Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences

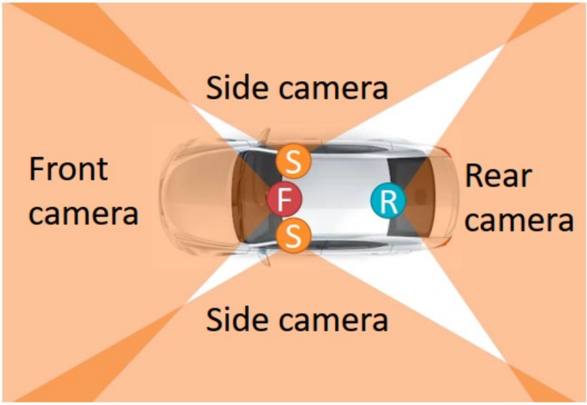
OSDI'22

Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen

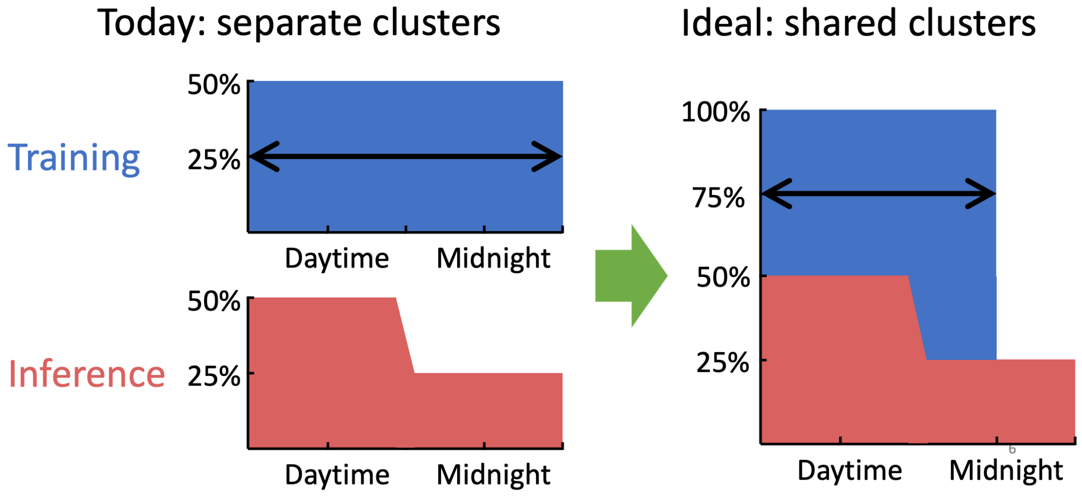
Shanghai Jiao Tong University, Shanghai AI Laboratory

Introduction: Multi-DNN Inference Serving System

- Multi-DNN with Multi-Input (RTSS'19 DART)



- Executing Inference and Training Tasks Concurrently (OSDI'20 PipeSwitch)



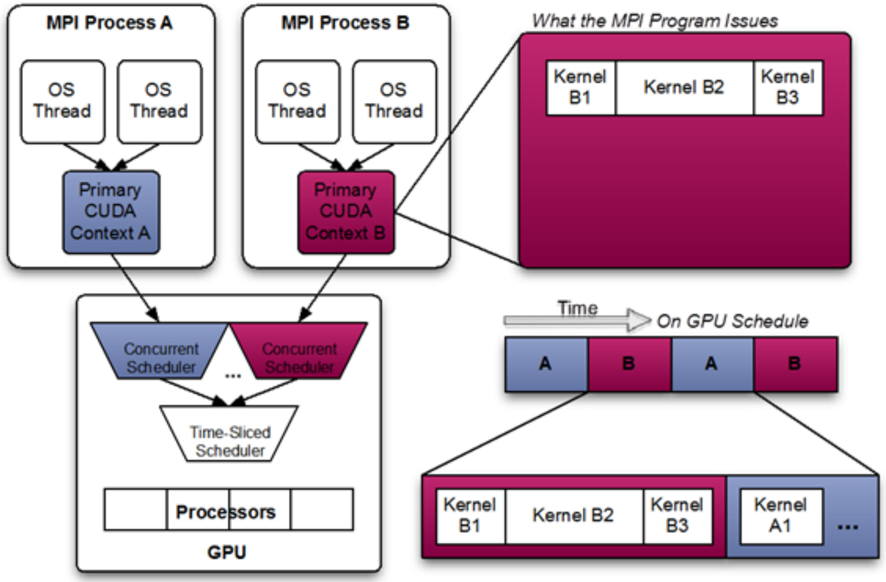
Introduction: Multi-DNN Inference Serving System

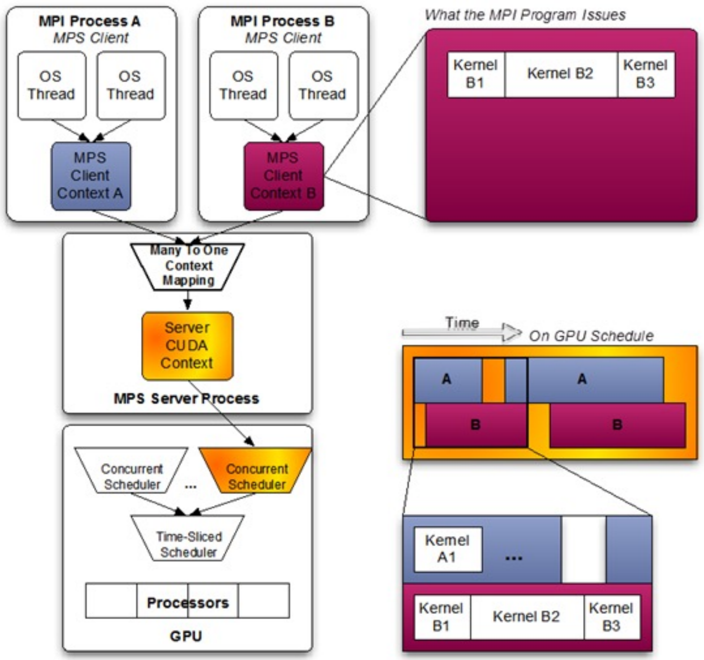
- A driving scenario
  - Real-time task: obstacle detection
  - Best-effort task: fatigue detection

