Machine Learning & GPUs

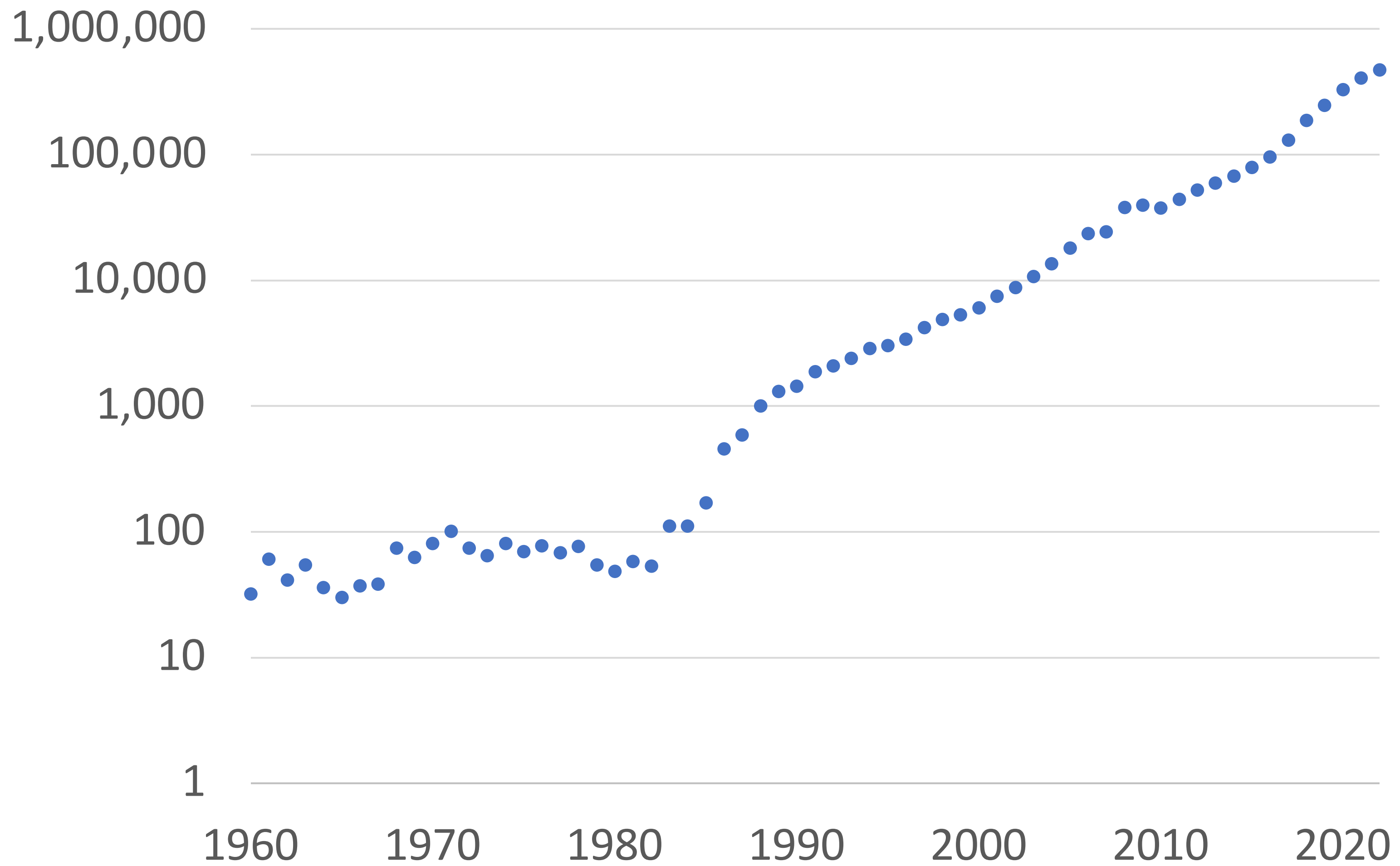
ARCHI 2023

David Novo

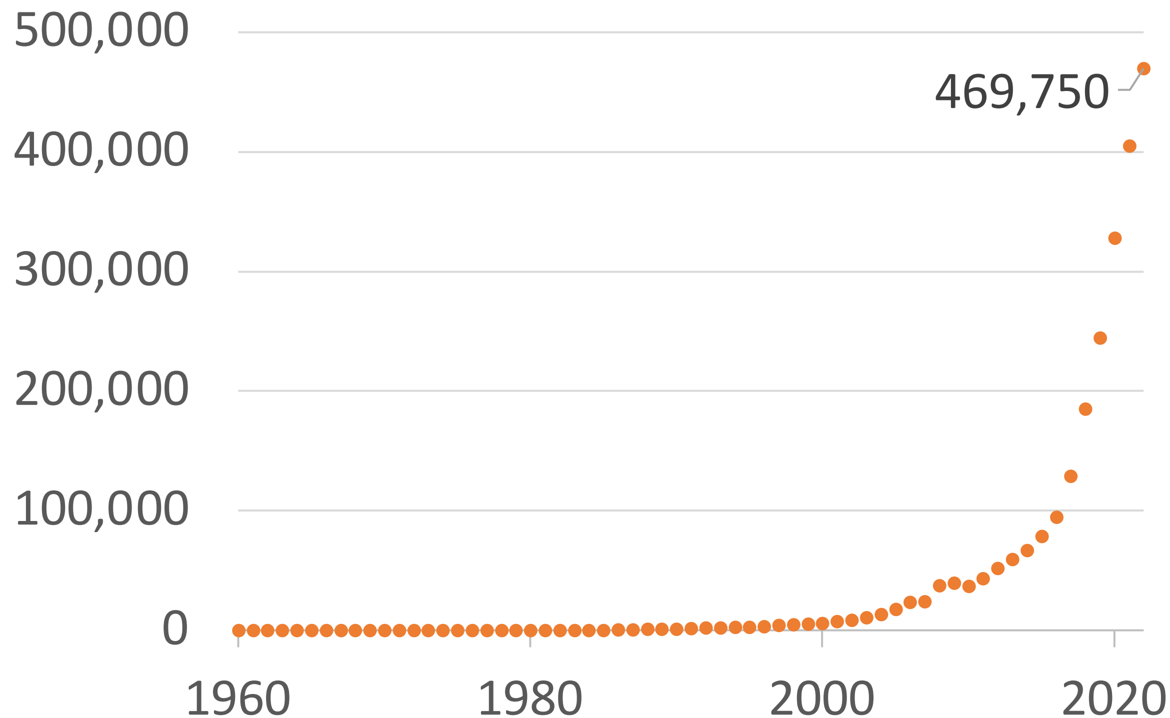
Chargé de Recherche CNRS

https://www.lirmm.fr/david-novo/

Another exponential



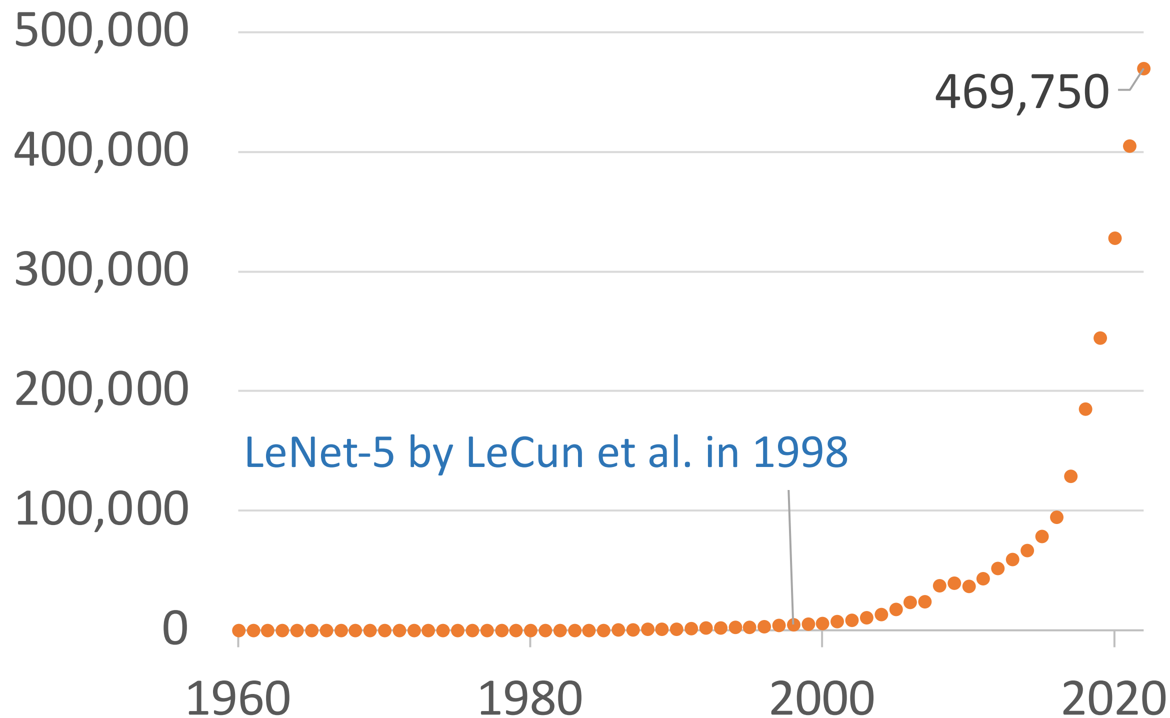
Publications on Machine Learning per year



The term Machine Learning was coined in 1959 by Arthur Samuel, an IBM employee.

