Intro to GPU programming and GPU virtualization (CS695)

- Pramod under the guidance of Prof. Puru

Outline

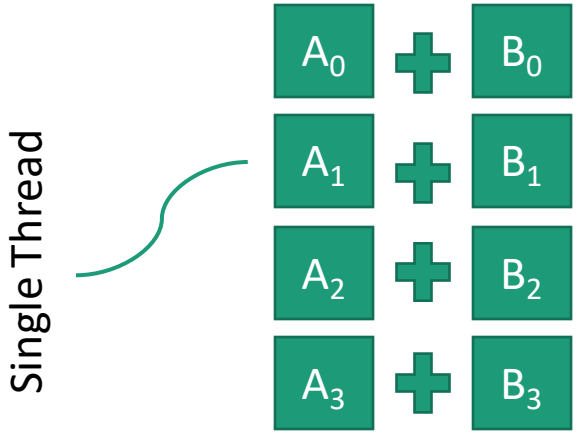
- Background
- CUDA Programming Framework
- Why virtualize GPUs?
- Mechanisms to virtualize GPUs
- Existing problem formulations and setups

Background

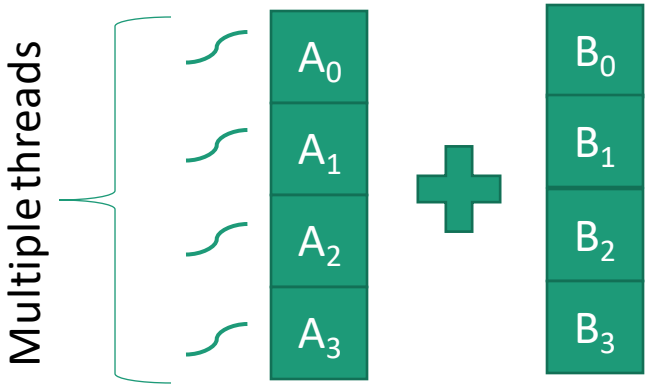
- SIMD and it's applications
- CUDA Programming Framework

Single Instruction Multiple Data (SIMD)

SIMD architecture is primarily used for vector processing and matrix operations.



Scalar Operations



SIMD Operations

1. Elements of a vector are processed one at a time in a loop.
2. One thread performs the same operation on multiple data

1. Multiple elements of a vector are processed at one shot.
2. Multiple threads performs the same operation on multiple data simultaneously.

SIMD Applications

AI/ML

1. Involves considerable matrix computations
2. Same instruction operating on multiple data

Image processing/3-D Rendering

1. Involves RGB vector computations.
2. Same instruction operating on multiple points/pixels of the image.

Cryptography

1. Involves bitwise operations on vectors.
2. SIMD paradigm accelerates vector processing.

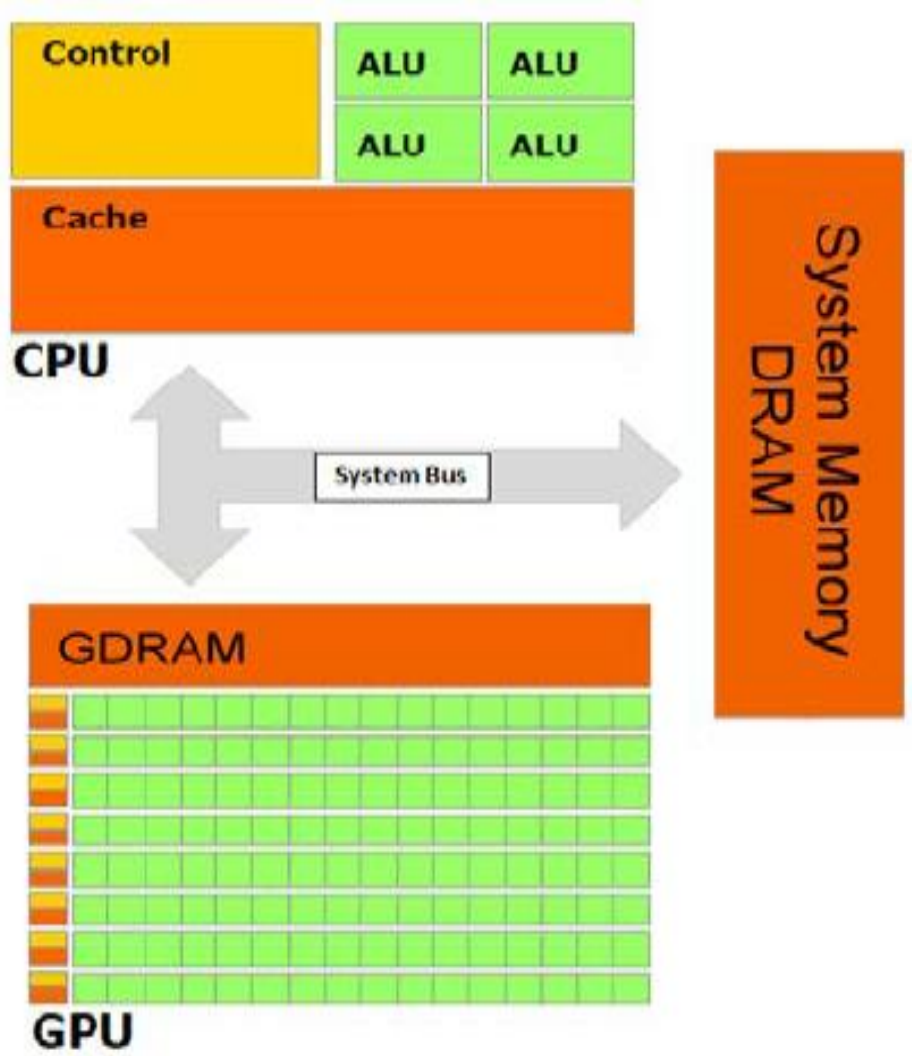
Audio processing

1. Arithmetic operations on audio frames, which are stored in vectors.
2. Same instruction on multiple audio frames can be applied.

High level GPU architecture

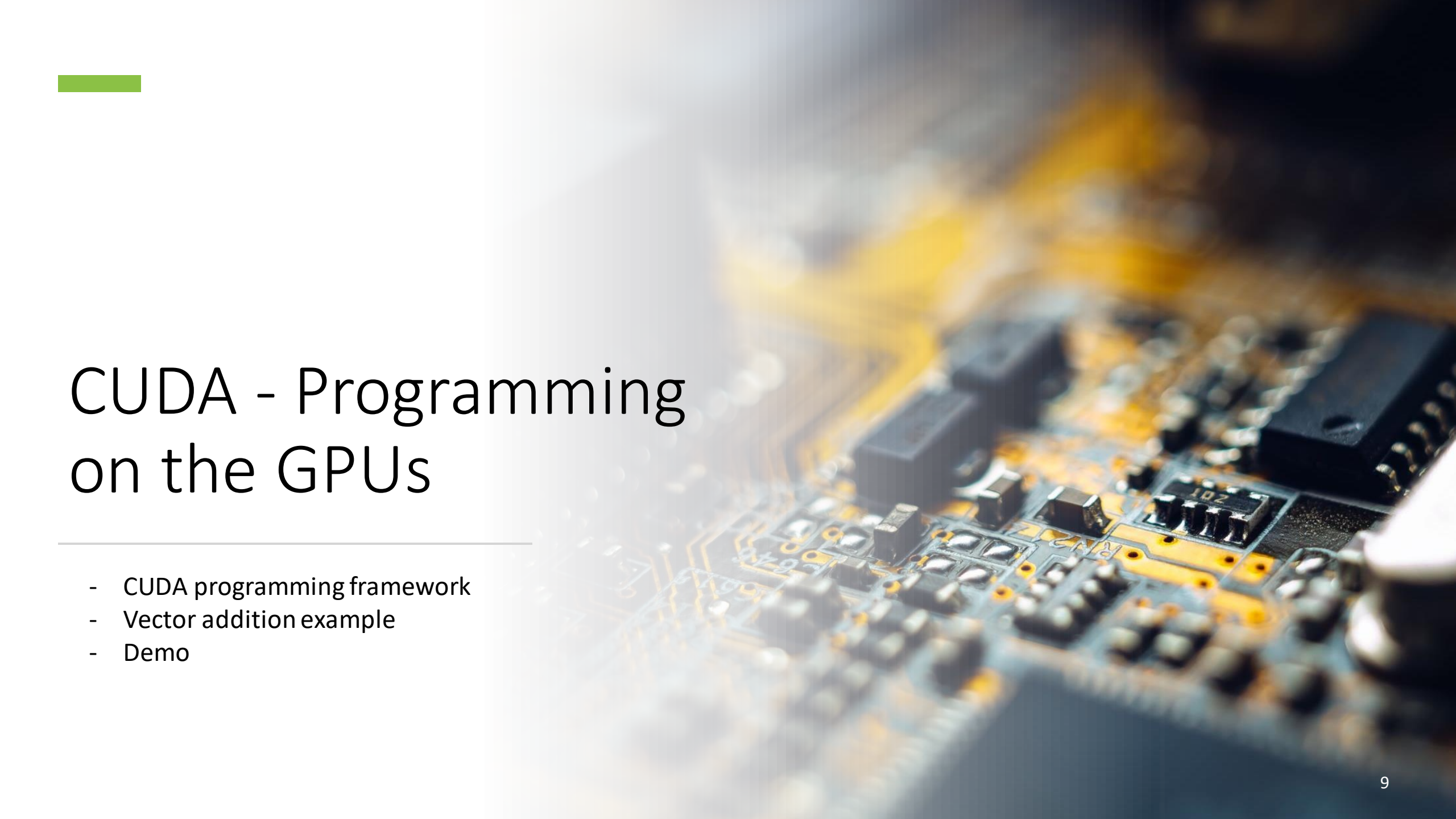
- Thousands of small cores
  - Streaming multiprocessors (SMs) contain the cores
- SM – Group of cores with it's own cache
- Own device memory
- Mounted on a PCIe bus