Introduction: Multi-DNN Inference Serving System

- MOT: Execute Multiple SOT process

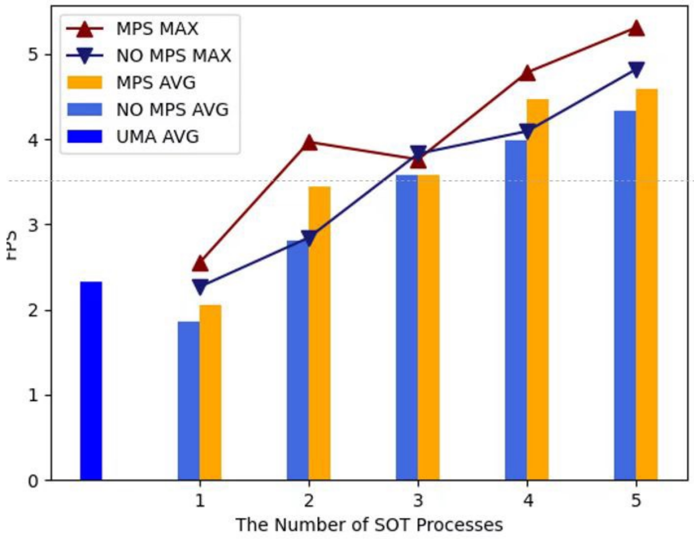
Background: NVIDIA MPS



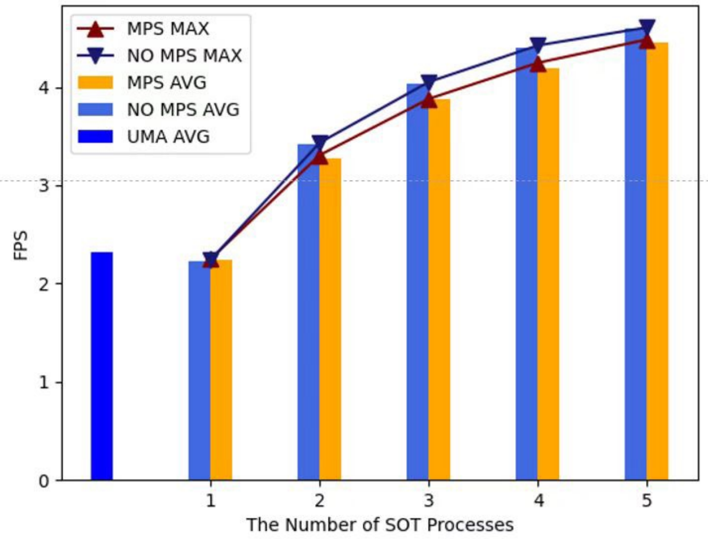


Introduction: Multi-DNN Inference Serving System

- MOT: Execute Multiple SOT process



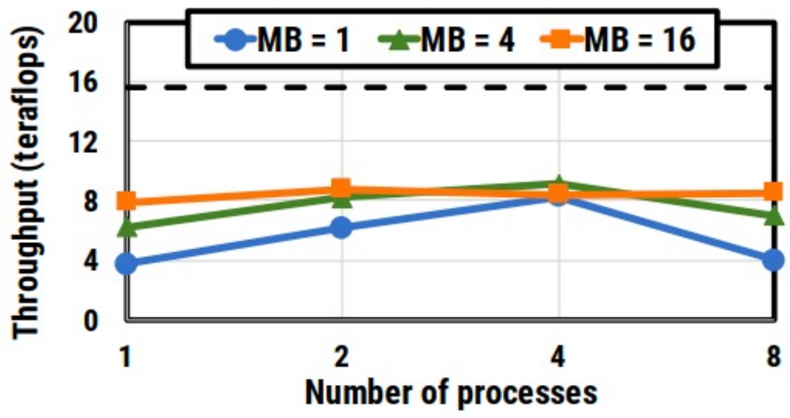
a. 基于共享内存的实现

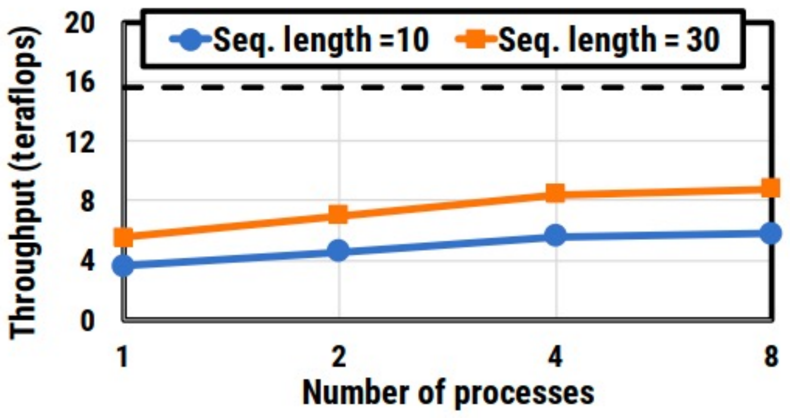


b. 基于文件存取的实现

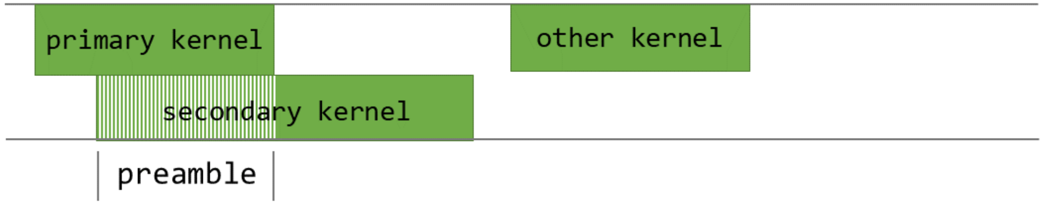
Introduction: MPS problem

- HiveMind (NIPS’18 MLSys workshop)





Background: GPU Streams

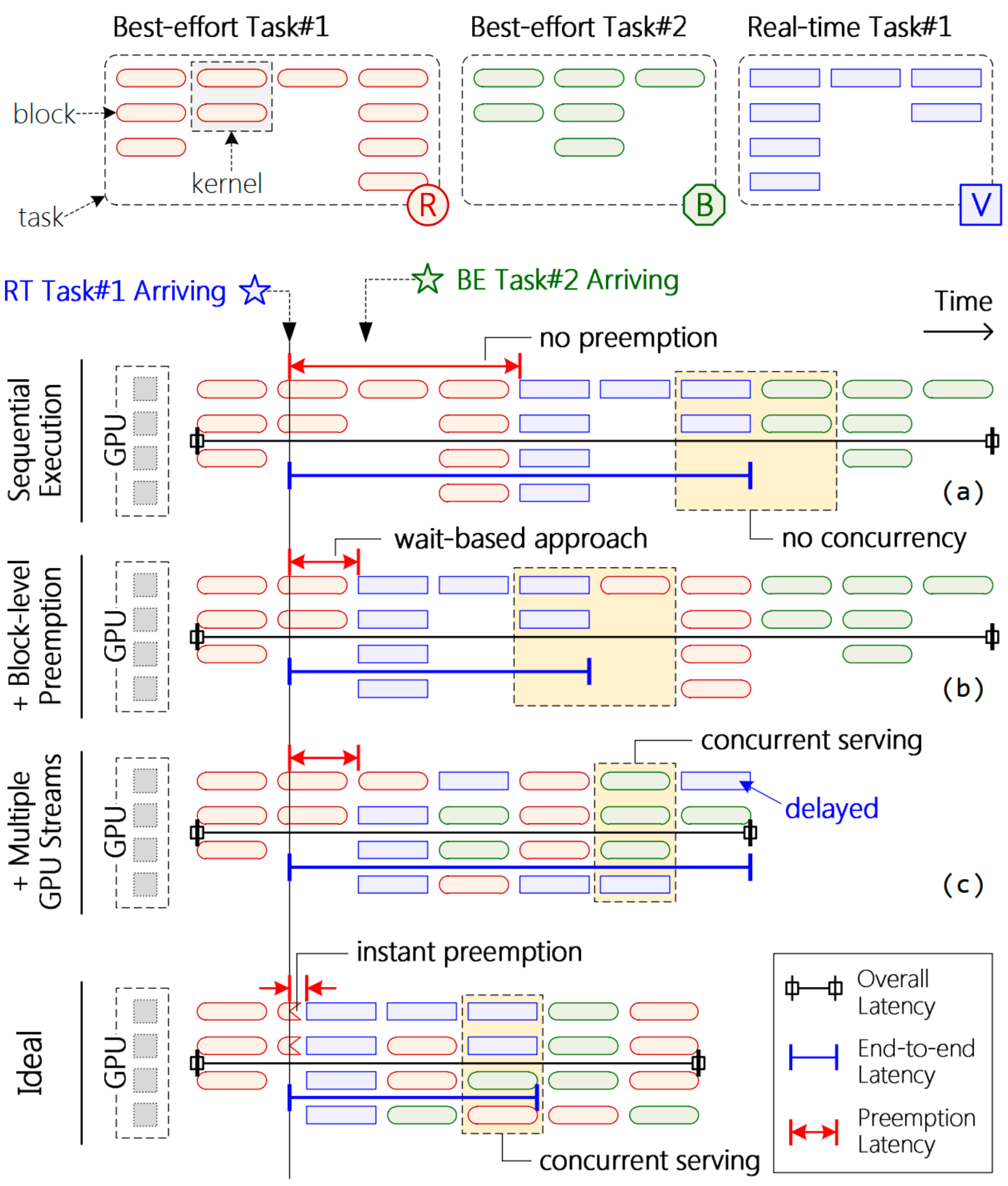



```
__global__ void primary_kernel() {
   // Initial work that should finish before starting secondary kernel

   // Trigger the secondary kernel
   cudaTriggerProgrammaticLaunchCompletion();

   // Work that can coincide with the secondary kernel
}

__global__ void secondary_kernel()
{
   // Independent work

   // Will block until all primary kernels the secondary kernel is dependent on have completed and flushed results to global memory
   cudaGridDependencySynchronize();

   // Dependent work
}

cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attribute[0].val.programmaticStreamSerializationAllowed = 1;
configSecondary.attrs = attribute;
configSecondary.numAttrs = 1;

primary_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaLaunchKernelEx(&configSecondary, secondary_kernel);
```