-- Wikipedia

Publications on Machine Learning per year



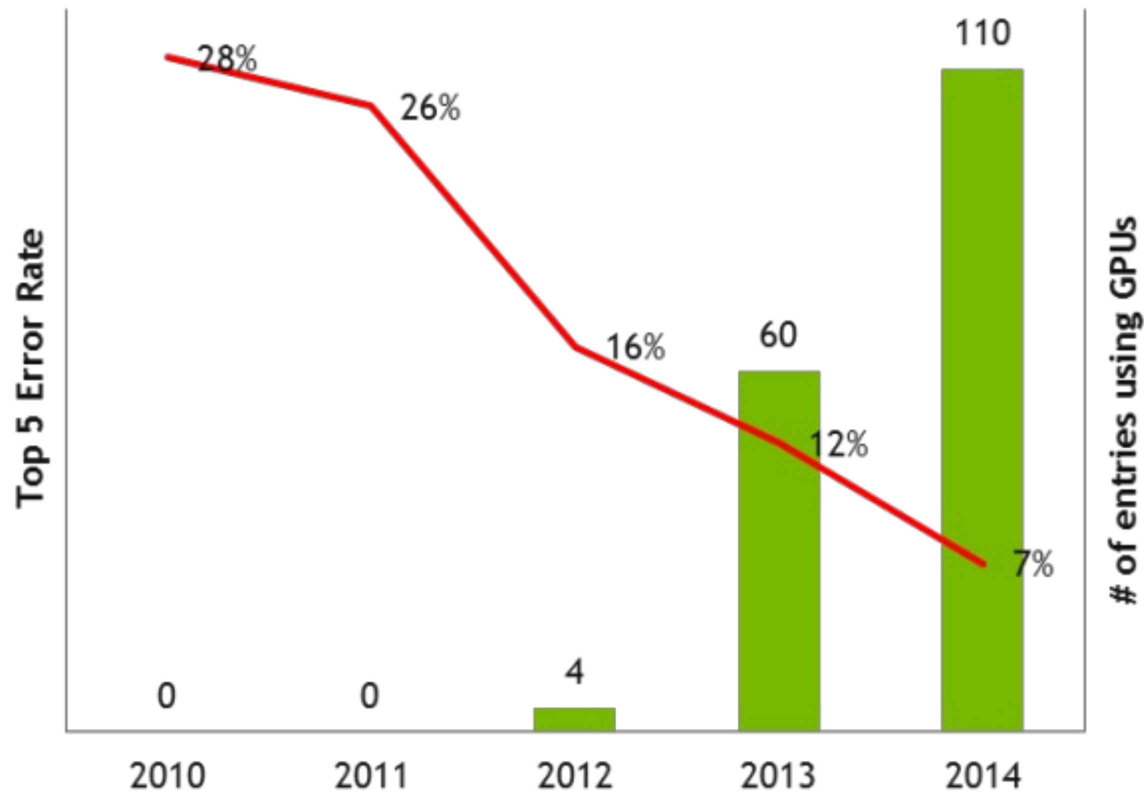
LeCun et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE. 1998

The catalyst of the ML revolution

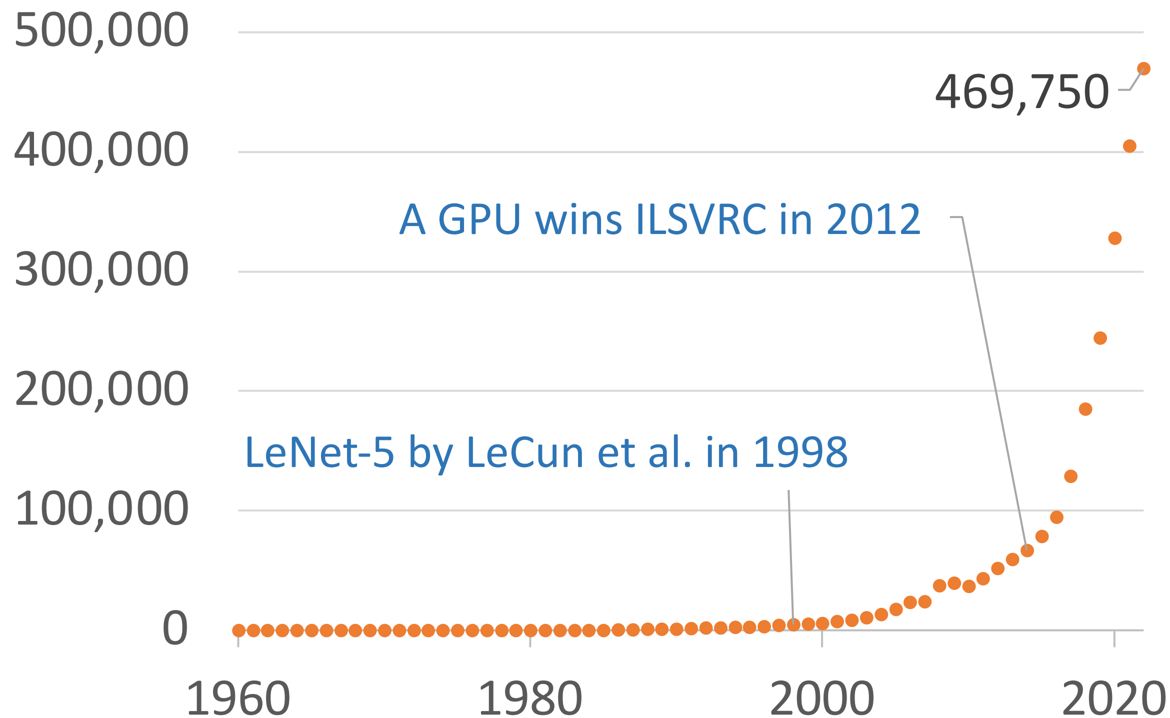
The ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- Identify 1000 classes (e.g., goldfish, collie, etc.)