- CUDA/OpenCL API for interfacing

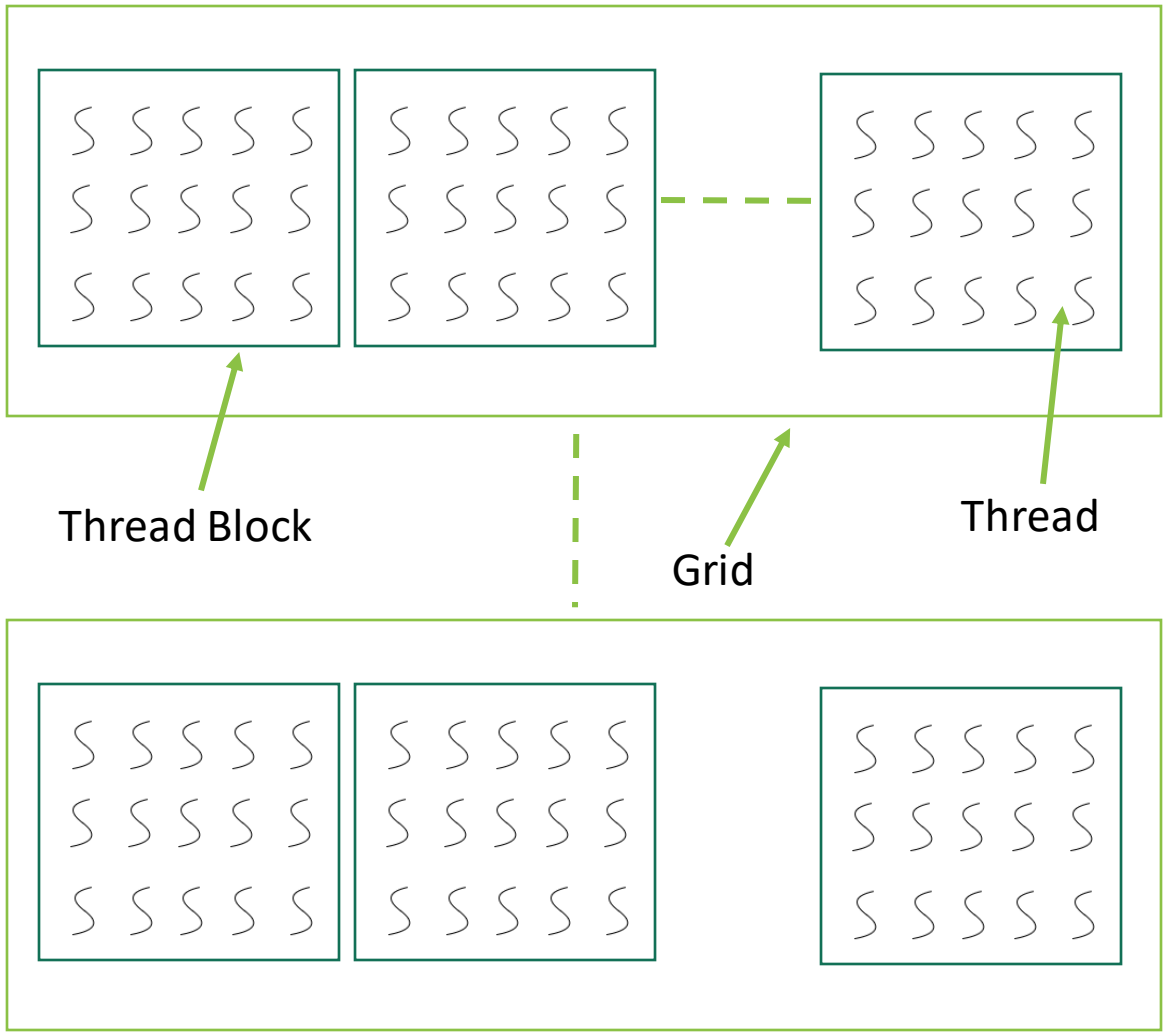


CUDA - Programming on the GPUs

- CUDA programming framework
- Vector addition example
- Demo



CUDA Programming Framework



1. Each thread runs on one GPU core
2. Group of threads form a thread block.
3. All the threads in a block run on one Streaming multiprocessor.
4. Group of thread Blocks is a Grid
5. Grids are run parallelly across many Streaming multiprocessors.
6. Each Streaming multiprocessor in a GPU has it's own cache.

CUDA vector addition – Single Block, Single thread

Kernel: On GPU

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i ++){
        out[i] = a[i] + b[i];
    }
}
```

Kernel setup: On CPU

Allocate GPU memory
transfer data,
Initiate GPU kernel

```
// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);

// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);

// Executing kernel
vector_add<<<1, 1>>>(d_out, d_a, d_b, N);
```

CUDA vector addition – Single Block, Multiple threads

Kernel: On GPU

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
    }
}
```

Kernel setup: On CPU

Allocate GPU memory
transfer data,
Initiate GPU kernel

```
// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);

// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);

// Executing kernel
vector_add<<<1,1024>>>(d_out, d_a, d_b, N);

// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

CUDA vector addition – Multiple Blocks and threads

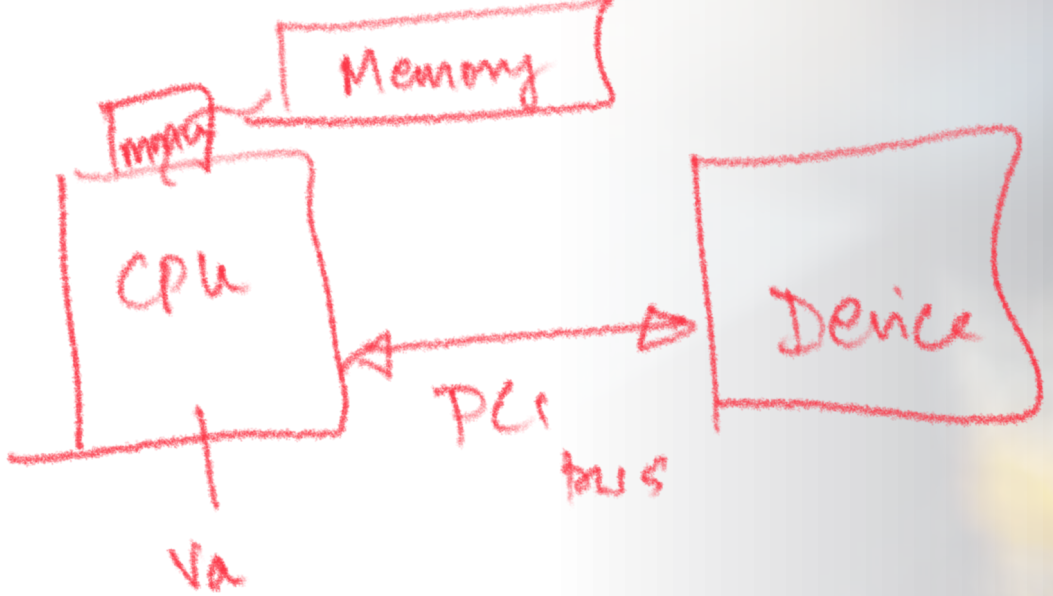
Kernel: On GPU

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Handling arbitrary vector size
    if (tid < n){
        out[tid] = a[tid] + b[tid];
    }
}
```