Motivation

- **Goal**: Real-time task can be launched quickly with low latency and execute best-effort task concurrently to achieve higher overall throughput.
- Existing methods:
  - Sequential execution
  - Wait-based block level preemption
  - Execute with multiple GPU streams

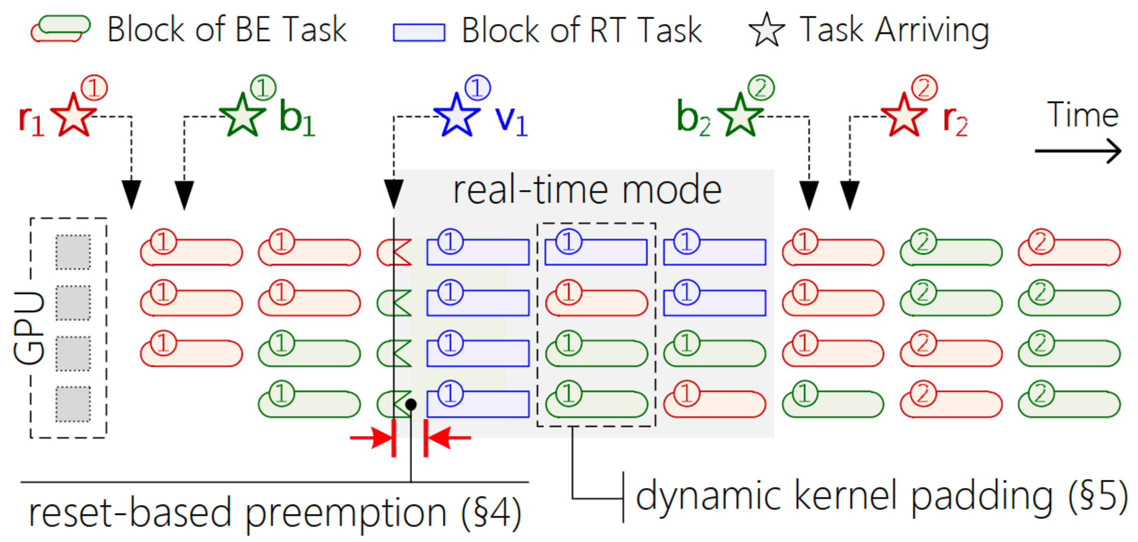
Motivation



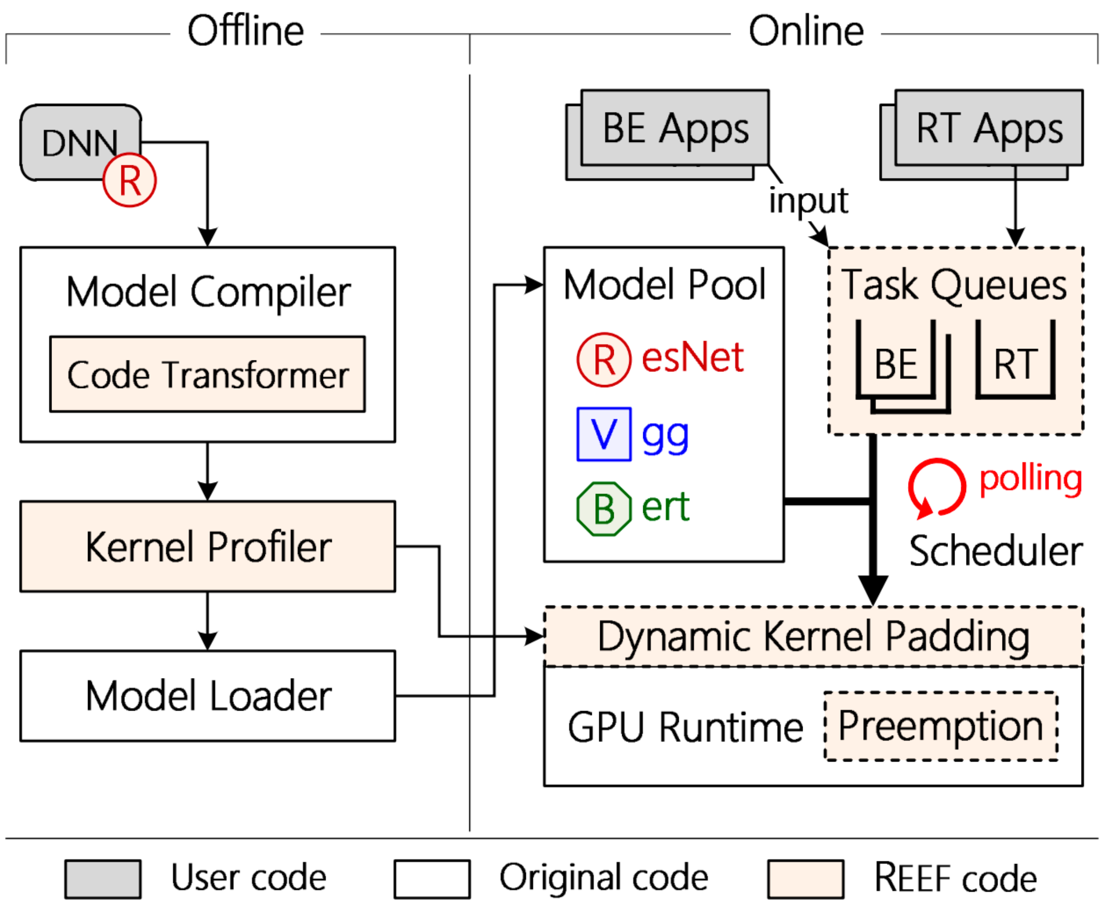
Best-effort Task#1
Best-effort Task#2
Real-time Task#1
block
kernel
task
R
B
V
RT Task#1 Arriving
BE Task#2 Arriving
Time
no preemption
Sequential Execution
GPU
(a)
no concurrency
wait-based approach
+ Block-level Preemption
GPU
(b)
concurrent serving
delayed
+ Multiple GPU Streams
GPU
(c)
instant preemption
Ideal
GPU
concurrent serving
Overall Latency
End-to-end Latency
Preemption Latency