GPUs revolutionized machine learning



Publications on Machine Learning per year



Krizhevsky et al. “ImageNet classification with deep convolutional neural networks.” NeurIPS. 2012

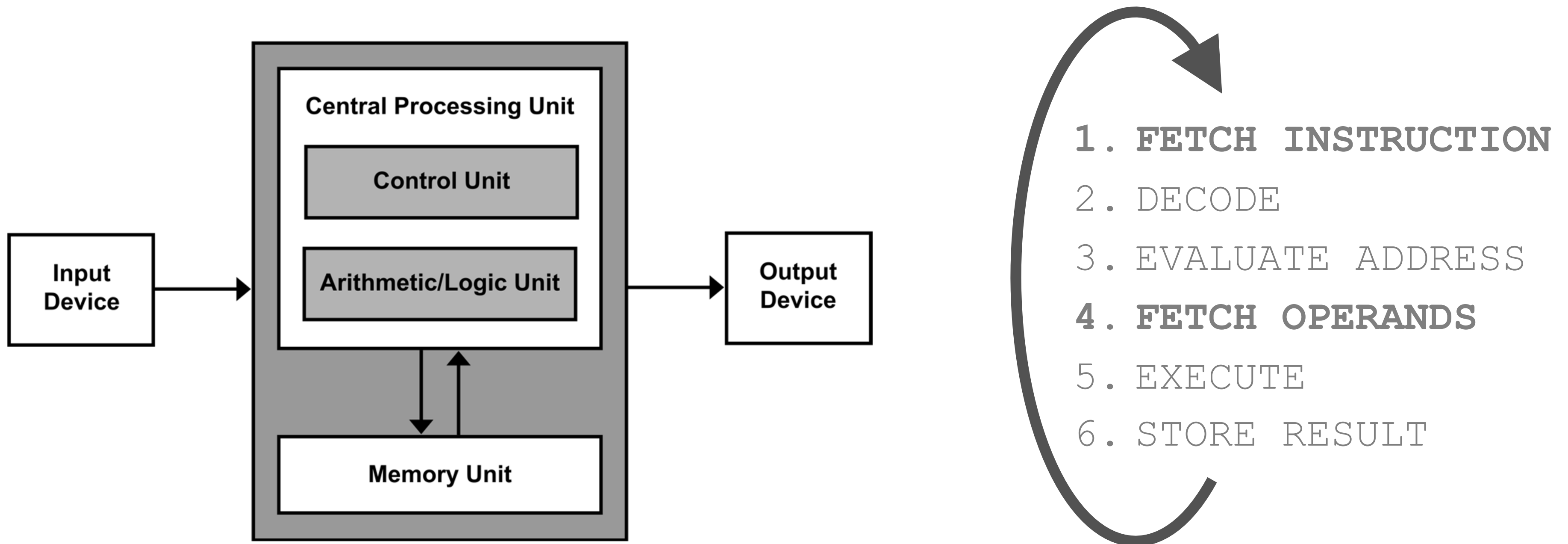
Outline

1. What is a GPU?

2. How ML frameworks use GPUs

The von Neumann architecture

A von Neumann architecture = stored-program computer

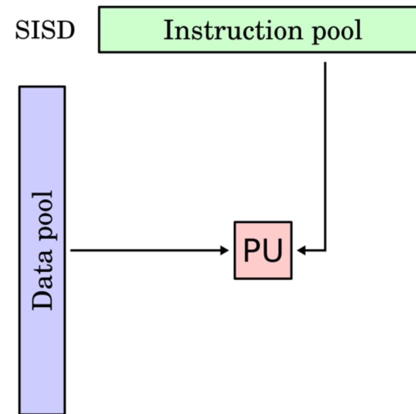


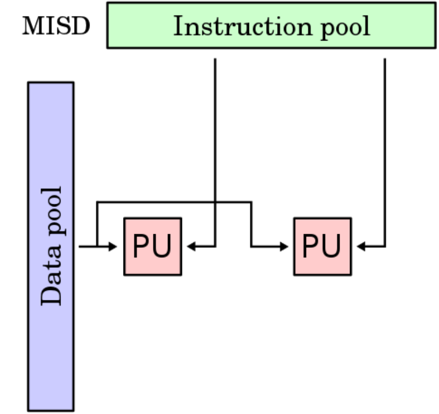
von Neumann. "First Draft of a Report on the EDVAC." 1945

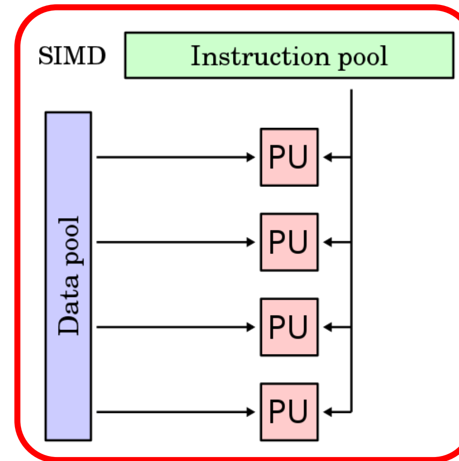
The Flynn's taxonomy

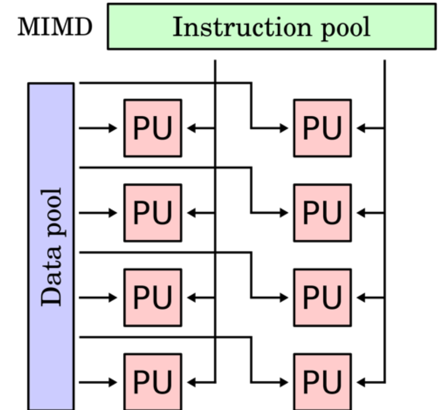
Classification of computer architectures:

- **S**ingle **I**nstruction, **S**ingle **D**ata
- **M**ultiple **I**nstructions, **S**ingle **D**ata
- **S**ingle **I**nstruction, **M**ultiple **D**ata
- **M**ultiple **I**nstructions, **M**ultiple **D**ata