Kernel setup: On CPU

Allocate GPU memory
transfer data,
Initiate GPU kernel

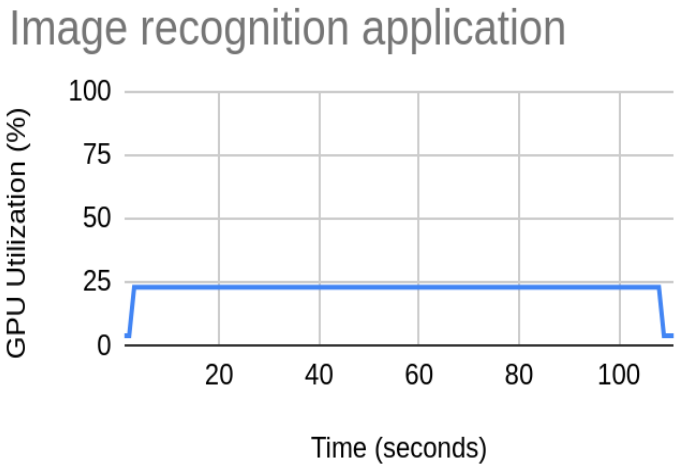
```
// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);

// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);


// Executing kernel
int block_size = 1024;
int grid_size = ((N + block_size) / block_size);
vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);

// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```



Virtualization aspect of GPUs

- Why virtualize GPUs
- Mechanisms to virtualize GPUs
- Challenges

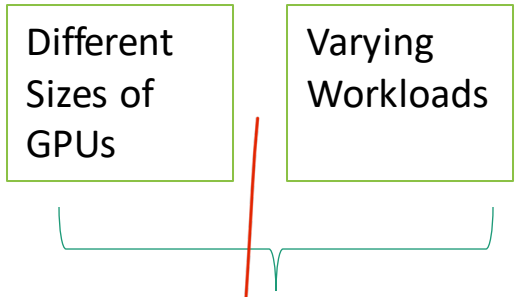
Why virtualize GPUs?



GPU utilization with one application

1. Single user/application not enough to fully utilize a GPU
2. This results in under-utilization of a GPU

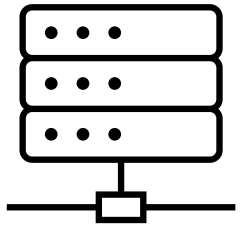
MAXIMIZE UTILIZATION



Different requirements

1. Multiple requests/VMs can have different GPU sizes
2. Varying workloads such as compute/graphic can be optimized.

FLEXIBILITY

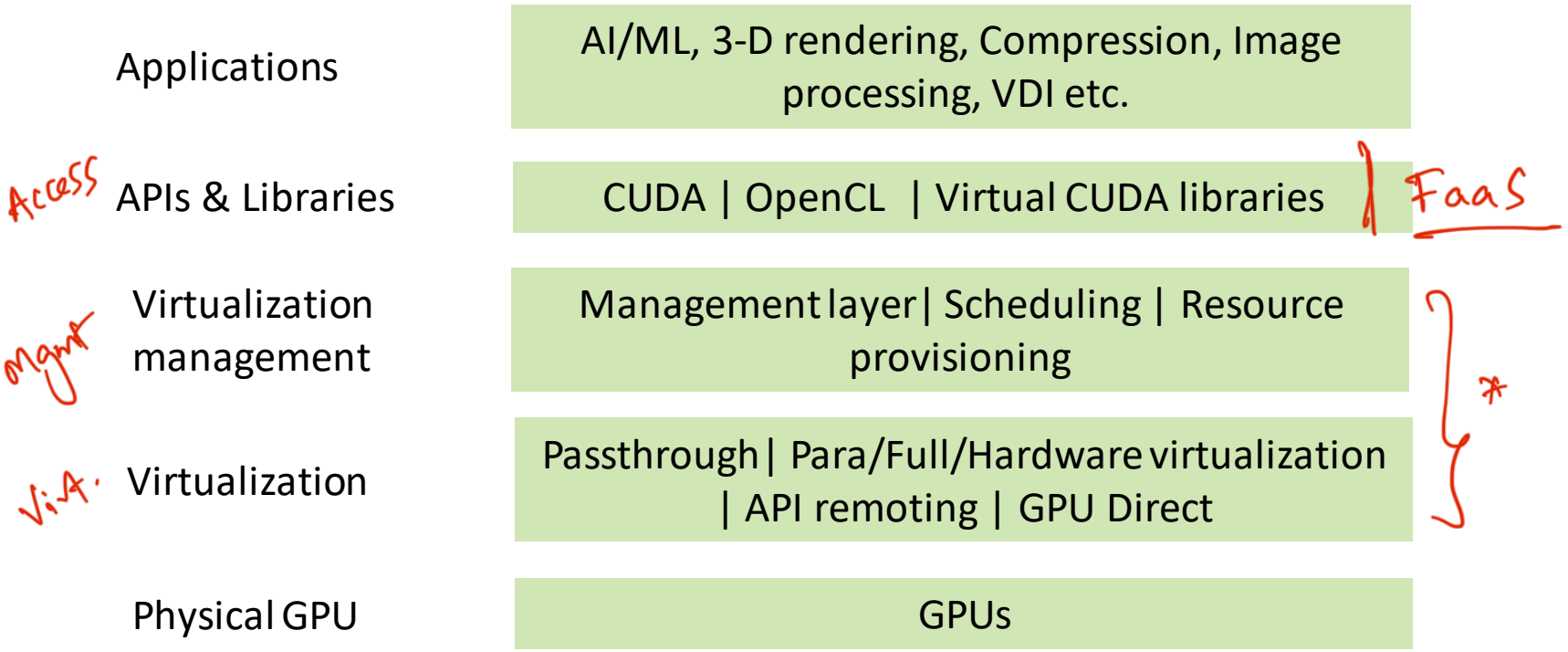


First class resource in most data centres

1. GPUs are nowadays found in almost all data centres.
2. Due to high cost of GPUs, we want to maximise their usage.

FIRST CLASS RESOURCE

GPU virtualization software stack



to provide efficient GPU accesses locally and on the network.

Virtualization options for the GPU. ⓐ NVIDIA GPUs ⟦AMD & Intel.

① MiG — multi-instance GPUs — device abstraction

② vGPU — h/w assisted virtual GPUs — h/w — spatial multiplexing of cores

temporal multiplexing of cores.

③ MPS — multi-process service ‖ s/w + GPU driver assisted.

④ API virtualization

⑤ Kernel slicing

