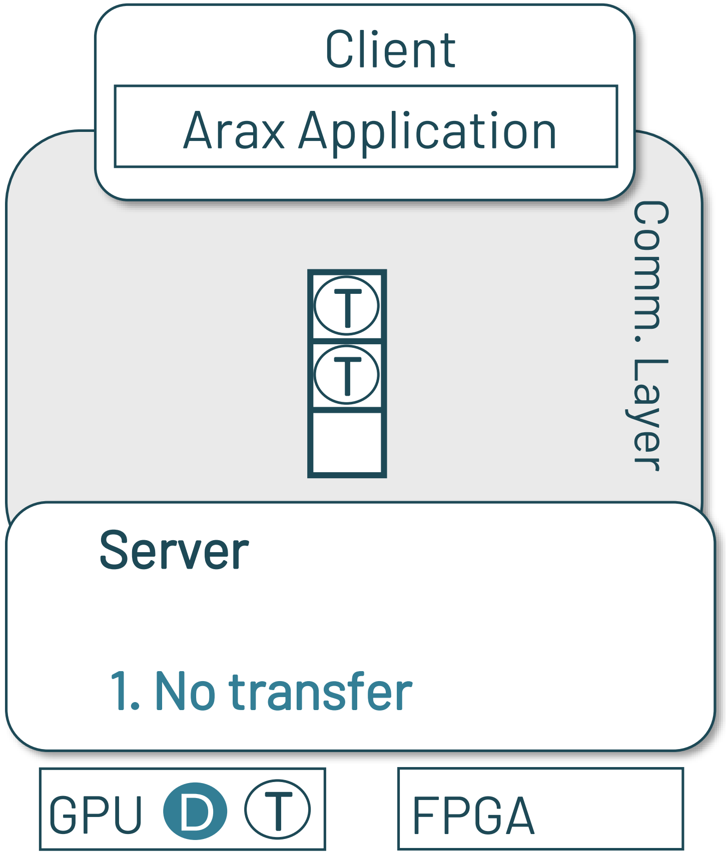
Dynamic task assignment at runtime



✓ Goal: <u>Adaptation</u> to application <u>load change</u>

- Arax moves <u>all</u> task management to the <u>server</u>
  - Select accelerator, transfer data, issue kernel, manage memory
  - Applications only issue tasks
- Arax performs <u>late</u> task assignment
  - Native: **Assignment** → <u>Issue</u> → Execution
  - Arax: <u>Issue</u> → **Assignment** → Execution
- Arax server
  - Hold kernels per accelerator → Kernel registry
  - Identifies appropriate accelerator → Policies
  - Handles thousands of tasks & queues → Multi-threaded
  - Maintains task order → Mapping tasks to streams/cmd queues