Flynn. "Very high-speed computing systems." Proceedings of the IEEE. 1966

SIMD instruction set extensions

SIMD machines are good at exploiting regular data-level parallelism

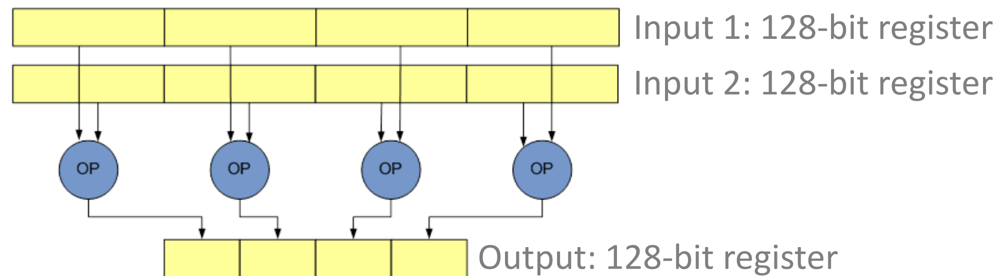
- Improve performance, simplify design, improve energy efficiency (less instruction need to be fetched)

Many existing ISAs include SIMD instructions

- ARM Advanced SIMD (Neon)/ Scalable Vector Extensions (SVE), Intel MMX/SSEn/AVX, PowerPC AltiVec, etc.

```
add v0.4s, v0.4s, v1.4s
```

Register v0, which includes
4 single precision (32-bit) operands



GPUs (Graphics Processing Units)

GPUs are SIMD engines underneath

- The instruction pipeline operates like an SIMD pipeline

However, we program GPUs using threads, not SIMD instructions

- Each thread executes the same code but operates a different piece of data
- Each thread has its own context (i.e., PC, registers, stack, etc.)

A set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware

- A warp is essentially a SIMD operation formed by hardware!

Programing model vs. execution model

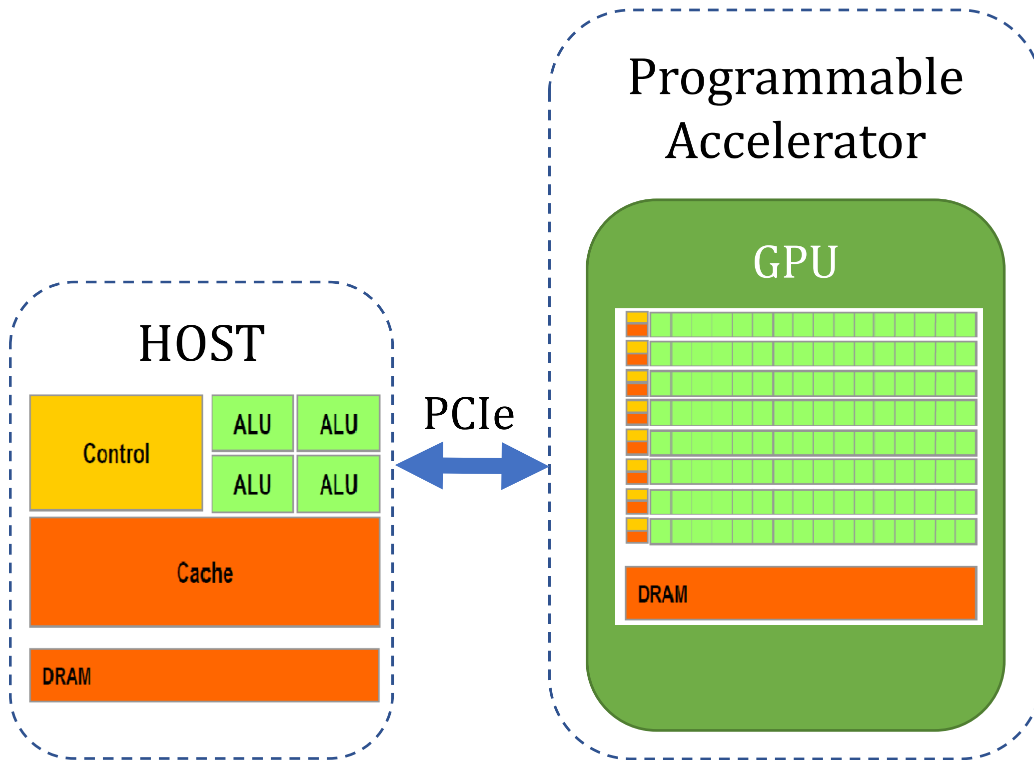
- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential, Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD: Single-Program, Multiple-Data), etc.

- Execution Model refers to how the hardware executes the code
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, etc.

The execution model does NOT need to be the same as the programming model

- E.g., a sequential model implemented by an OoO processor
- E.g., SPMD model implemented by an SIMD processor = a GPU
  - Single-Instruction Multiple Threads (SIMT) is a subclass of SIMD

The CPU-GPU tandem



A GPU is a programmable accelerator controlled by a host

- High compute density
- Favorizing parallelism vs. single core performance

- Programmable via the CUDA platform

Running example

Addition of two vectors (A & B) of T elements

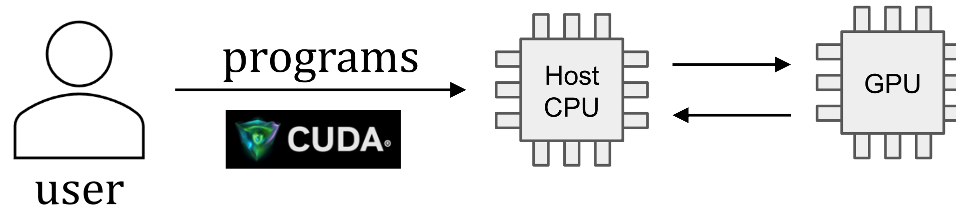
Reference CPU code

```
// Function definition
void VecAdd(float* A, float* B, float* C, int T) {
    for (int i = 0; i < T; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

```
int main() {
  ...
      // function call
      VecAdd (A, B, C, T);
  ...
}
```

CUDA overview

- CUDA is a programming abstraction built on top of C++ to program NVIDIA GPUs
- User must program the behavior of both the **CPU** and the **GPU**



- **GPU code** (kernel code) expresses the behavior of one thread

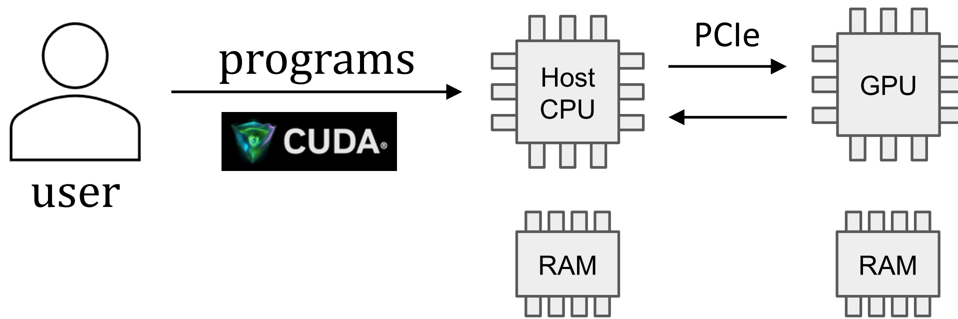
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    C[i] = A[i] + B[i];
}
```

- **CPU code** (host code) invokes the kernel on T threads

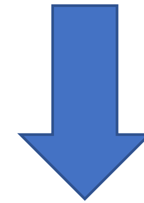
```
int main() {
  ...
      // Kernel invocation with T threads
      VecAdd<<<..., T>>>(A, B, C);
  ...
}
```