REEF System

- How to preempt with low overhead?
  - Reset-based Preemption
- How to execute real-time and best-effort tasks concurrently and efficiently?
  - Dynamic Kernel Padding

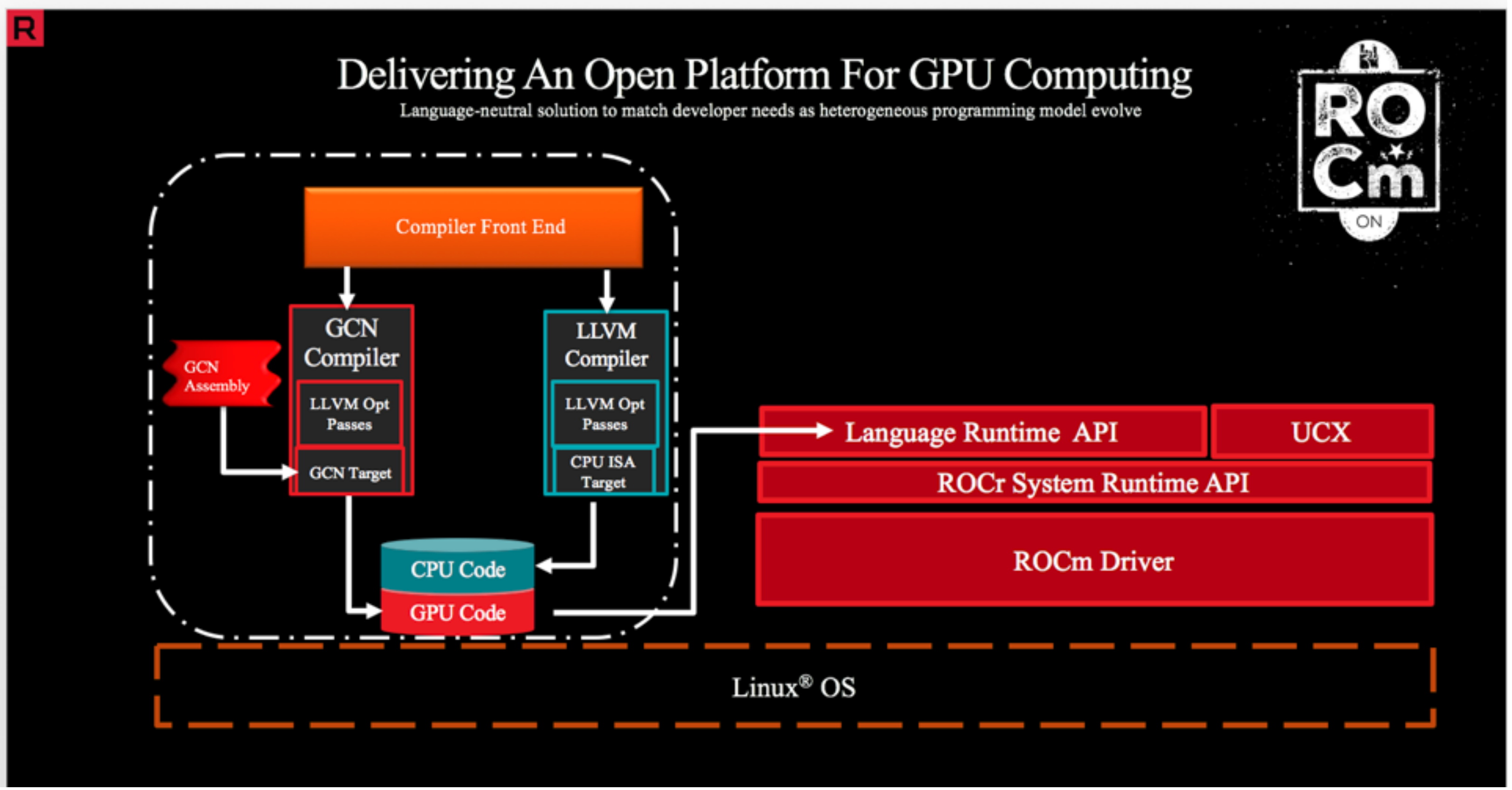


REEF System

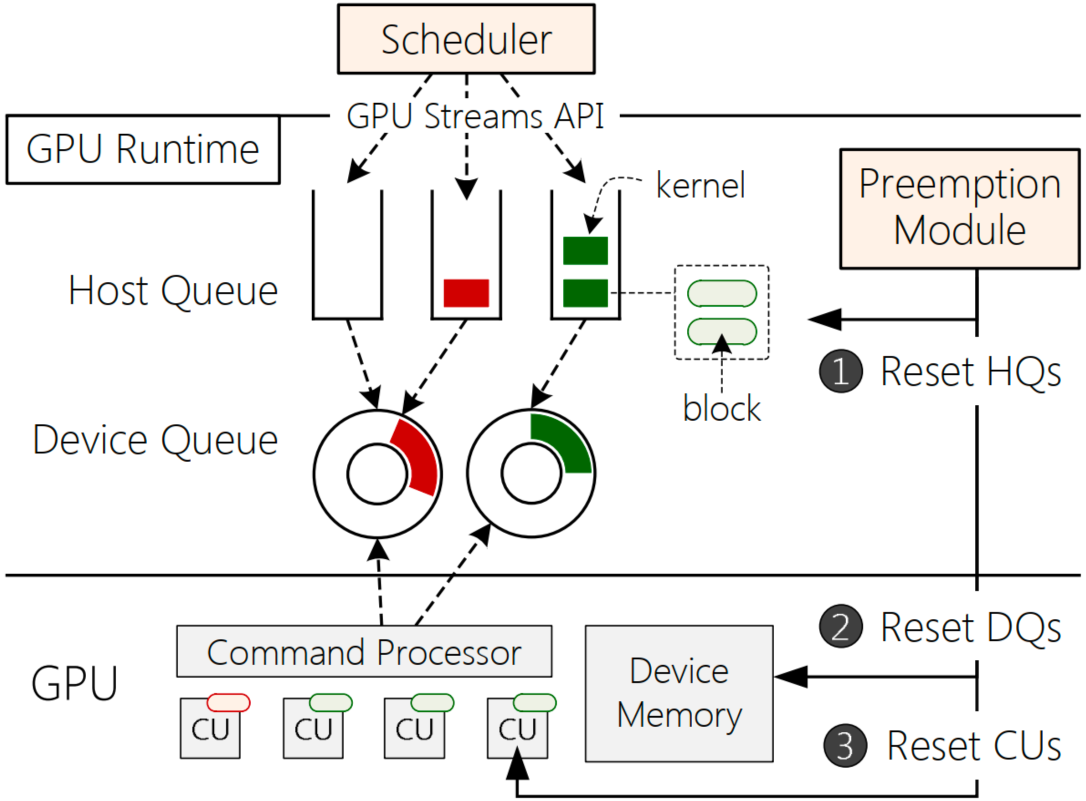


- **Code Transformer**: Based on Apache TVM to compile the provided DNN inference program.
- Kernel Profiler
- BE and RT Queues
- Dynamic Kernel Padding
- Preemption Module