Lazy data placement

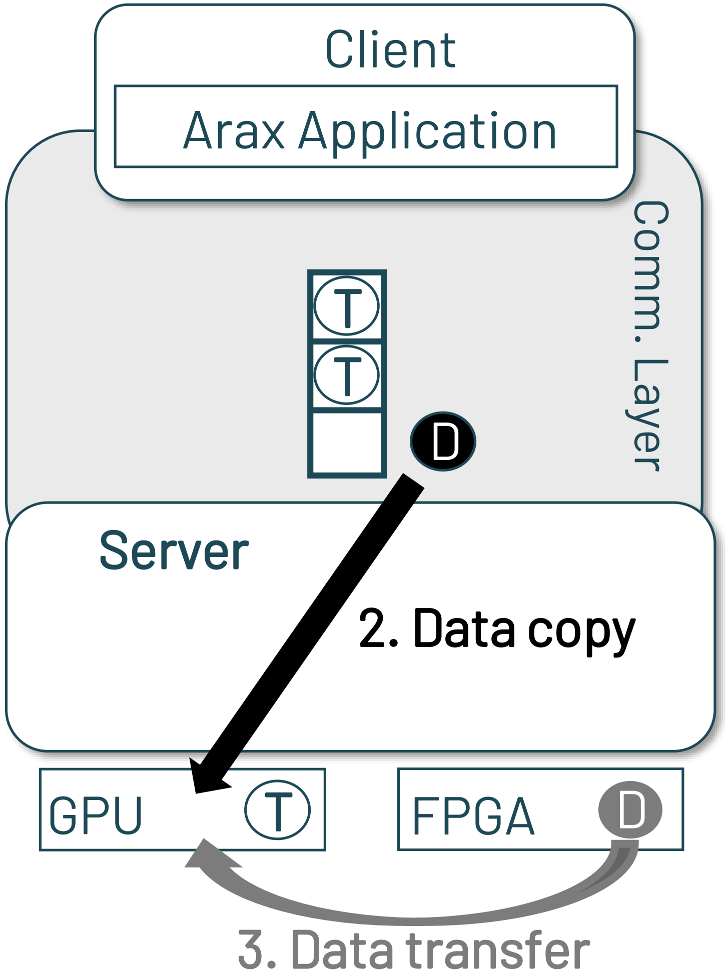


✓ Goal: Flexibility in task placement

- Prepare data for task execution **lazily**

1. Same accelerator → No transfer

Lazy data placement



✓ Goal: Flexibility in task placement

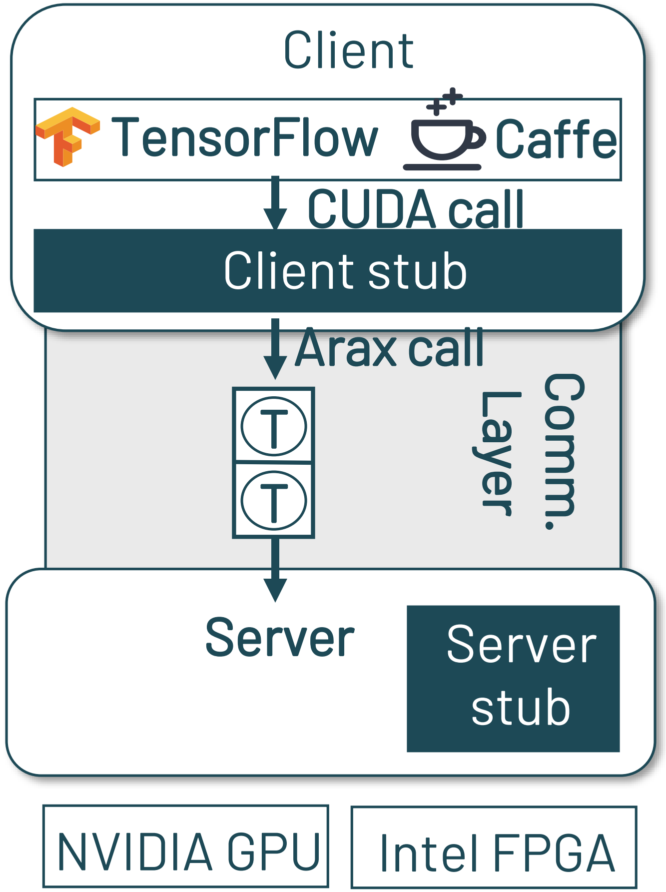
- Prepare data for task execution **lazily**
  1. Same accelerator → No transfer
  2. Staging area → Data copy (HostToDevice)
  3. Other accelerator → Data transfer (DeviceToDevice)

Spatial sharing

✓ Goal: Collocate tasks from different apps on the same accelerator

- Each accelerator has a mechanism for spatial sharing
  - GPUs → streams
  - FPGAs → multi-kernel bitstreams and command queues
- Arax **unifies** and **hides** these mechanisms
  - Reconfigures FPGAs depending on concurrently executing kernels
  - Uses a single CUDA context for all streams in each NVIDIA GPU

Automatic stub generation



✓ Goal: Reduce porting effort
  - To modify apps for Arax (we target CUDA)
  - To add a new accelerator and its kernels under Arax

- Arax provides tools to generate client & server stubs
  - Client stubs translate CUDA to Arax calls
  - Server stubs are wrappers for existing accelerator kernels
  - Most CUDA calls translate to a single Arax call that invokes kernels
- Reality is more complicated → fat binaries
  - In CUDA, host and kernel code are included in a single binary
  - Arax tools extract automatically kernels offline for loading in server
- We successfully run TensorFlow+Keras, Caffe
  - With tasks executing on CPU, GPU, FPGA

Outline

- Motivation and overview
- Design
  - Abstraction primitives
  - Global resource management
  - Dynamic task assignment
  - Lazy data placement
  - Spatial accelerator sharing
  - Automatic stub generation
- <u>Evaluation</u>
- Conclusions