CUDA memory exchanges

To execute a kernel, the GPU needs the inputs in its dedicated memory




```
int main() {
  ...
      // Kernel invocation with T threads
      VecAdd<<<..., T>>>(A, B, C);
  ...
}
```

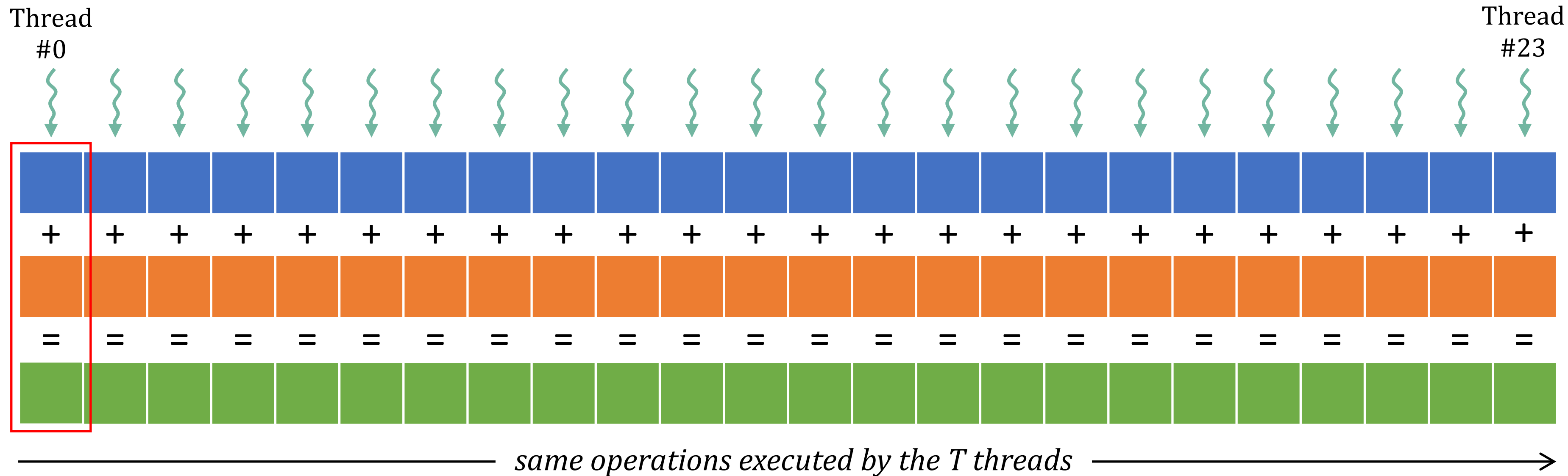


```
int main() {
      // Send inputs
      cudaMemcpy(A, …, HostToDevice);
      cudaMemcpy(B, …, HostToDevice);

      // Kernel invocation with T threads
      VecAdd<<<..., T>>>(A, B, C);

      // Receive outputs
      cudaMemcpy(C, …, DeviceToHost);
}
```

Data exchanges between CPU and GPU is also handled in the CPU code:

- MemCpy CPU->GPU (via PCIe/NVLINK)
- Execution of the Kernel (invocation)
- MemCpy GPU->CPU

Kernel implementation and invocation



- **GPU code** (kernel code) expresses the behavior of one thread
- **CPU code** (host code) invokes the kernel on T threads