Background: ROCm



Reset-based Preemtion



- Reset kernels in everywhere
  - Host Queue: reset by GPU runtime
  - Device Queue: lazy eviction, add a preemption flag in advance when this strem is been preempted

```cpp
// step 1: reset device queue
// actually, this step should be the second one,
// but we can overlap this with the host queue reset.
ASSERT_GPU_ERROR(GPUWriteValue32Async(preempt_stream, preempt_flag, 1, 0));
auto num_be_queues = be_queue_cnt;

// step 2: reset host queue
for (int i = 0; i < num_be_queues; i++) {
    uint64_t temp;
    ASSERT_GPU_ERROR(GPUClearHostQueue(&temp, be_queues[i]->stream));
    if (!be_queues[i]->task_queue.empty()) {
        auto task = be_queues[i]->task_queue.front();
        if (task->state == TaskState::Executing) {
            LOG(INFO) << task->kernel_offset << ", " << task->launch_offset;
            task->kernel_offset = std::max(
                task->launch_offset - (int)temp - be_stream_device_queue_cap,
                task->kernel_offset
            );
            LOG(INFO) << "new kernel_offset " << task->kernel_offset;
            task->state = TaskState::Waiting;
            task->preempted = true;
        }
    }
}
```

Reset

```cpp
extern "C" __global__ void add(int* preempted, int* task_slo
    if (*preempted) return;
    add_device(a, b, temp);
    if (threadIdx.x + threadIdx.y + threadIdx.z == 0)
        atomicAdd(task_slot, 1);
```

Kernel

Reset-based Preemtion

- Overhead come from:
  - Reclaiming memory from the host queue
  - Waiting to fetch kernels from the device queue

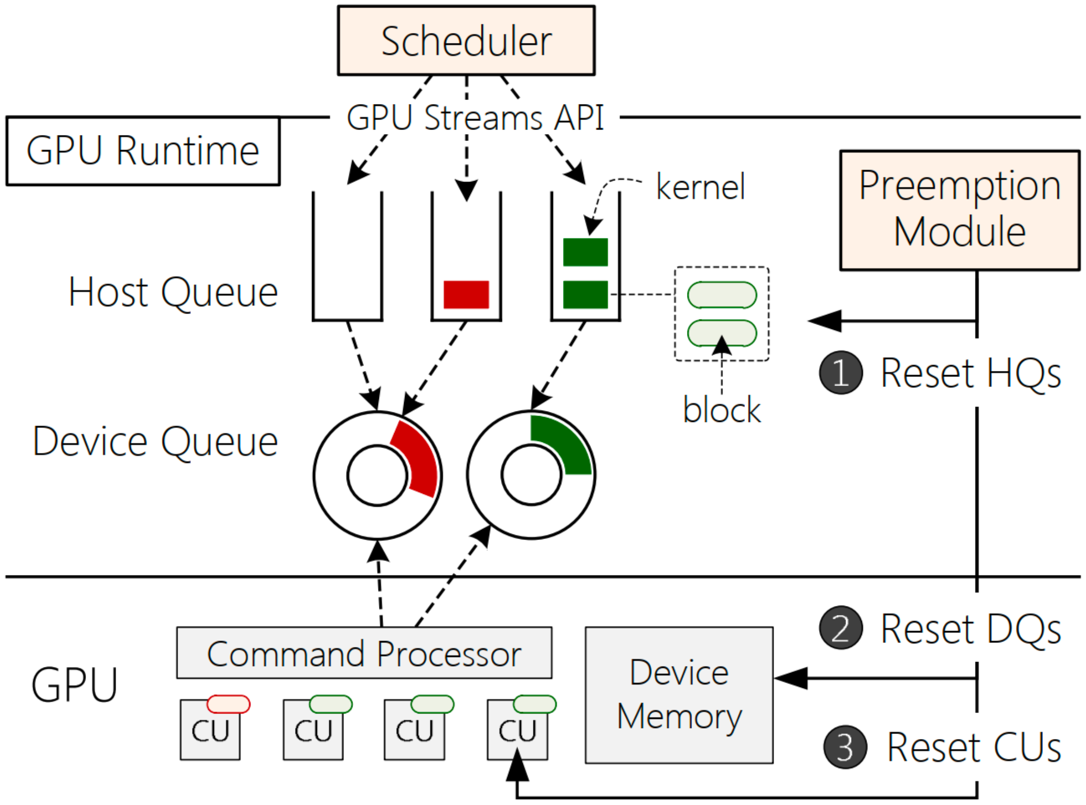
Reset-based Preemtion

- Asynchronous memory reclamation
  - Background GC thread to reclaim memory asynchronous

- Device queue capacity restriction

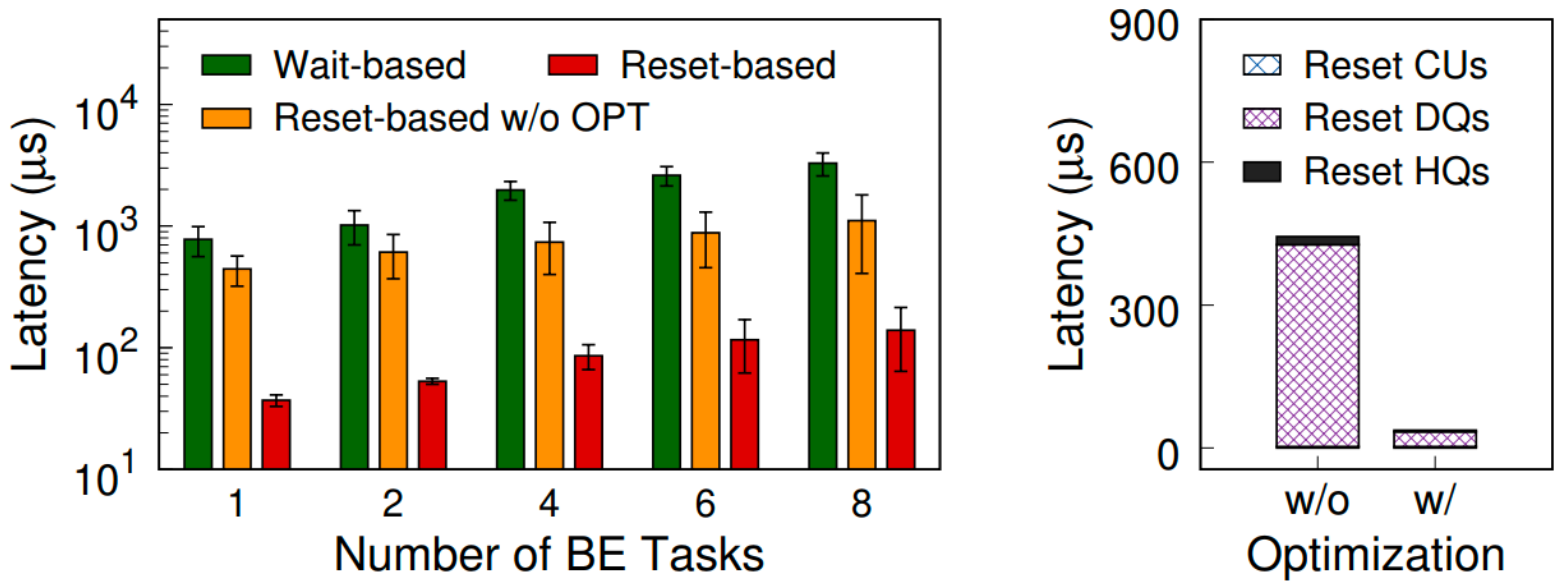
```
void REEFScheduler::set_be_stream_cap(int value) {
    be_stream_device_queue_cap = value;
}
```