Testbed

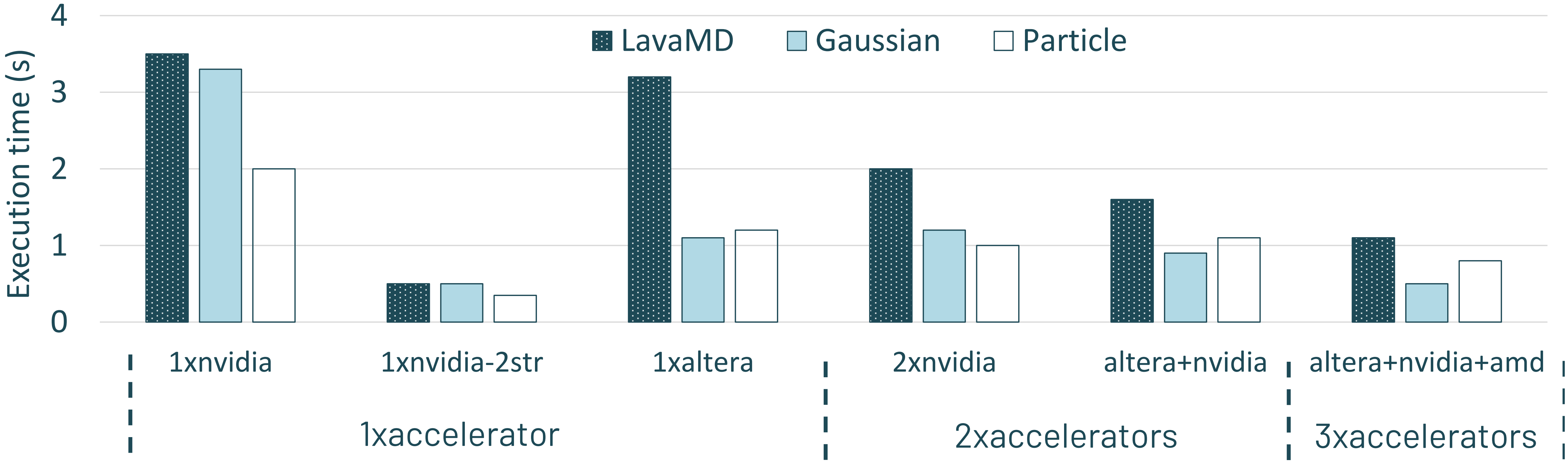
- Two server configurations with different accelerator types
    1. NVIDIA GPU, AMD GPU, and Intel FPGA
    2. Two RTX 2080 NVIDIA GPUs

- Microbenchmarks and real-world applications
    - Rodinia heterogenous benchmarks suite
    - Caffe deep learning framework
    - TensorFlow+Keras machine learning framework

- **We port applications to Arax once**
    - **Arax transparently manages accelerators in each configuration**
    - **Applications execute unmodified with different resources**

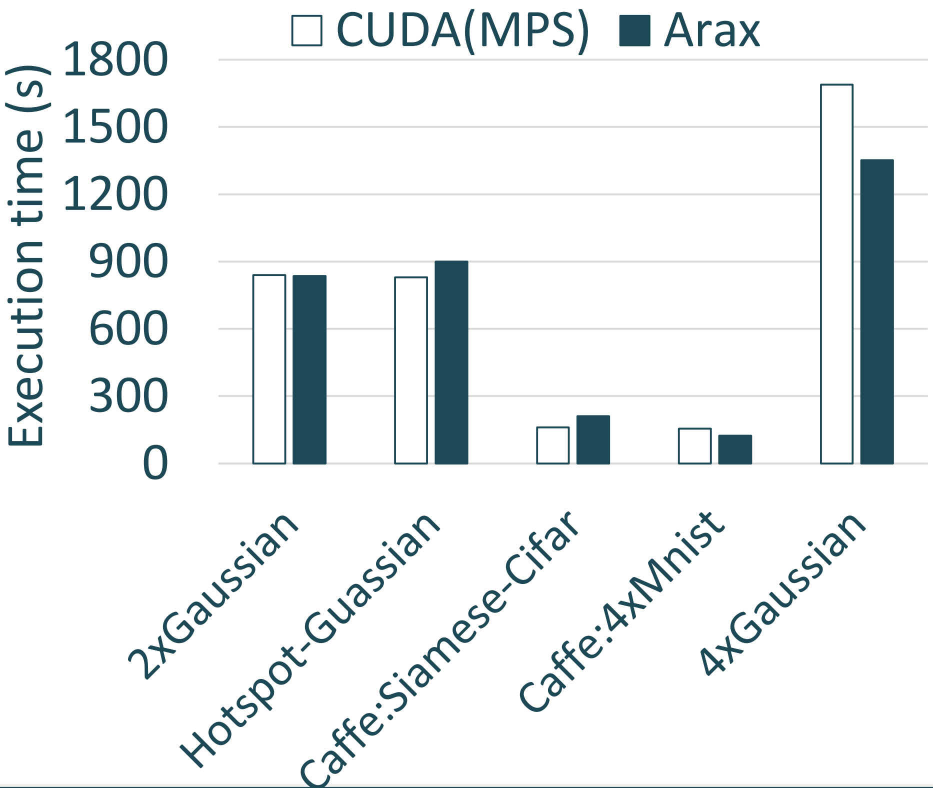
Use of multiple and heterogeneous accelerators