} s/w

⑥ FaaS

mig profiles

vGPU profiles

#vGPUs to create

mem

cores

mem

Mechanisms to virtualize GPUs

	Hardware assisted	Software assisted	Software + Developer assisted
Examples	VGPUs, MiGs	RCUDA, GPU manager, vCUDA, MPS etc.	Kernel Slicing
Multiplexing	Temporal(vGPUs) and Spatial (MiGs)	Temporal and Spatial (MPS)	Temporal
Control Plane	Native Drivers	Virtualization layer	APIs
App changes required	No	No	Yes
Abstraction	Device	Device, APIs	APIs

1. Virtualization mechanisms at different abstraction levels can be used together
2. The mentioned examples are the solutions that use these mechanisms to virtualize GPUs at different abstraction levels.

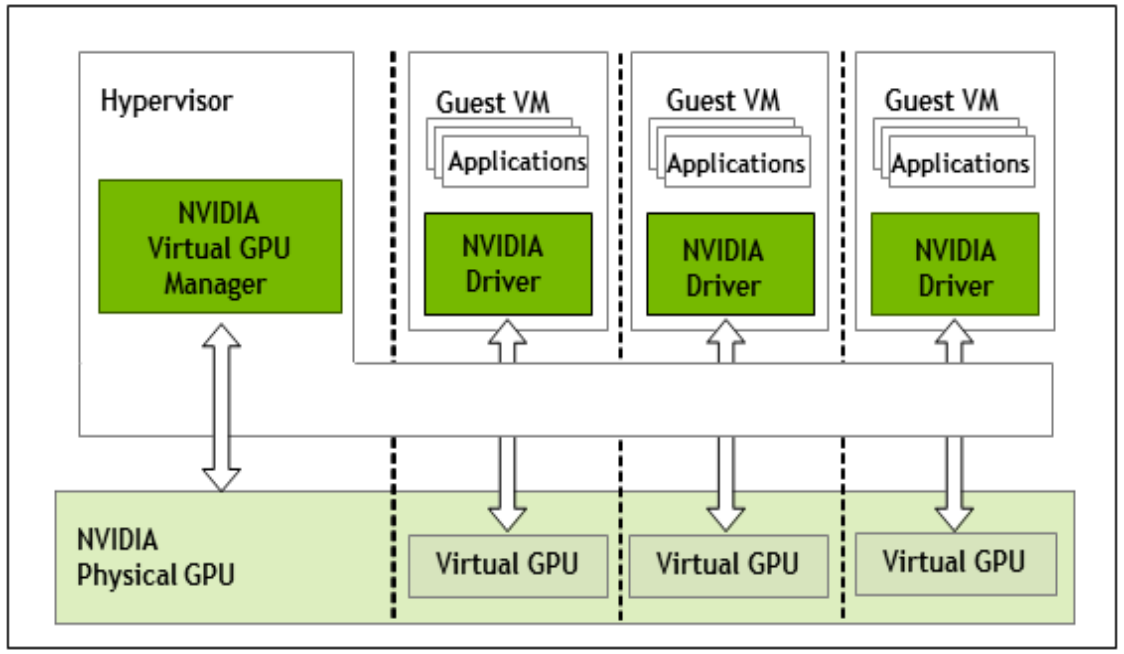
vGPU Profiles

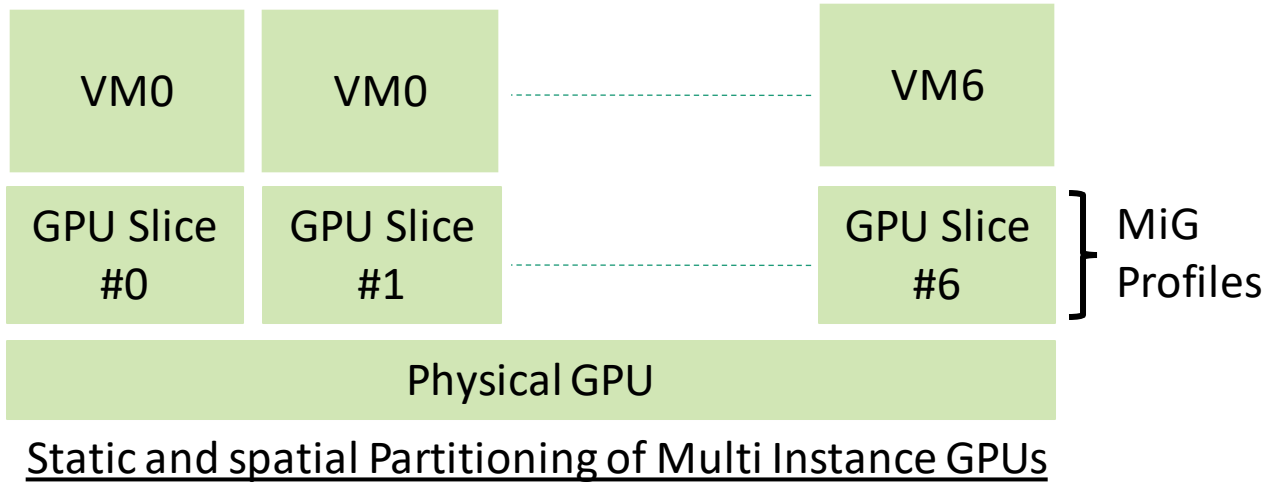
GPU: NVIDIA RTX A5000 (24GB memory), 8192 cores

Profile	Num of vGPUs	Memory per vGPU
1q	24	1GB
2q	12	2GB
4q	6	4GB
8q	3	8GB
24q	1	24GB
4C	6	4GB
6C	4	6GB
8C	3	8GB
24C	1	24GB

vGPU profile is configured by user via the (host) driver
Cannot have a mix of profiles on the host for the same GPU

vGPUs & MiGs





Static and spatial Partitioning of Multi Instance GPUs

vGPUs:

1. Each virtual GPU is assigned to a VM.
2. The host driver contains the scheduler through which work is assigned on a physical GPU.
3. vGPUs are temporally share the GPU

RR, Best effort, Fixed share

MiGs

1. Each partition of a MiG enabled GPU is attached to a VM
2. Spatially multiplexing of cores and memory.
3. Max of 7 slices of the GPU can be created.

MiG Profiles for A100

Config	GPC Slice #0	GPC Slice #1	GPC Slice #2	GPC Slice #3	GPC Slice #4	GPC Slice #5	GPC Slice #6
1	7						