Reset-based Preemtion

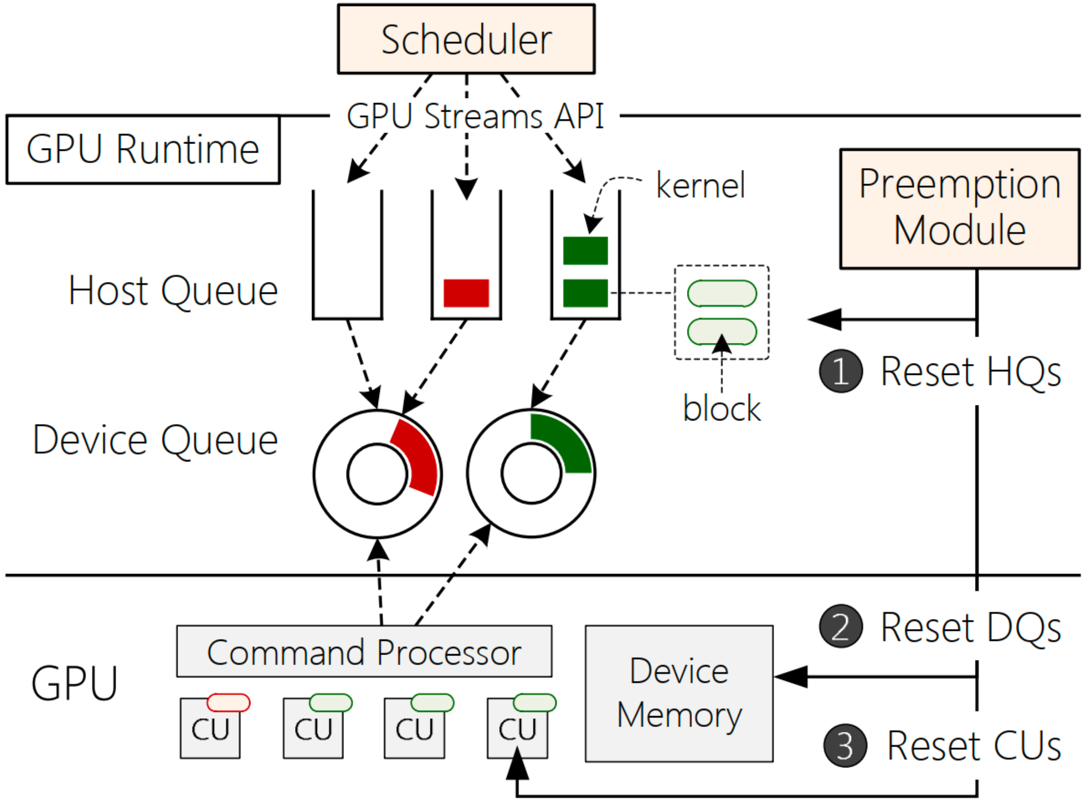


- Reset kernels in everywhere
  - CUs: Rewriting kill function in GPU driver

Reset-based Preemtion



Reset-based Preemtion (REEF-N version)



- GPU runtime is a black box
- Cannot manipulate GPU runtime directly
- vHQs are outside the GPU runtime

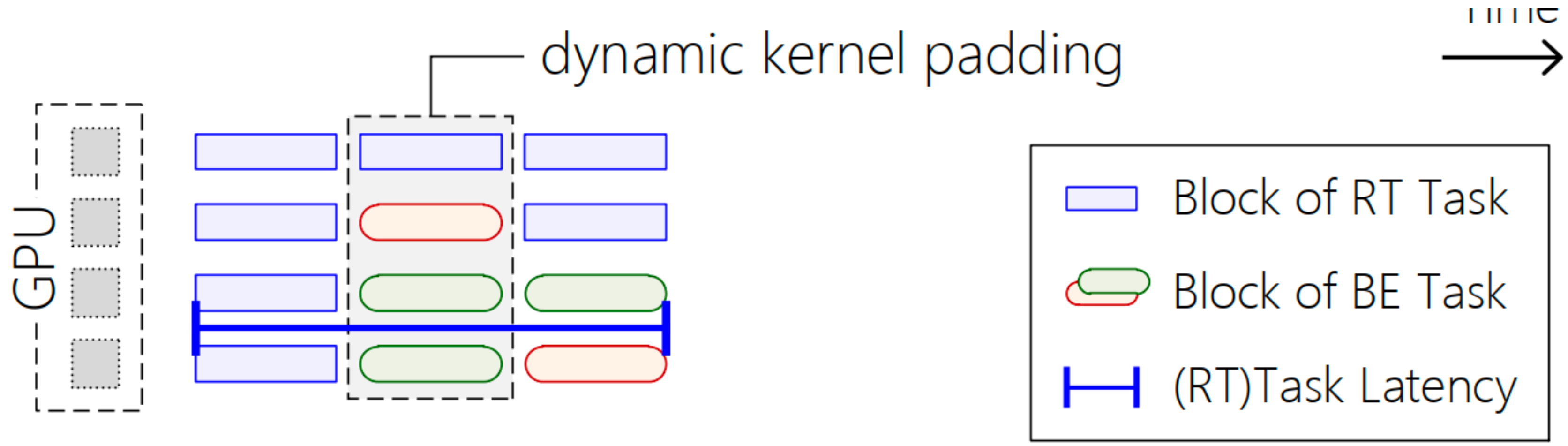
- DQs
- CUs cannot be reset

Reset-based Preemtion

- Restoring preempted tasks
  - Restore the preempted task from the kernel close to where it was interrupted
  - Approxiamation approach to ensure the integrity of re-execution

```
Status HybridExecutor::launch_preempt_kernel(int kernel_offset, GPUStream_t stream) {
    KernelArg &kernel_arg = trans_args[kernel_offset];
    GPUFunction_t func = preempt_kernels[kernel_offset];
    // LOG(INFO) << "launch " << kernel_offset;
    GPUDevicePtr_t task_slot = (GPUDevicePtr_t)((char*)task_slot_base + kernel_offset * 4);
    this->preempt_args[kernel_offset][1] = &task_slot; // TODO:
    GPU_RETURN_STATUS(GPUModuleLaunchKernel(func,
        kernel_arg.task_dim.x, kernel_arg.task_dim.y, kernel_arg.task_dim.z,
        kernel_arg.thread_dim.x, kernel_arg.thread_dim.y, kernel_arg.thread_dim.z,
        0, stream, (void**)(this->preempt_args[kernel_offset].data()), 0
    ));
    return Status::Succ;
}
```