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main() {
  ...
      // Kernel invocation with T threads
      VecAdd<<<..., T>>>(A, B, C);
  ...
}
```

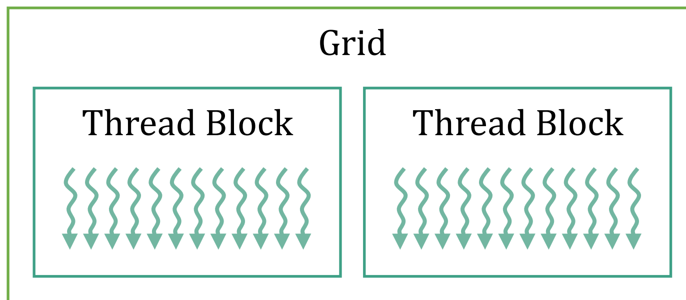
CUDA software abstractions

On kernel invocation, CUDA generates a grid of threads

- The grid includes all the threads of the kernel

All the threads in the grid are organized into multiple thread blocks

- Thread block dimensions are defined by the user



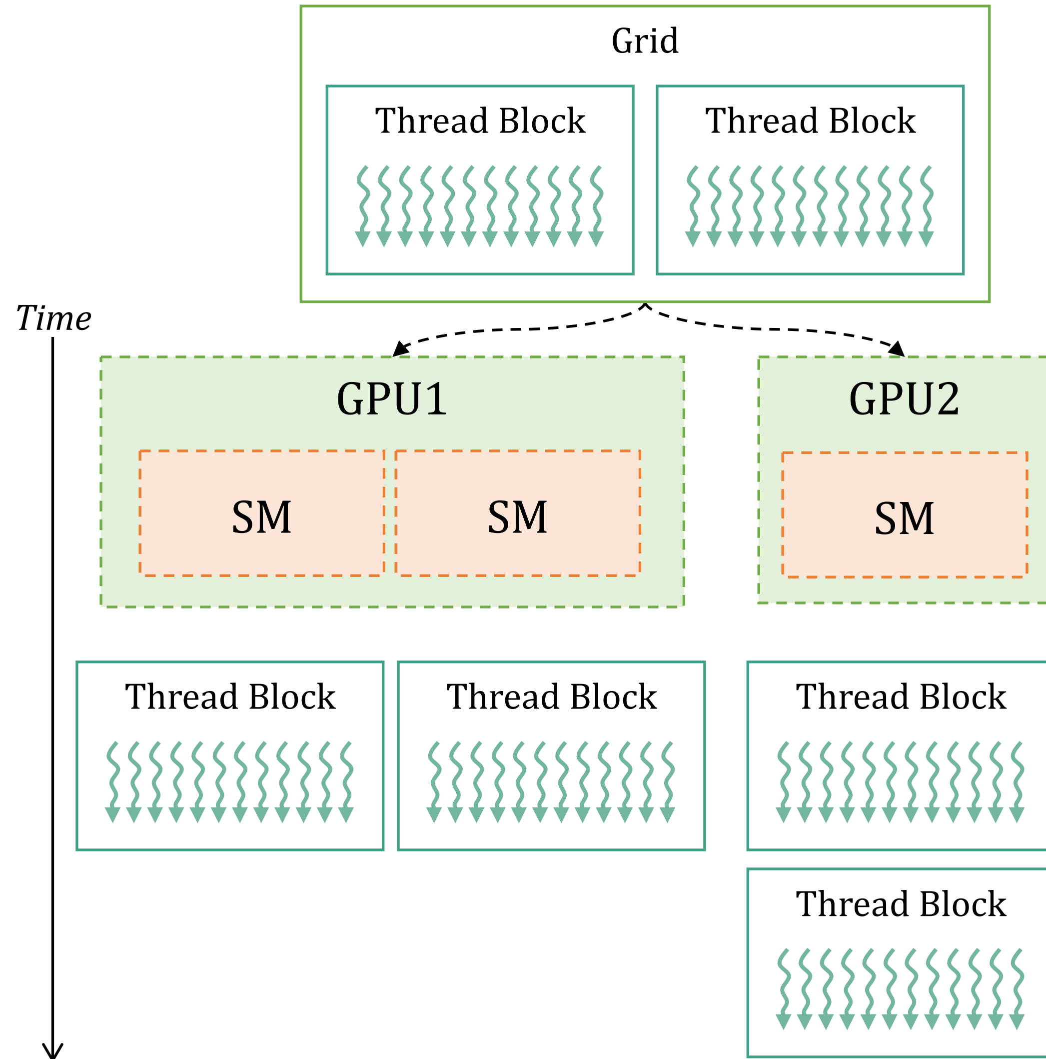
Example:

The vector addition needs
24 threads (T=24)
The user organizes all the threads into
2 thread blocks (M=2)
Each thread blocks includes
12 threads (N=T/M=12)

Automatic scalability

Each thread blocks is mapped onto a multi-core processor: Streaming Multiprocessor (SM)

The grouping in thread blocks enables automatic scalability of the workload to GPUs of different sizes (i.e., number of SMs)

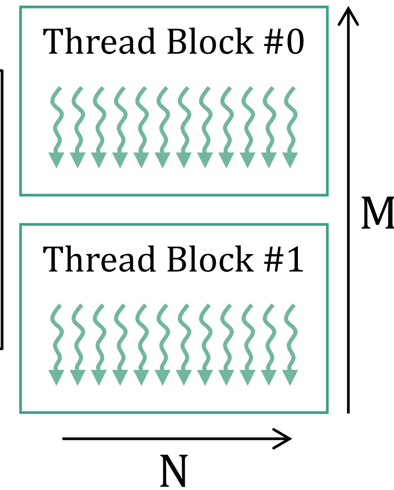


Kernel invocation: multiple blocks

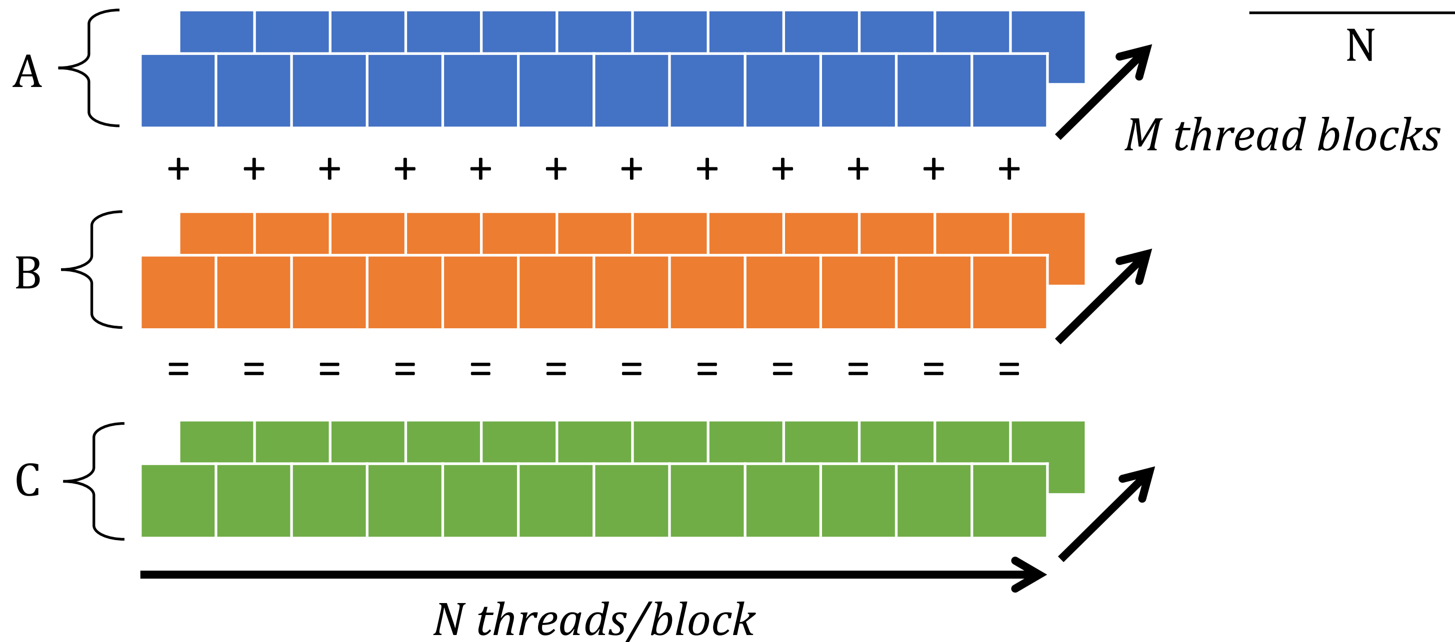
The CPU code invokes M thread blocks, each containing N threads

```
int main() {
  ...
      // Kernel configuration
      int numBlocks = M;
      int threadsPerBlock = N;