2	4				3		
3	4				2		1
4	4				1	1	1
5	3			3			
6	3			2		1	
7	3			1	1	1	
8	2		2		3		
9	2		1	1	3		
10	1	1	2		3		
11	1	1	1	1	3		
12	2		2		2		1
13	2		1	1	2		1
14	1	1	2		2		1
15	2		1	1	1	1	1
16	1	1	2		1	1	1
17	1	1	1	1	2		1
18	1	1	1	1	1	2	
19	1	1	1	1	1	1	1

Combinations of different possible combinations

1. The numbers indicate the fraction of how the GPU can be split

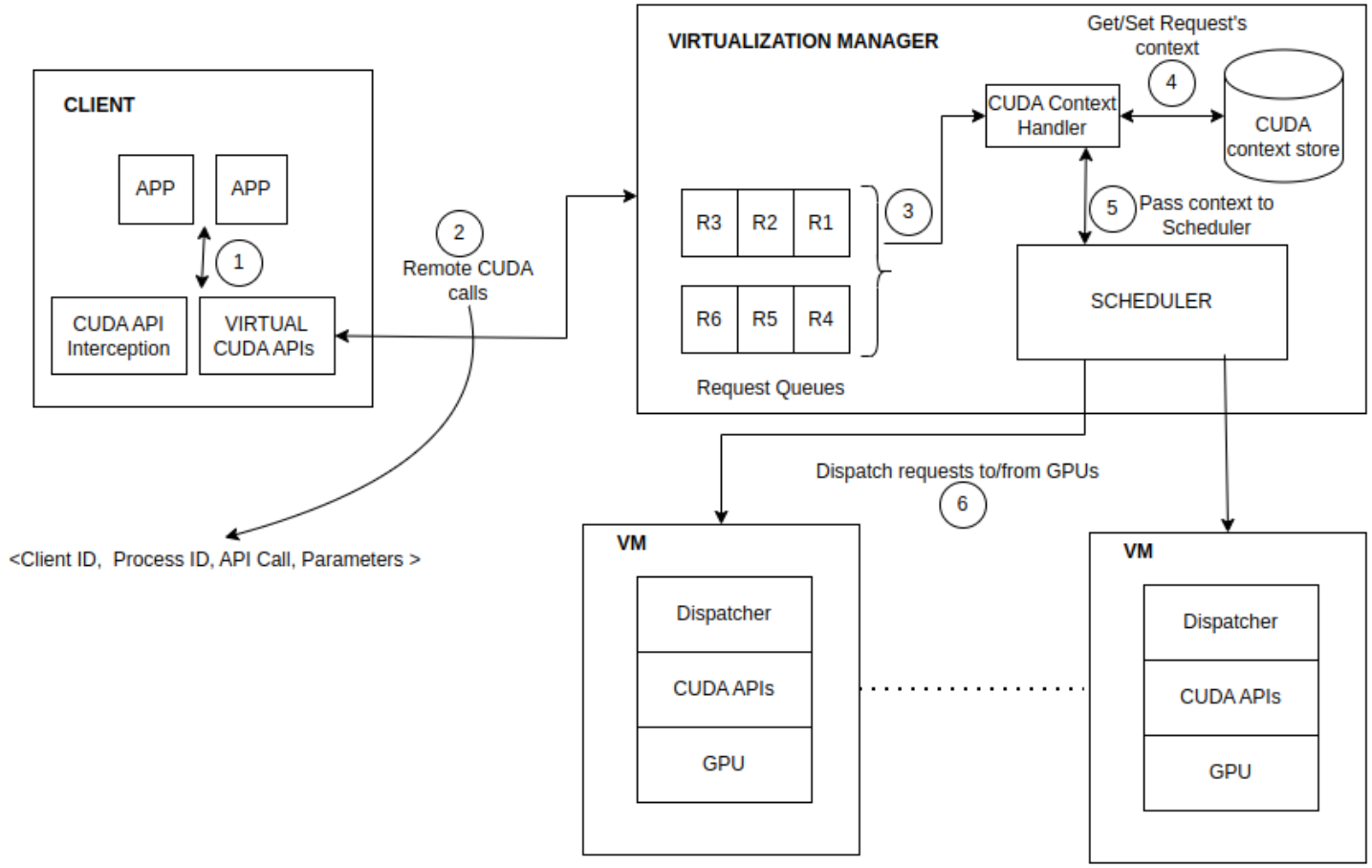
2. For example, for config no. 2
   1. First slice will have 4/7 of the GPUs resources
   2. Second slice will have 3/7 of the GPUs resources

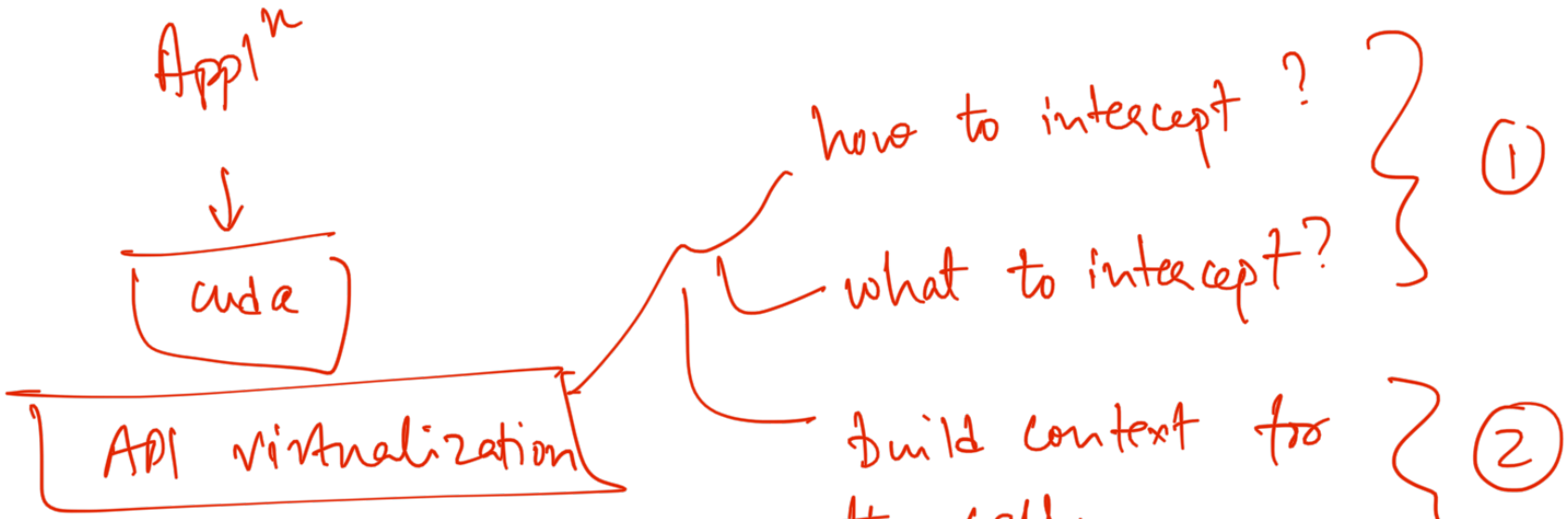
Challenges with vGPUs and MiGs

- Reserve-and-use service model limits resource usage efficiency
- Static partitioning of GPUs
- Tight-coupling with control plane for access and management

API Virtualization of GPUs

- Client CUDA calls are virtualized
- Multiplexing and resource management happens with the virtualization manager
- Client need not worry about management of GPUs.
- CUDA calls are either intercepted, or a virtualized CUDA library is provided.





how to intercept ?

what to intercept?

} (1)

build context for the call.

} (2)

↳ client id, func id, parameters, data, ...

interpret & setup local context } (3)

make the API call } (4)

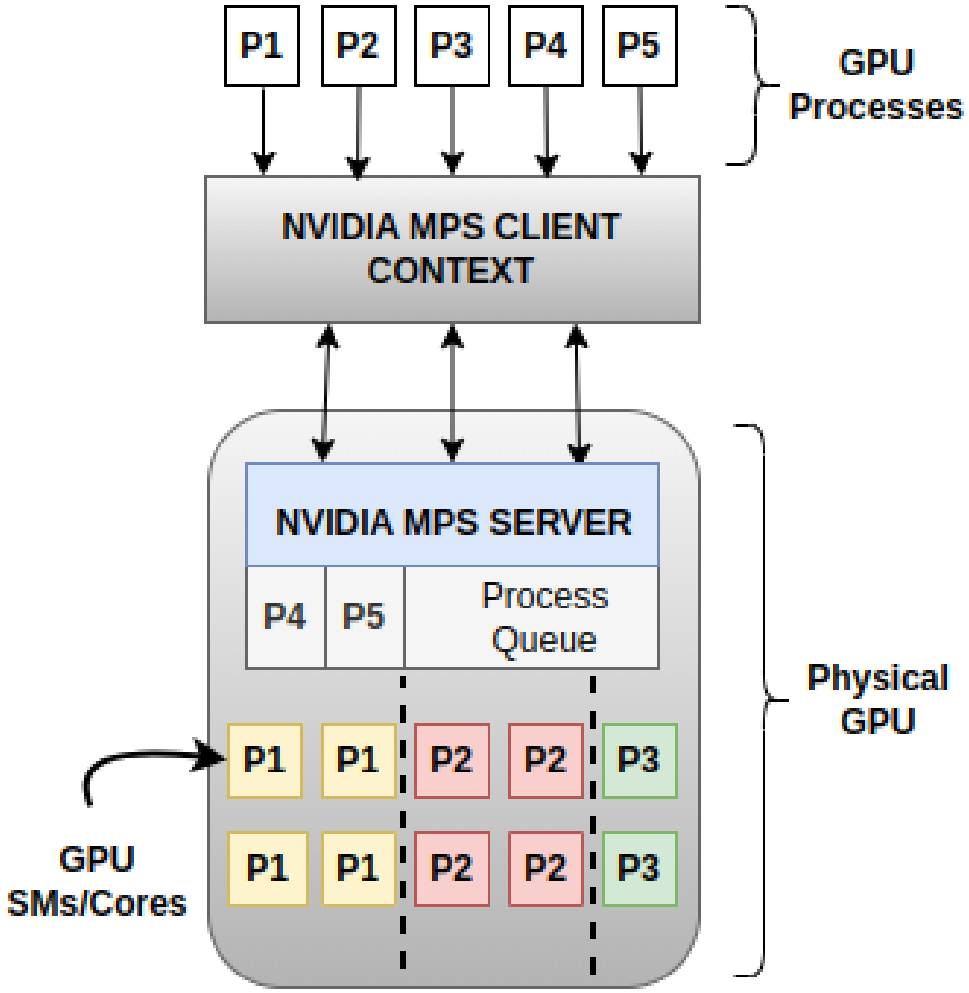
Store results & context locally return results } (5)

solns.

- vCuda
- vCuda
- mCuda
- Bitfusion
- Cricket
- qCuda

Multi Process Service (MPS)

- MPS supports limited execution resource provisioning for Quality of Service (QoS)
- Allows a max of 48 GPU processes to execute simultaneously.