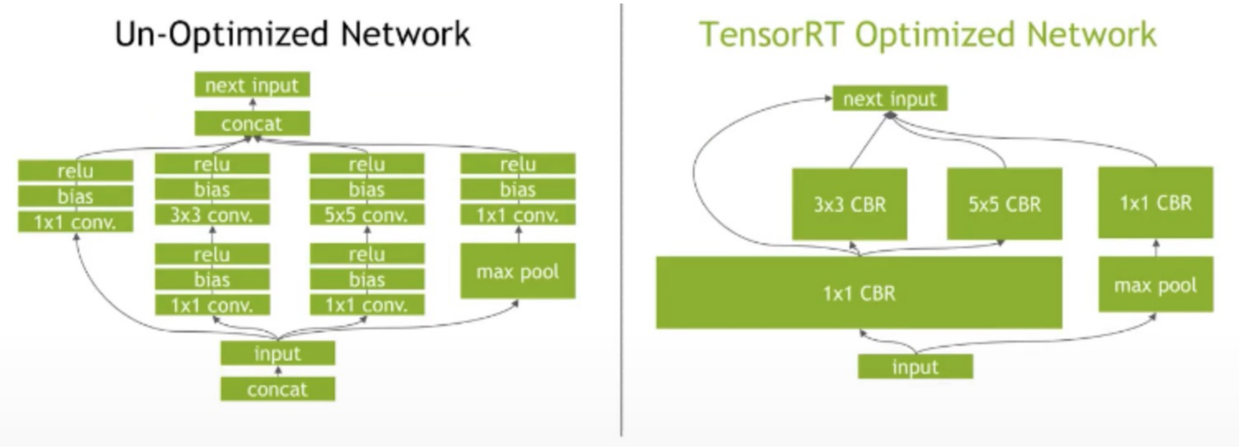
Dynamic Kernel Padding



Dynamic Kernel Padding

```
# device codes
__device__ void dense(in, weight, bias, out): ...

__global__ void dkp(rt_kern, rt_args,
                    be_kerns, be_argss):
    ncus = rt_kern.ncus  # number of CUs
    if (cu_id() < ncus) then
       rt_kern(rt_args)  # run RT/kernel
    else
       ncus += be_kerns[i=0].ncus
       while (cu_id() >= ncus)
          ncus += be_kerns[++i].ncus
       be_kerns[i](be_argss[i])  # run BE/kernel

# host codes
void inference(...):
    # set the real-time kernel w/ its args (e.g., dense)
    rt_kern, rt_args = ...
    # select a set of best-effort kernels w/ their args
    be_kerns, be_argss = kern_select(rt_kern)
    dkp <<<..>>> (rt_kern, rt_args, be_kerns, be_argss)
    ... # launch other dynamic padded kernels
```

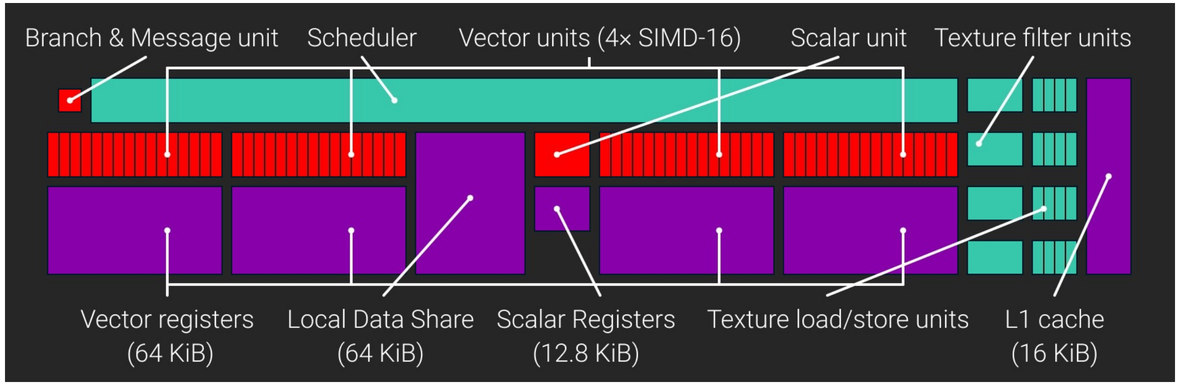
- Inspired by kernel fusion



Dynamic Kernel Padding

- Global function pointer
  - Limited register allocation
  - Expensive context saving

```
__device__ void dense(in, weight, bias, out): ...
__global__ void dkp(rt_kern, rt_args,
                    be_kerns, be_argss):
```



Dynamic Kernel Padding

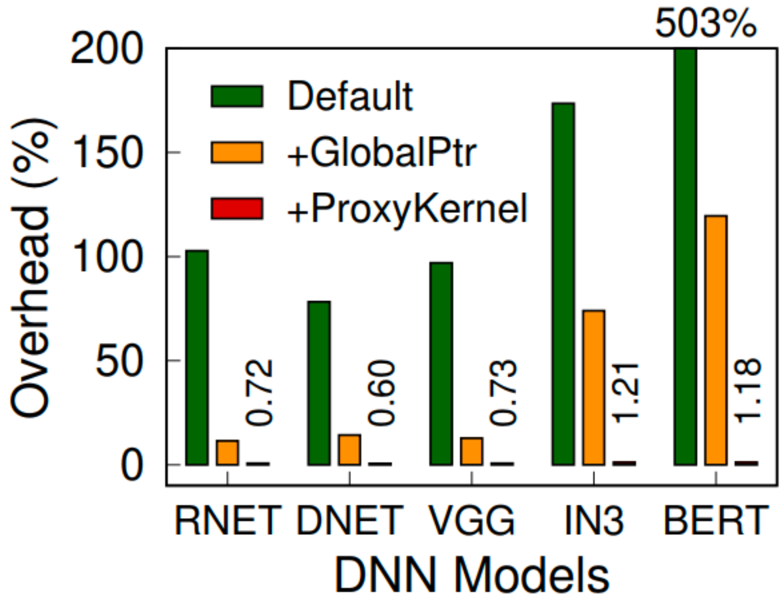
- Dynamic register allocation
  - To solve over-allocation problem
  - Proxy kernels: Multiple kernel candidate with varied ccupancy grain rather than register combination grain

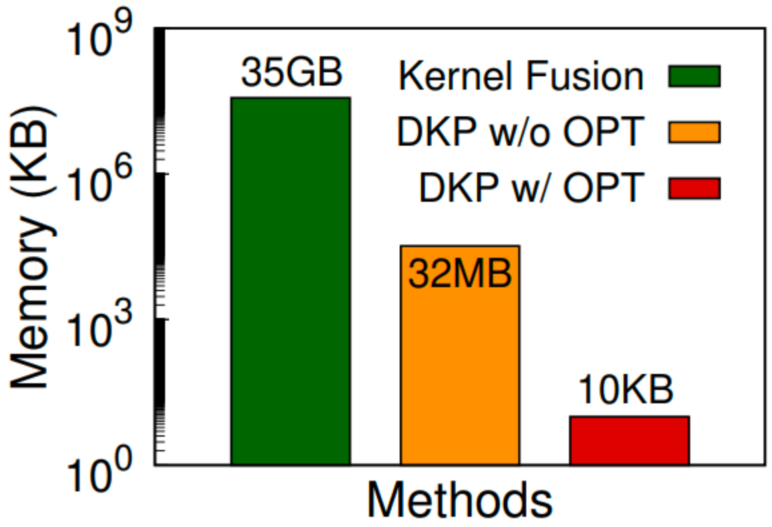
```
extern "C" __device__ __noinline__ dim3 get_3d_idx_64_1_1(int idx) {
    dim3 dim(64, 1, 1);
    dim3 result;
    result.x = idx % dim.x;
    result.y = idx / dim.x % dim.y;
    result.z = idx / (dim.x * dim.y);
    return result;
}

extern "C" __device__ __noinline__ dim3 get_3d_idx_4_8_4(int idx) {
    dim3 dim(4, 8, 4);
    dim3 result;
    result.x = idx % dim.x;
    result.y = idx / dim.x % dim.y;
    result.z = idx / (dim.x * dim.y);
    return result;
}
```