      // Kernel invocation with N threads
      VecAdd<<<numBlocks, threadPerBlocks>>>(A, B, C);
  ...
}
```

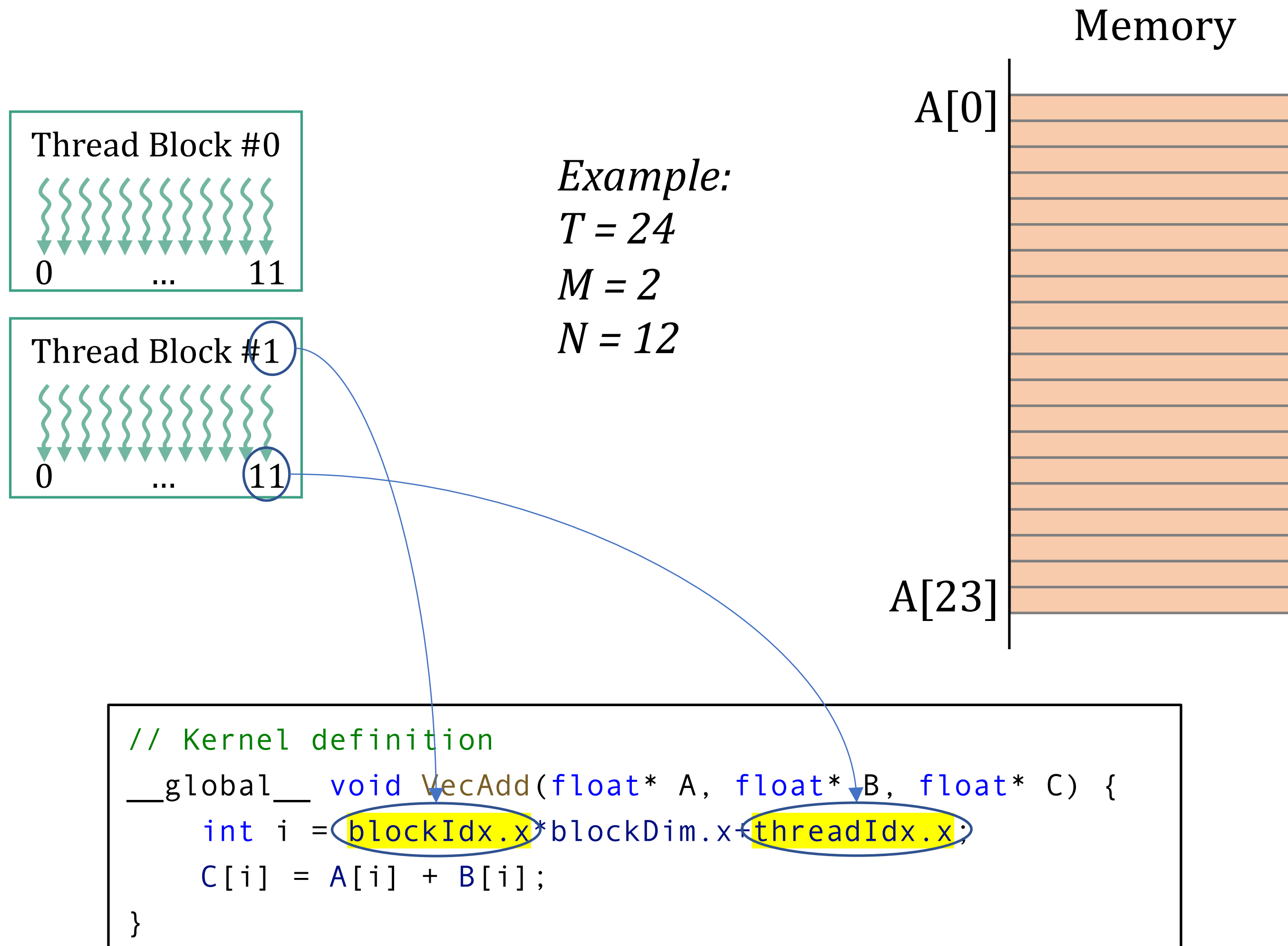
(numBlocks: M thread blocks; threadPerBlocks: N thread per block)



Example:
T = 24
M = 2
N = 12



Thread identification within the kernel



CPU code: complete example

Typical steps of the CPU code

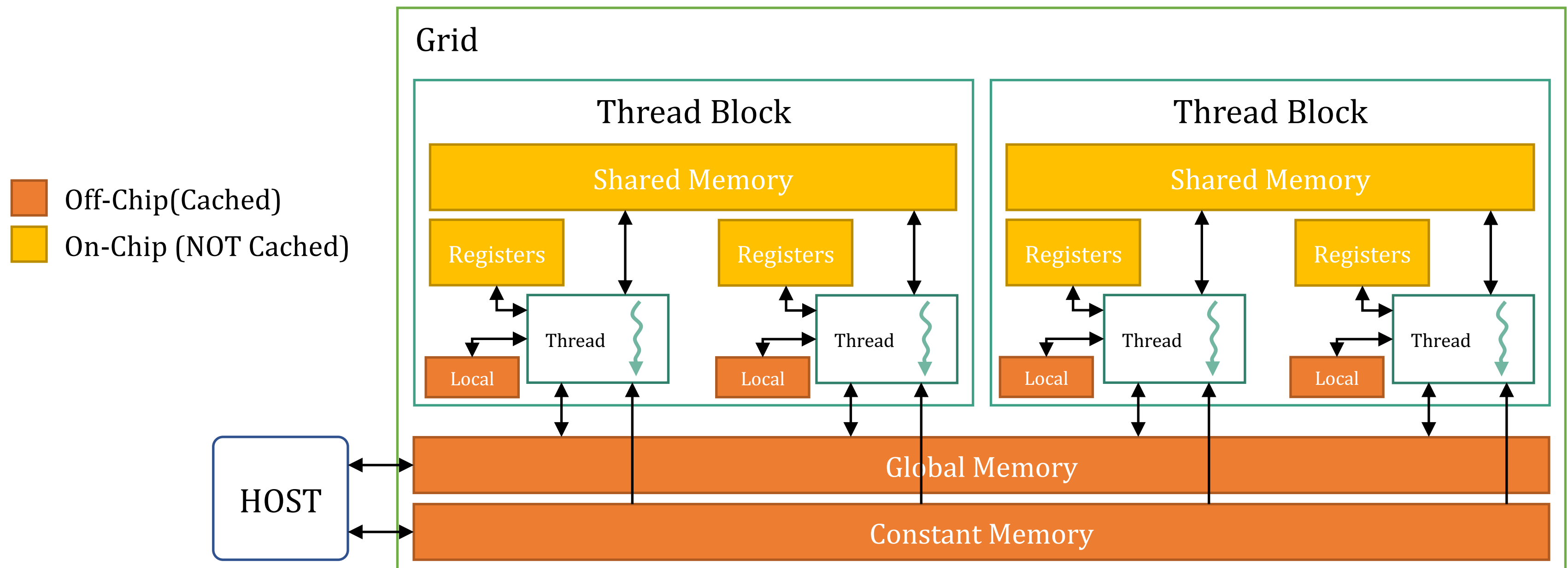
- Allocation of host memory
- Allocation of GPU memory
- Copy from host to GPU
- Kernel config & invocation
- Copy from GPU to host
- Deallocation of GPU memory
- Deallocation of host memory

```
// Host code
int main() {
  int T = 24;
  size_t size = T * sizeof(float);

  // Allocate input vectors h_A and h_B in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);

  // Initialize input vectors
  ...

  // Allocate vectors in device memory
  float* d_A; cudaMalloc(&d_A, size);
  float* d_B; cudaMalloc(&d_B, size);
  float* d_C; cudaMalloc(&d_C, size);

  // Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // Invoke kernel
  int threadsPerBlock = 12;
  int blocksPerGrid = 2;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, T);