- Allows for different resource limits to be set between processes running on the GPU.



Kernel Slicing

K1 ~ 1M threads

K2 ~ 1M threads

run to completion

⇒ does not allow preemption/multiplexing.

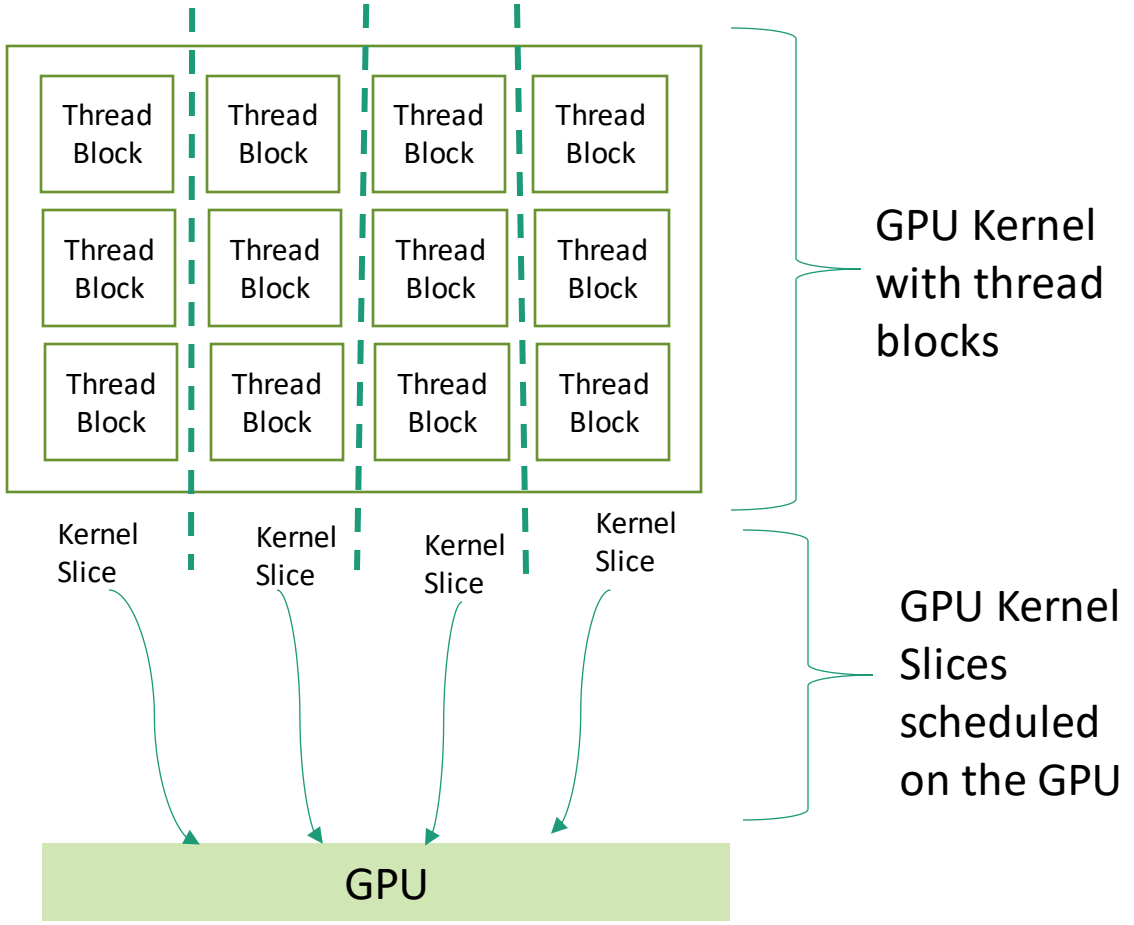
1. A GPU kernel is made up of thread blocks (Shown previously in vector addition example)

2. Kernel slicing is a mechanism, by which, instead of invoking all the thread blocks at once, we can invoke parts of them.

3. This creates more scheduling opportunities for other Kernel slices to be executed on a GPU as there is no pre-emption on GPUs

4. Kernel slicing perform psuedo pre-emption of a CUDA program

5. The CUDA program is responsible for managing/deploying the kernel slices



K1

issue 1M threads

K2:

issue 1M threads.

K1.

{

issue SLICE #threads

}

do this till 1M threads done.

K2:

{

issue SLICE #threads.

}

wait for slice to complete

repeat till 1M threads done.

Challenges with API virtualization

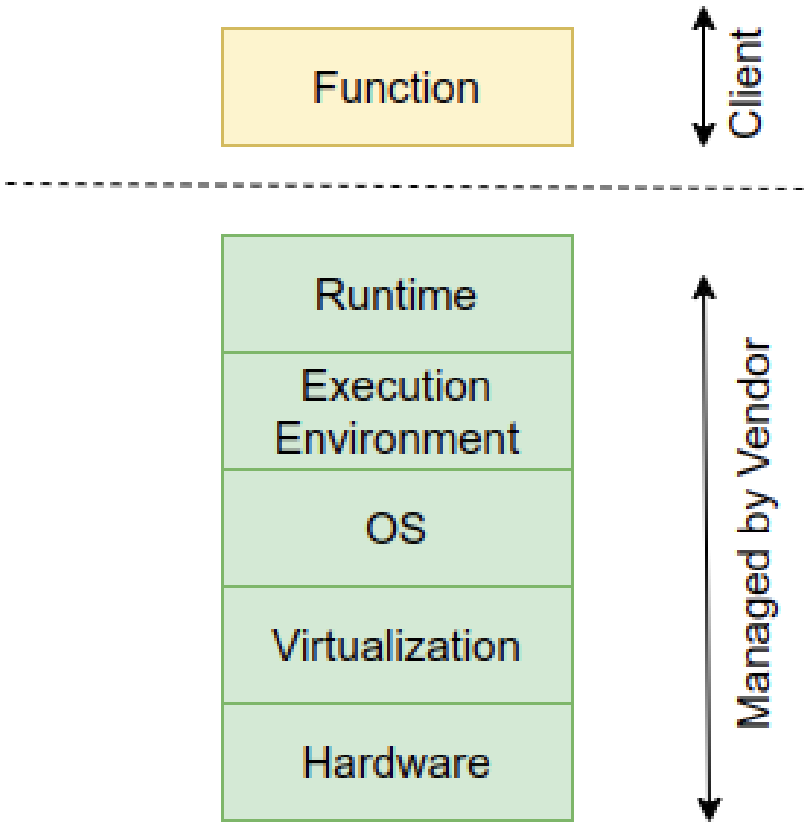
- Performance overhead added when a software virtualization layer is introduced.
- Maintaining context/state of every process being virtualized.
- Providing isolation between the workloads

Use cases for GPU virtualization

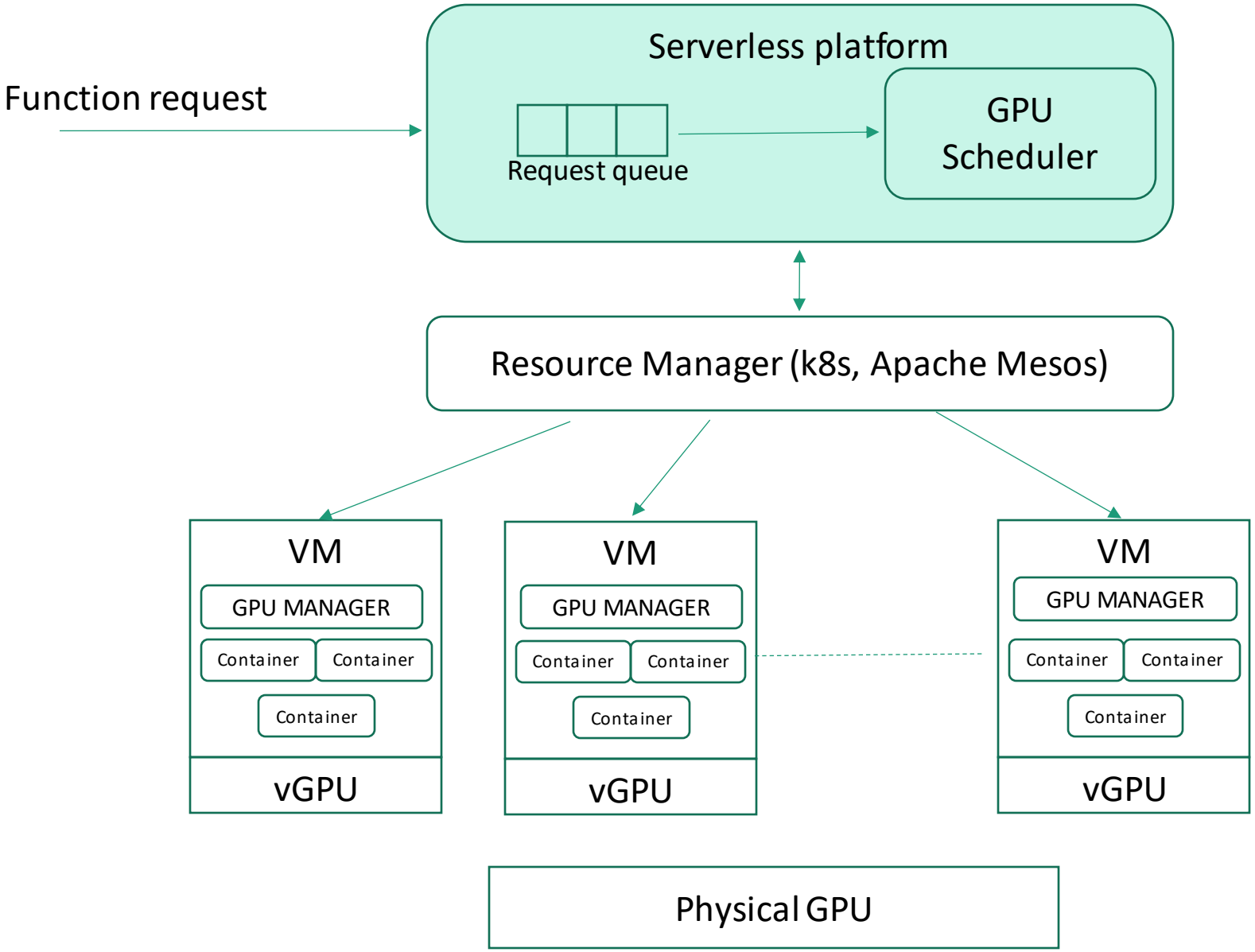
- Use of GPUs in serverless platforms
  - (Serverless platforms are also a mechanism to virtualize GPUs at a function abstraction level)
- High performance computing
- Virtual Desktop Infrastructure

What is FaaS?

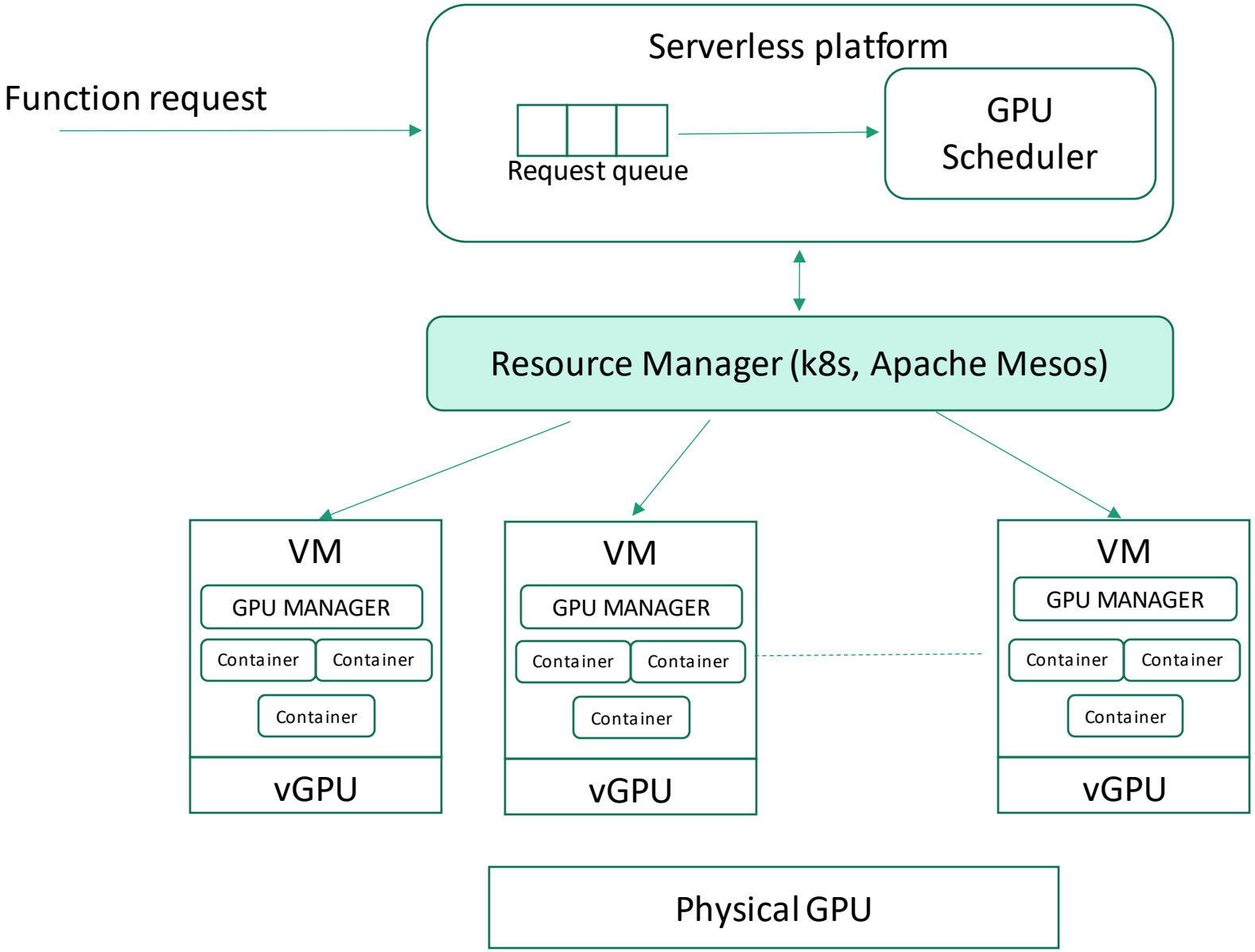
- Developers focus only on business logic, rest is managed by the vendor.
- Scale up or down automatically



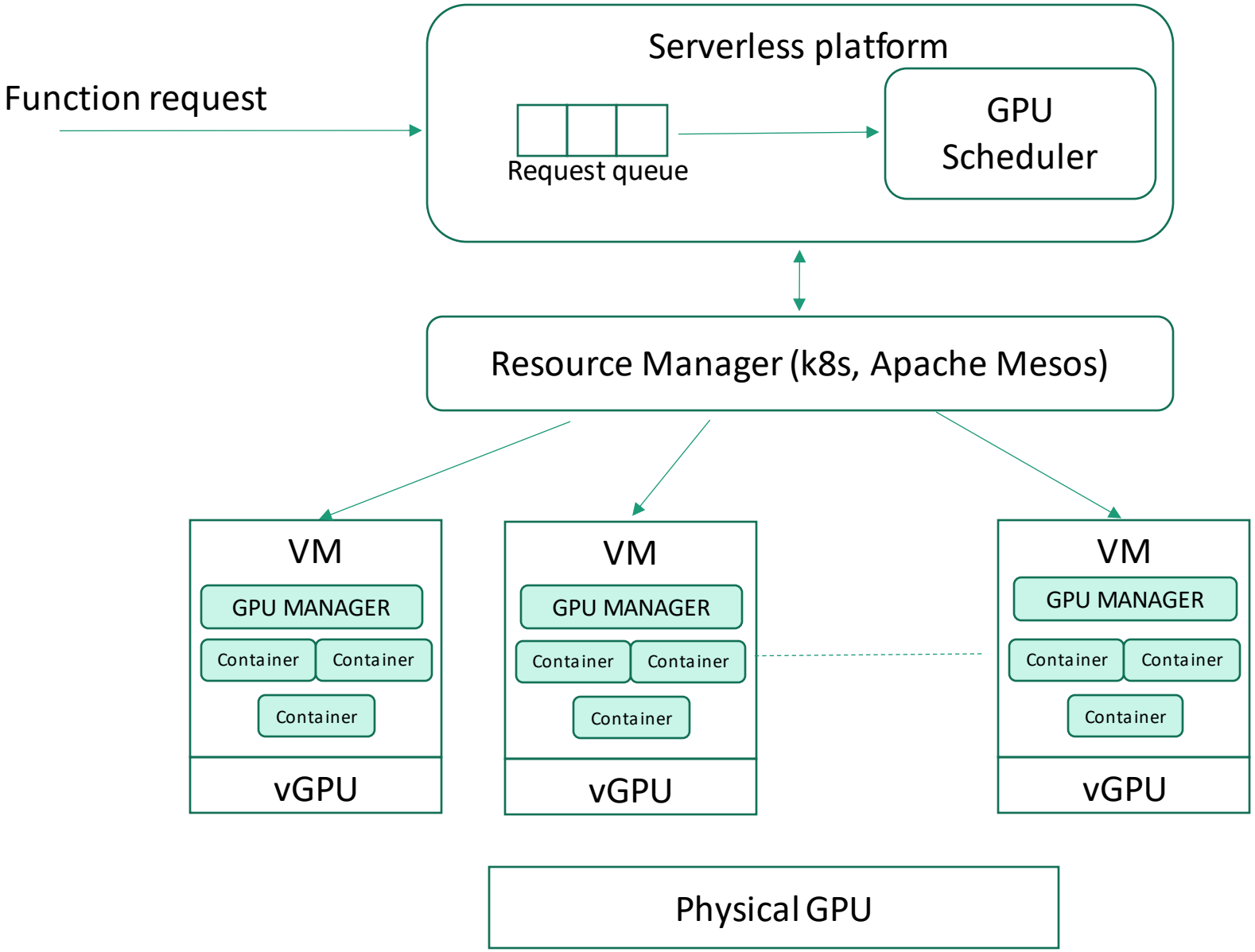