Dynamic Kernel Padding

- Dynamic shared memory
  - Add *extern* before __*shared*__
  - Change from launch by launch

Dynamic Kernel Padding



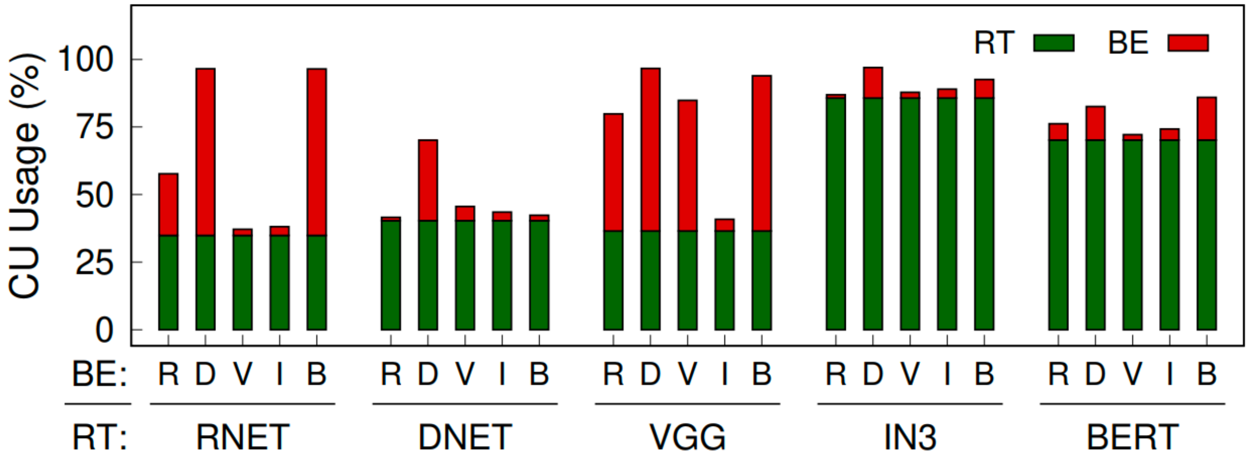


Dynamic Kernel Padding

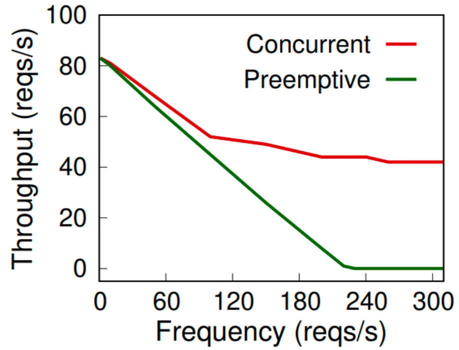
- Kernel selection
  - Rule1: The execution time of best-effort kernels must be shorter than that of the real-time kernel.
  - Rule2: The CU occupancy of best-effort kernels must be higher than that of the real-time kernel.

Dynamic Kernel Padding

- Kernel selection

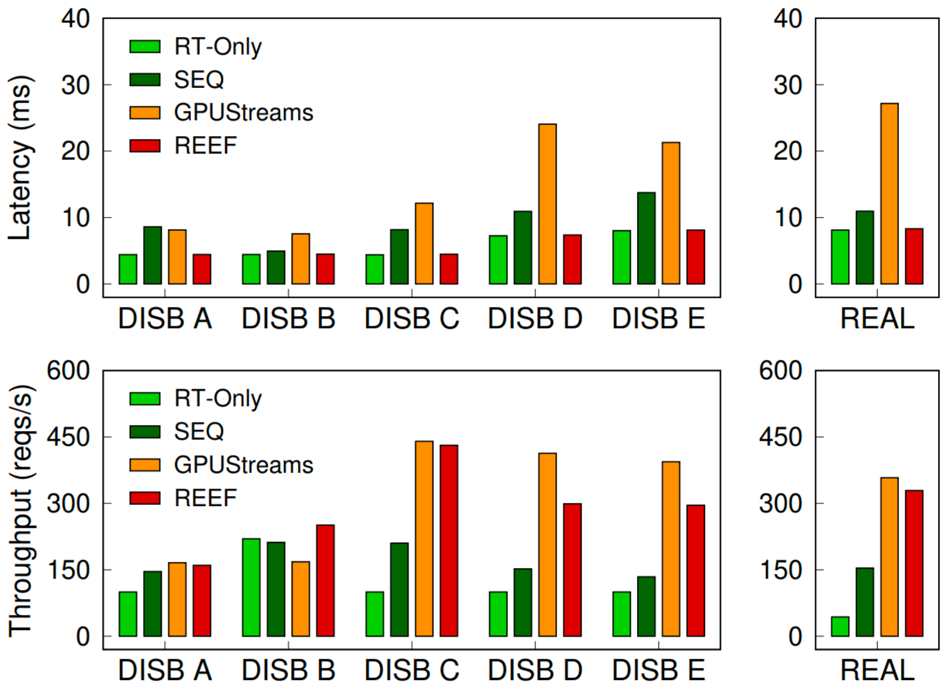


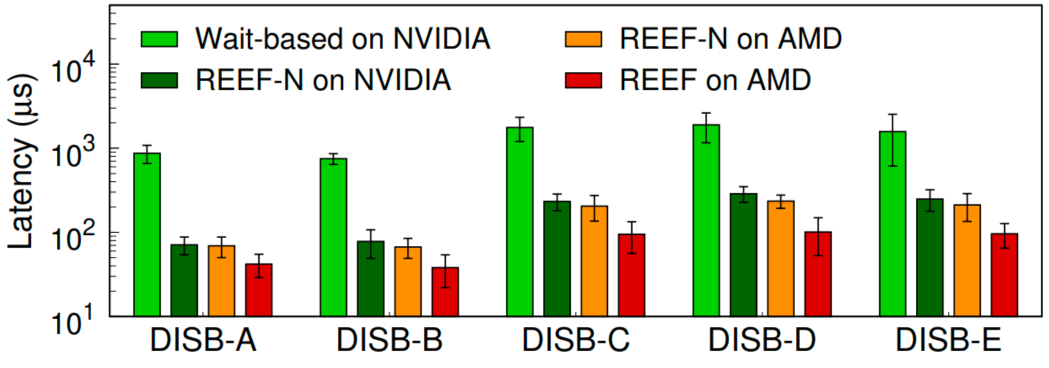
Too conservative?



Evaluation

DISB	A	B	C	D	E
Num. of RT clients	1/VGG	1/VGG	1/VGG	5/ALL	5/ALL
Frequency (reqs/s)	100 [U]	220 [U]	100 [U]	20 [U]	20 [P]
Num. of BE clients	1/RNET	1/RNET	5/ALL	5/ALL	5/ALL





Inspiration

- An in-depth analysis to the GPU programming.
  - Open source research: based on ROCm.
- **Clear goal**: microsecond-scale preemption and best-effort throughput.
- Block-level view in GPU resource scheduling.
- TVM as compiler to transform code.

Thank You!

Feb 6, 2023

Presented by Mengyang Liu