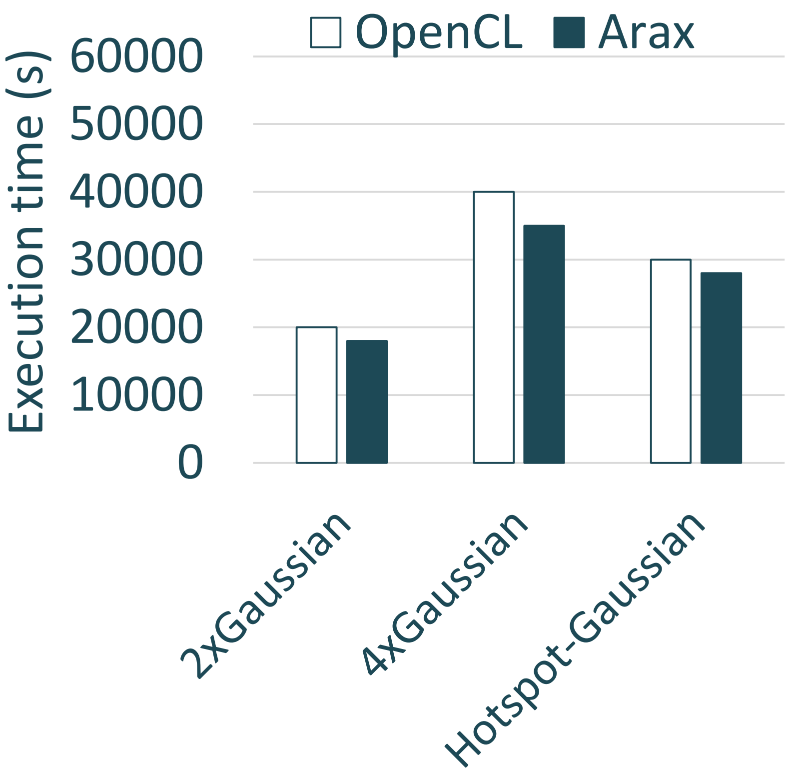
- Rodinia on **multiple** accelerators of the **same** and **different** types
  - Transparently, no application modifications