FaaS platform with GPUs



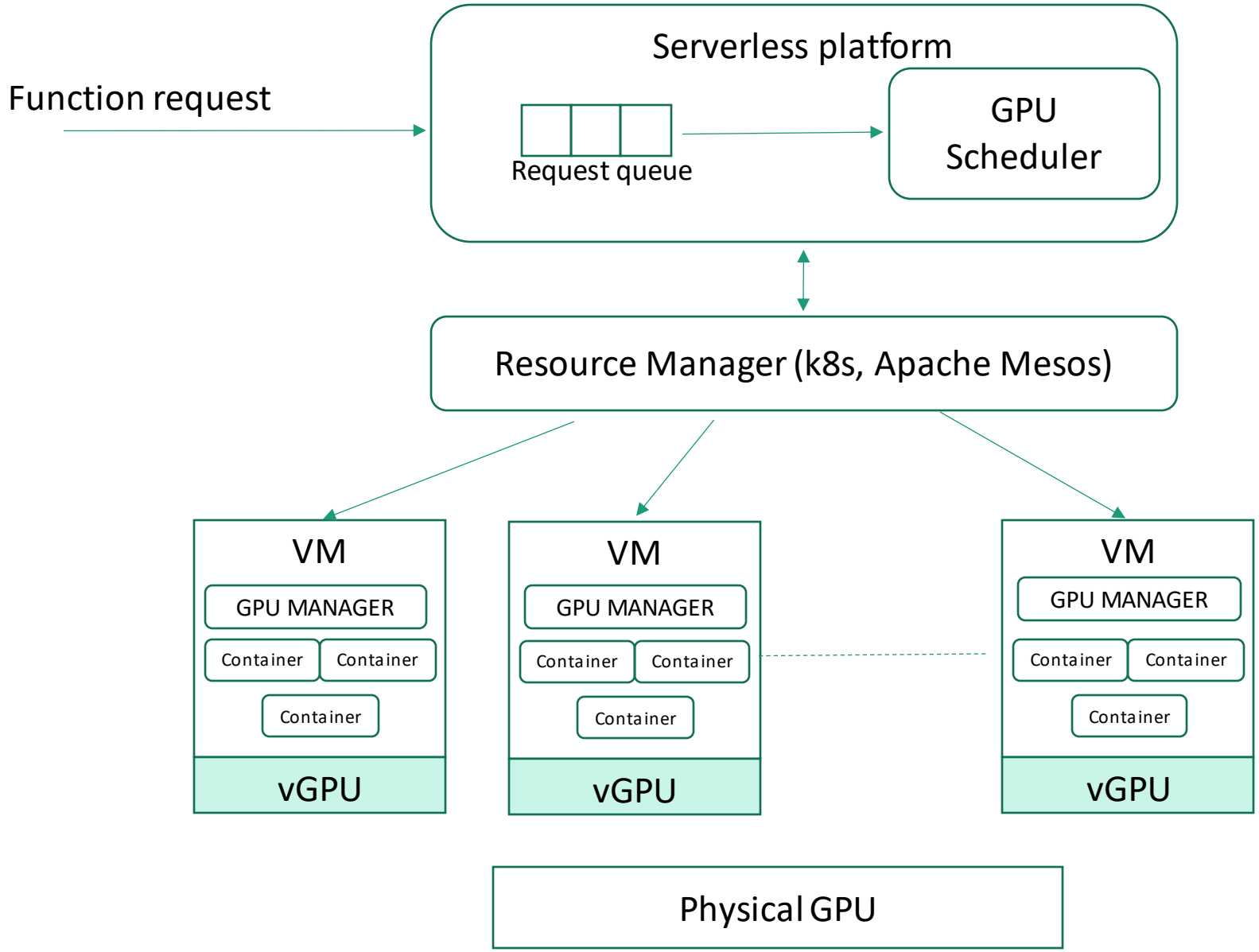
FaaS platform with GPUs



FaaS platform with GPUs

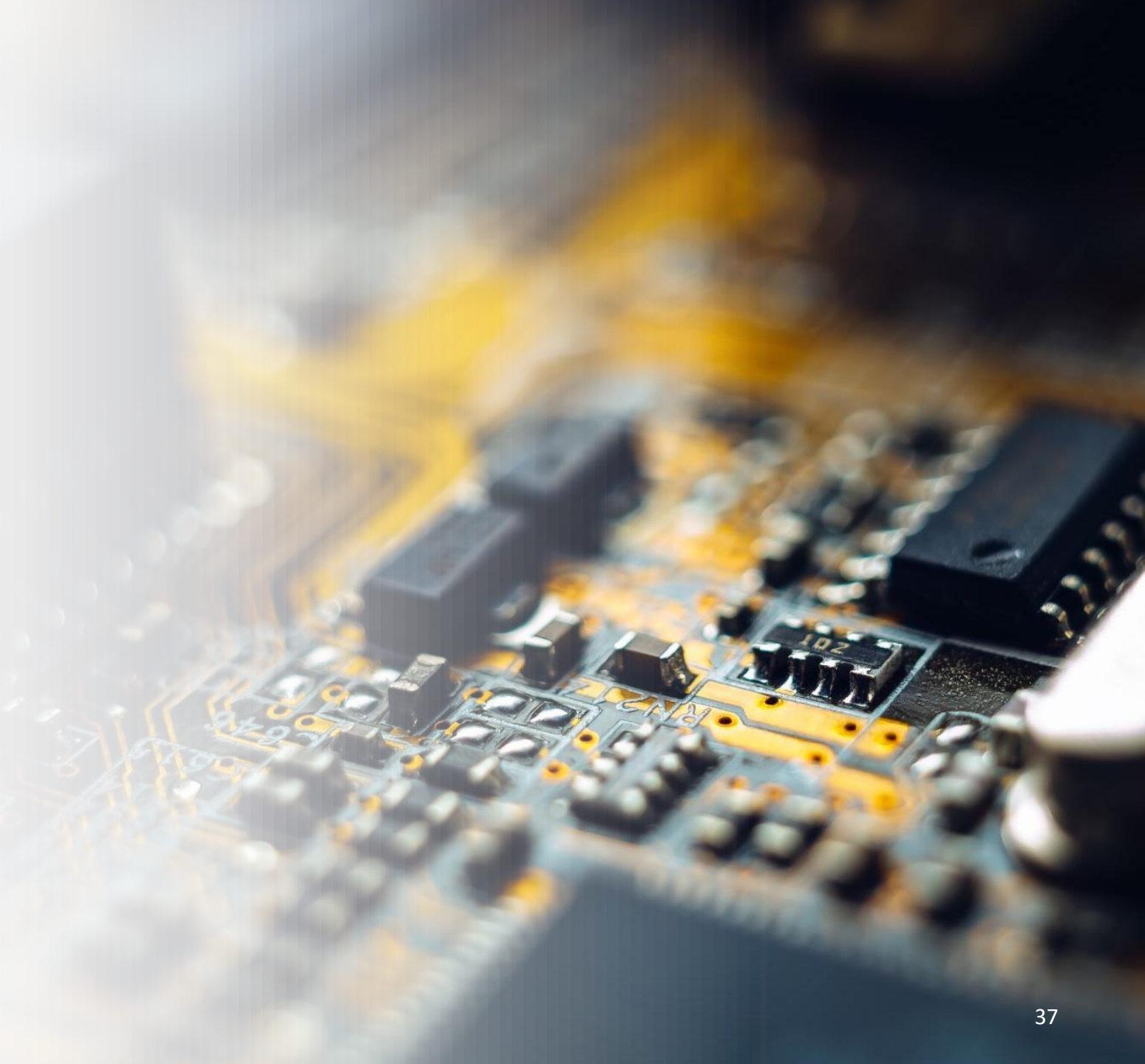


FaaS platform with GPUs

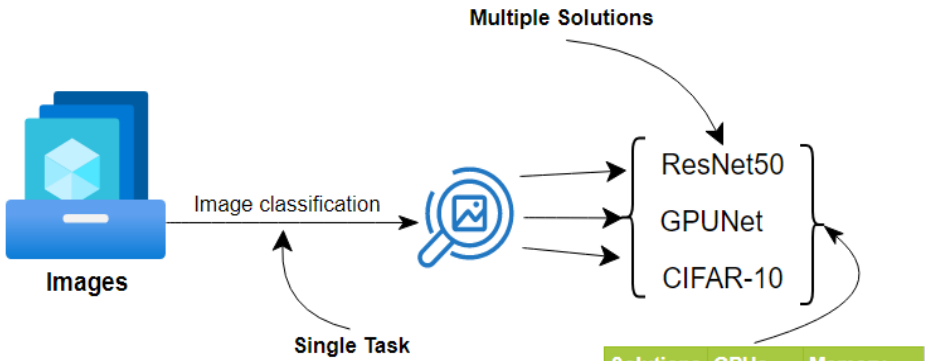


Existing Ongoing Projects

- Variants based scheduling of GPU functions
- Characterization and profiling for NVIDIA MPS
- MPS plugin for Kubernetes
- Orchestration of GPU kernel workflows
- Eureka: Share based vGPU Task Scheduling



Variants based scheduling of GPU functions



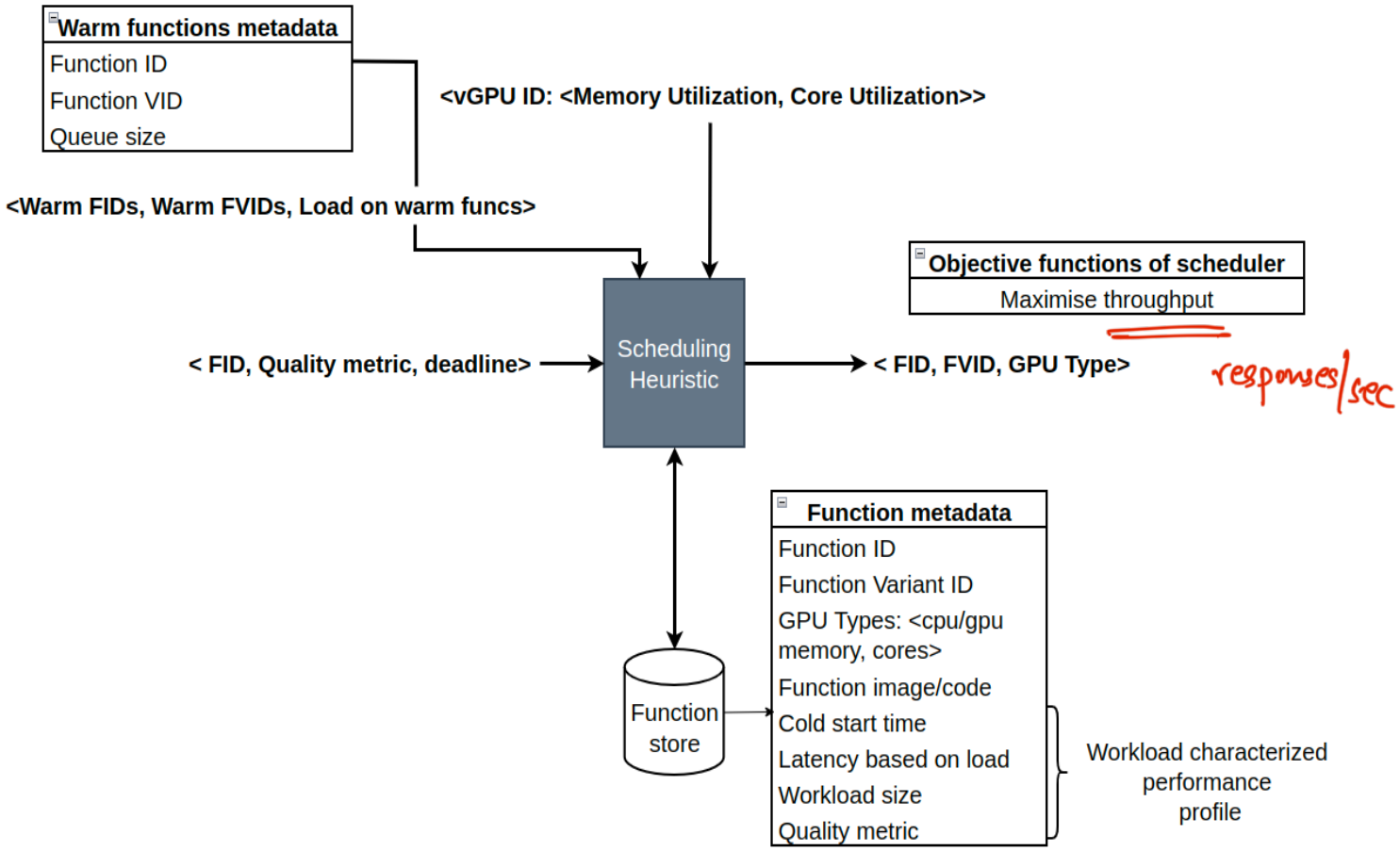
Solutions	GPUs	Memory
ResNet50	Pascal >	7GB - 16GB
GPUNet	Volta >	~14GB
CIFAR-10	Pascal >	1GB >

Different GPU requirements[1]

performance vs. cost vs. resources

1. Each functionality has multiple variations, i.e it's variants

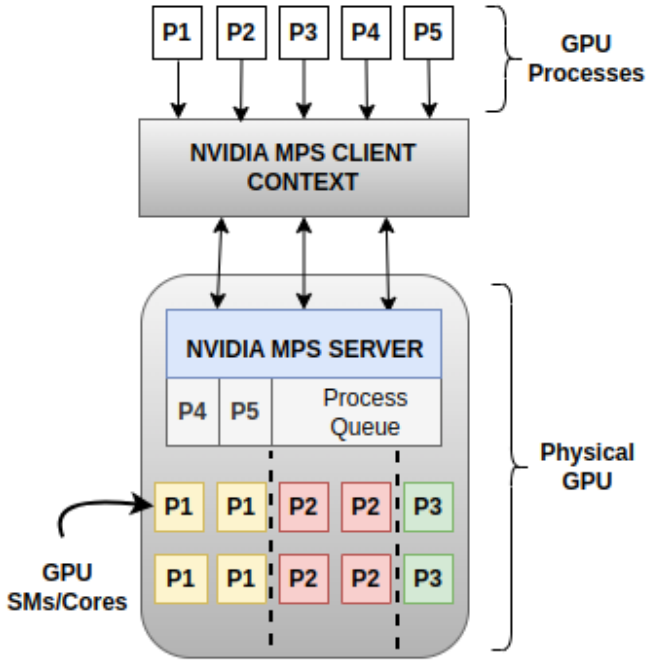
2. Function variants provide a resource v/s performance trade off



Problem overview for designing the FaaS GPU scheduling heuristic

Project by, **Pramod S Rao (**pramodrao@cse.iitb.ac.in**)**

Characterization and profiling for NVIDIA MPS



MPS client/server architecture

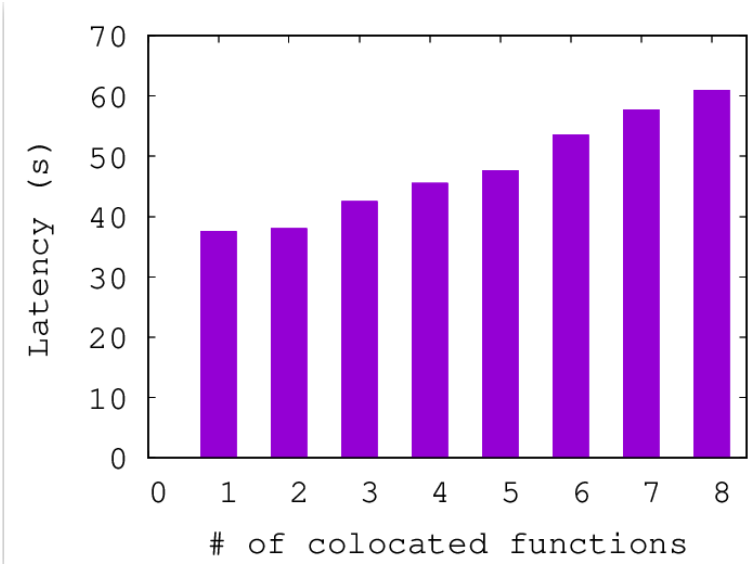
1. Performance of different workloads under different setups:
   I. MPS on bare metal
   II. MPS on vGPUs
   III. MPS on bare metal v/s vGPUs

2. Characterization of GPU workloads

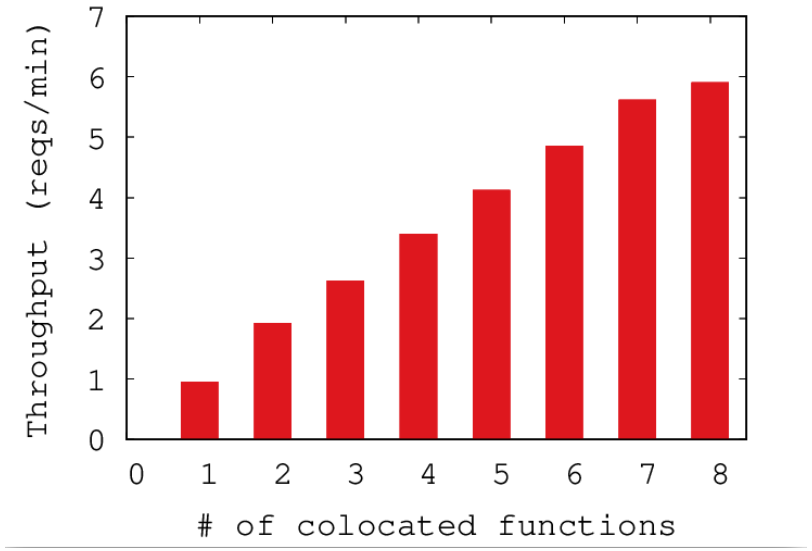
3. Creating a performance model to feed into inputs of GPU schedulers
   I. Observation of interference effects on co-location

Project by, **Mitali Sunil Meratwal (**190070033@iitb.ac.in**)**

Findings on NVIDIA MPS

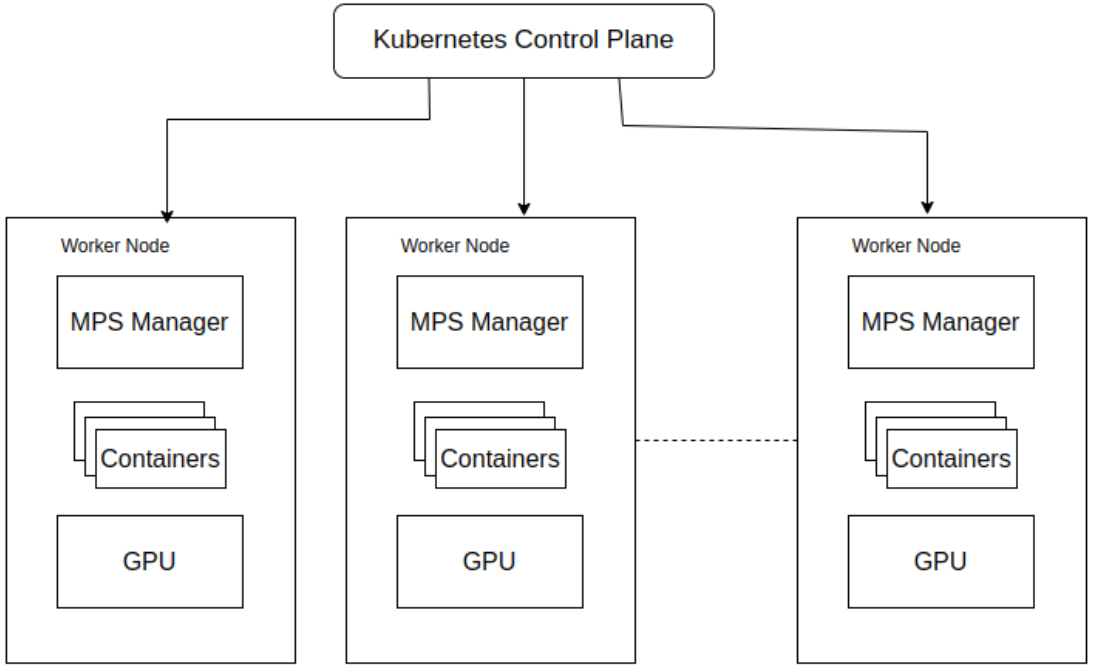