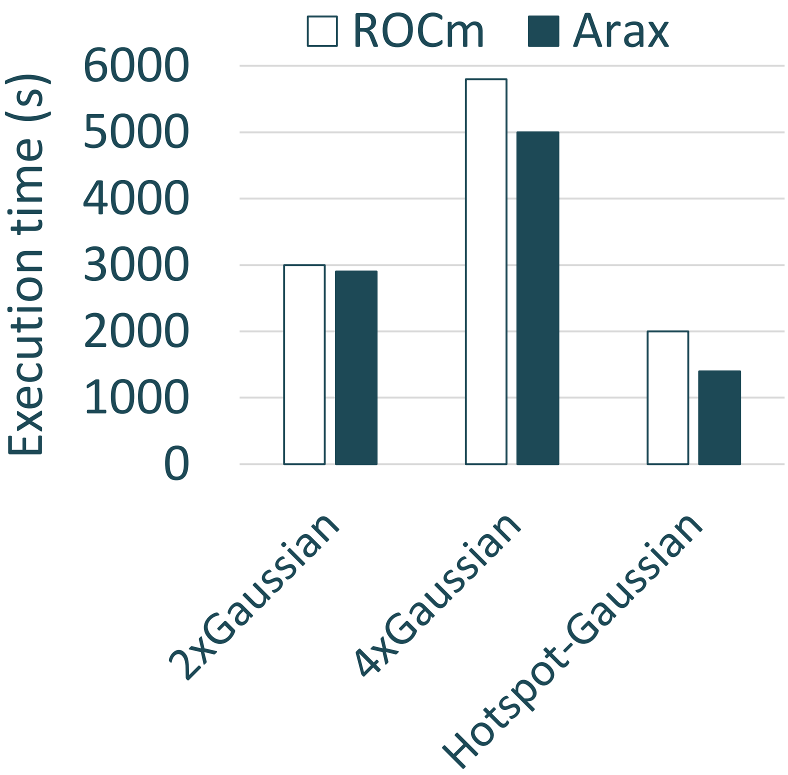


Spatial sharing

- Rodinia and Caffe sharing a single accelerator (NVIDIA, FPGA, AMD)
  - Several mixes of microbenchmarks with and without Caffe
  - **Comparable performance to native spatial sharing mechanisms**