- Increase in latency between co-located functions via MPS due to interference.



- Increase in throughput for co-located functions via MPS.

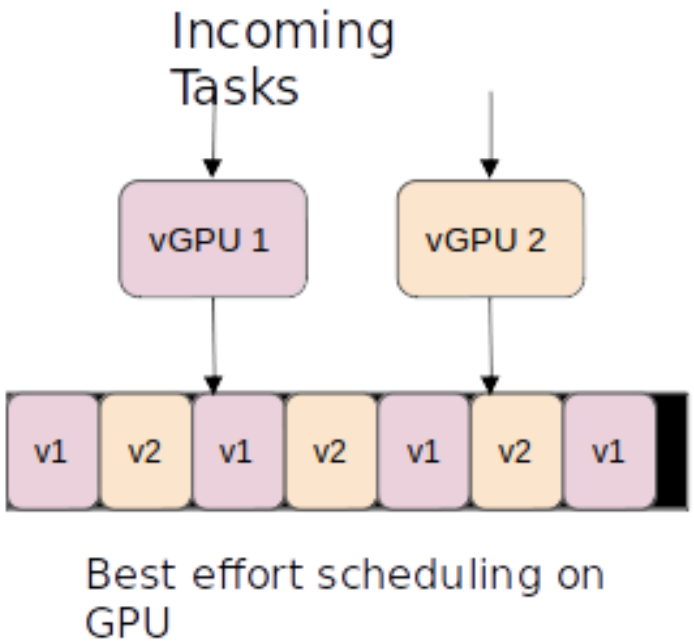
MPS plugin for Kubernetes



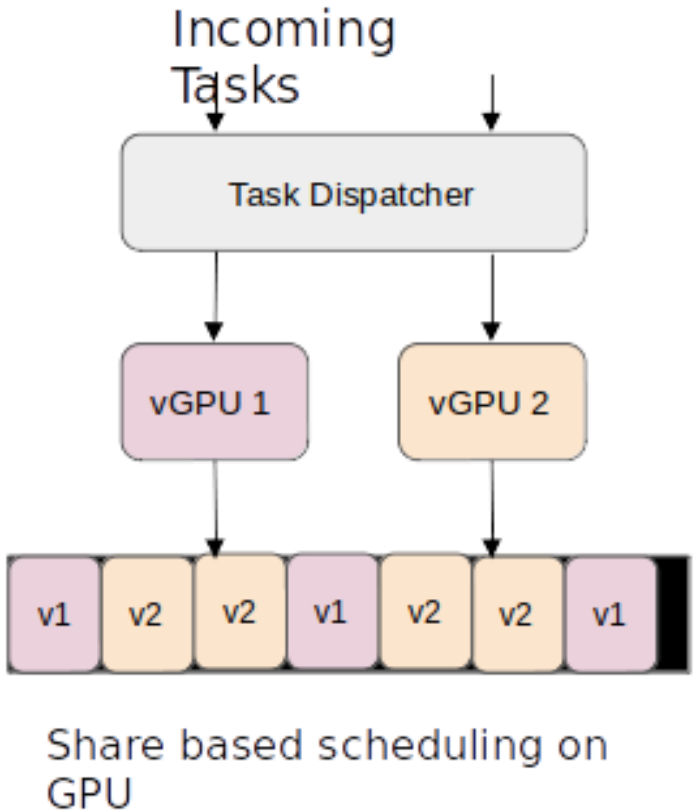
1. Extension of existing GPU manager to incorporate virtualization via NVIDIA MPS
2. Virtualization software which is spatially multiplexing in nature
3. Device plugin for Kubernetes
4. This feature enables dynamic multiplexing features of MPS for orchestrated containers on Kubernetes

Project by, **Rahul Shukla (**rahul4shukla64@gmail.com**)**

Eureka: Share based vGPU Task Scheduling



Best effort scheduling on GPU



Share based scheduling on GPU

- In best-effort scheduling vGPUs are scheduled in a Round Robin manner. All vGPUs are given same priority on the GPU.
- We are designing a task distributor that will schedule tasks on different vGPUs in some ratio.
- Suppose there are two vGPUs v1 and v2, we will be able to schedule tasks such that for every 100ms utilization of GPU by v1, v2 uses GPU for 200ms.
- We can provide QoS based GPU allocation.

Project by, **Sameer Ahmad (**22m0789@iitb.ac.in**)**

Orchestration of GPU kernel workflows

1. GPU memory operations are expensive

2. Same memory is often required by the next executing Kernel

3. This project aims at leaving behind output of the current kernel in the GPU as the input to the next kernel

4. This helps reduce the transfers between CPU and GPU memory

Project by, **Bhattad Krushnakant Dilip (**190100036@iitb.ac.in**) and Anshul Sanghi (**22m0758@iitb.ac.in**)**

Q & A