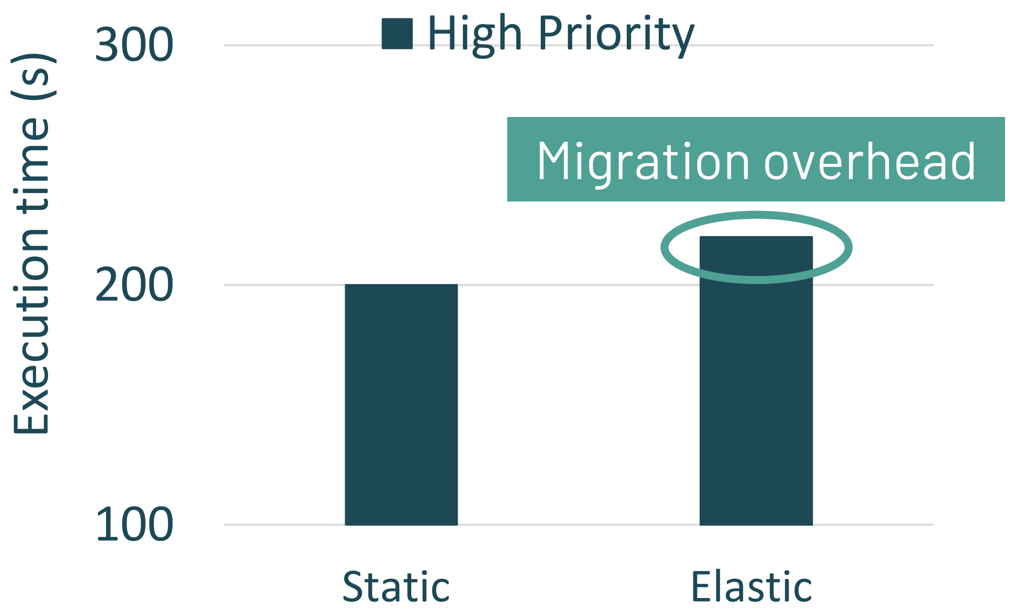


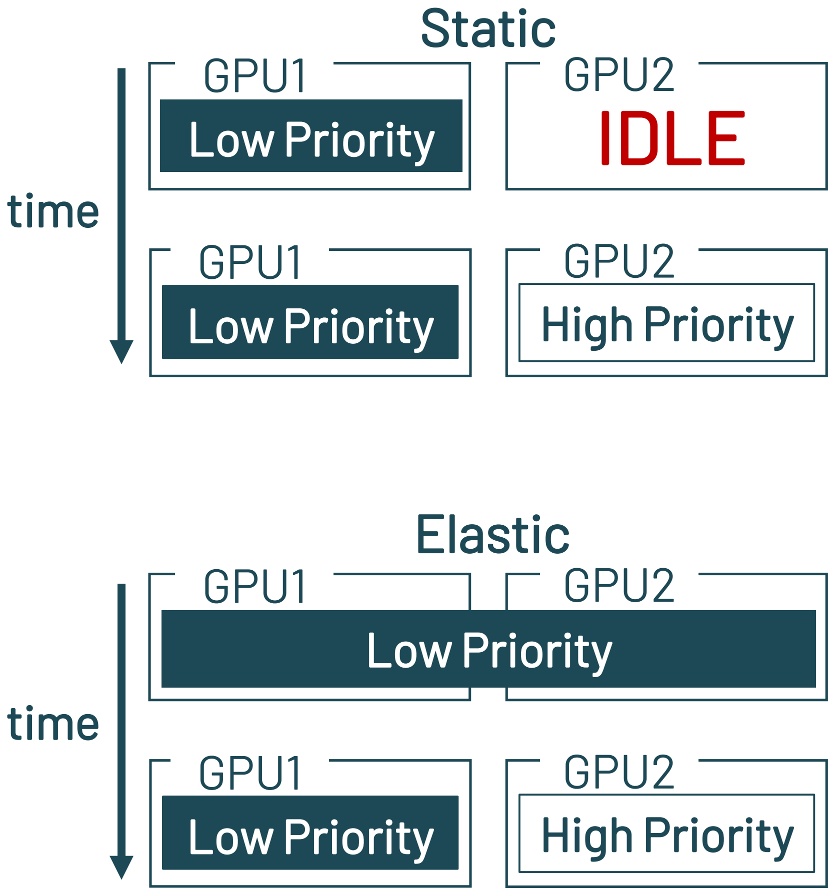




Elastic use of accelerators

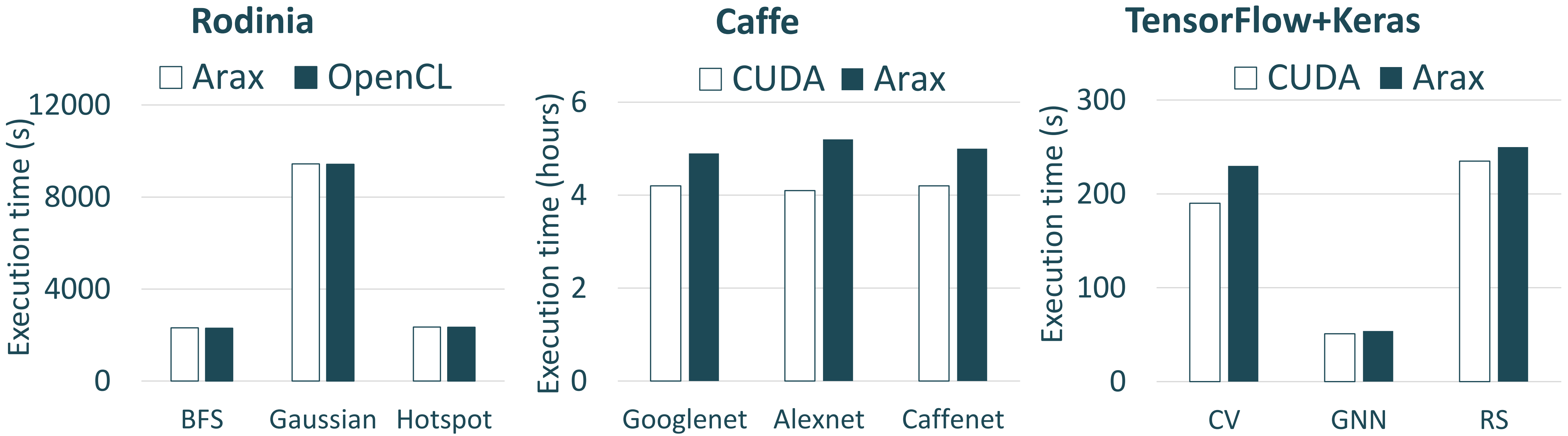
- <u>Dynamically</u> vary the number of accelerators provided to an app
- Low-priority app starts first and then the high-priority
- With elasticity **<u>all accelerators are utilized</u>**
- Small overhead to **<u>high-priority</u>** app





Overhead of Arax compared to native execution

- Arax overhead is mainly due to kernel computation-to-communication ratio
  - <u>High</u>: up to 5% (BFS, Gaussian, Hotspot, LavaMD, etc.)
  - Low: up to 70% (NW, pathfinder)
- For <u>real-world</u> apps (Caffe, TensorFlow) the overhead is 5-28%



Summary

- Arax is a runtime that decouples applications from accelerators using
  - Dynamic task assignment
  - Lazy data placement
  - Spatial sharing
  - Automatic stub generation

- We demonstrate Arax capabilities using
  - Real-world applications: Caffe, TensorFlow, and microbenchmarks: Rodinia
  - Multiple and heterogeneous accelerators: CPUs, GPUs, FPGAs

Arax: A runtime for decoupling apps from accelerators

Open-source: https://github.com/CARV-ICS-FORTH/arax

Questions?

Manos Pavlidakis
manospavl@ics.forth.gr

We thankfully acknowledge the support of the European Commission projects: HiPEAC (GA No 871174), EUPILOT (GA No 101034126) and DEEP-SEA (GA No 955606)