  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

  // Free host memory
  ...
}
```

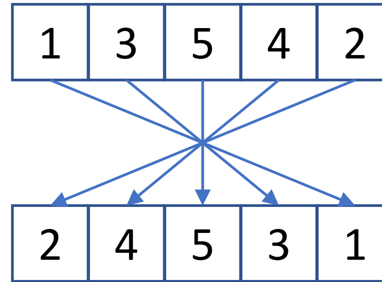
The GPU memory hierarchy

Threads may access data from multiple memory spaces during their execution

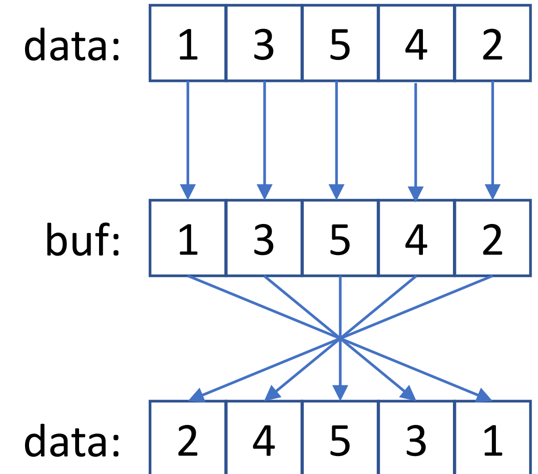
- A thread has access to its **registers** and **local memory**
- Threads in the same thread block have acess to **shared memory**
- All the thread in the GPU have access to **global and constant memories**



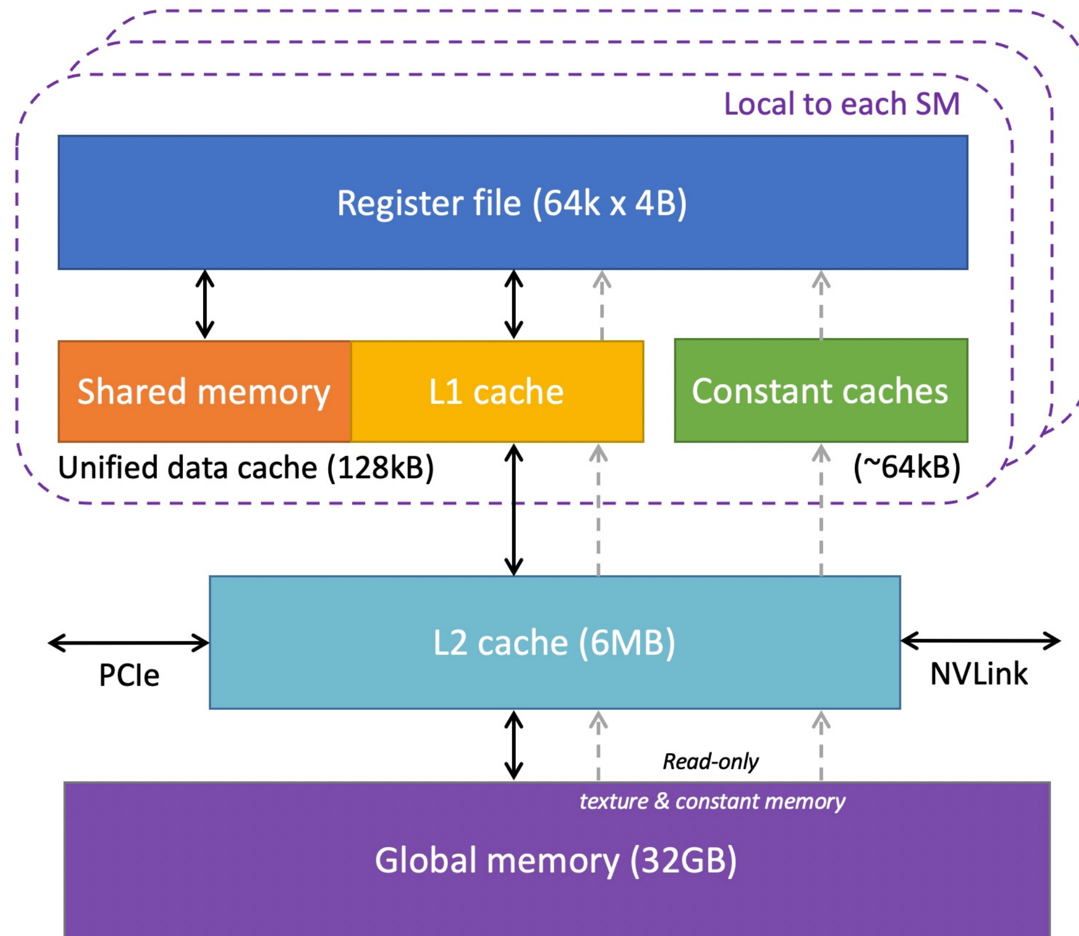
Shared memory example




```
// Kernel definition
__global__ void Reverse (int* data, int n) {
    __shared__ int buf[64];
    int i = threadIdx.x;
    int k = n - i - 1;
    buf[i] = data[i];

    __syncthreads();

    data[i] = buff[k];
}
```



Memory hierarchy in the NVIDIA V100



Memory hierarchy (Summary)

Memory	Access	Scope	Lifetime	Speed	Note
Global	RW	All threads and CPU	All	Slow, cached	Large
Constant	R	All threads and CPU	All	Slow, cached	Broadcasting data in a warp
Texture	R	All threads and CPU	All	Slow, cached	2D spatial locality
Local	RW	Per thread	Thread	Slow, cached	Register spilling, kernel alloc.
Shared	RW	Per block	Block	Fast	Fast communic. between threads
Registers	RW	Per thread	Thread	Fast	Limited

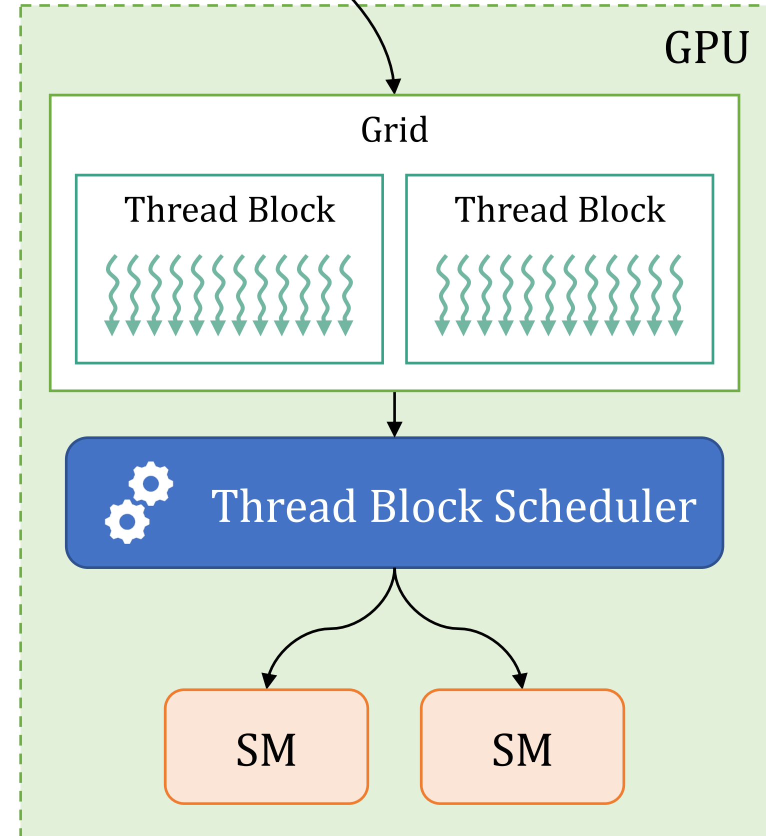
Scheduling thread blocks

Kernel invocation

CPU

```
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(A, B, C);
```

The Thread Block Scheduler maps all the thread blocks to the available SMs



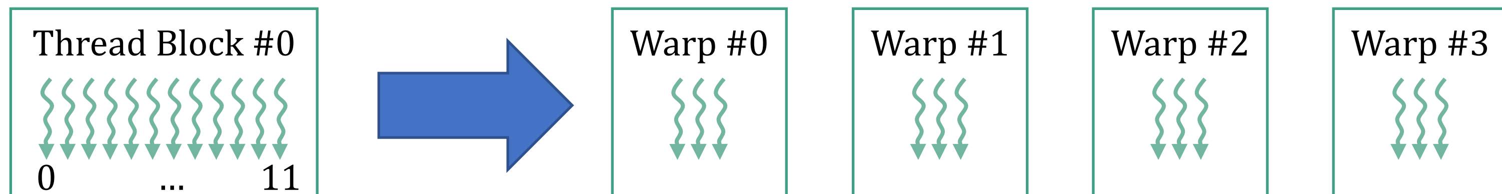
What is a warp?

A warp is essentially a SIMD operation formed by hardware

Threads are not executed independently; they are executed in a collective of (32) consecutive threads, referred to as a warp

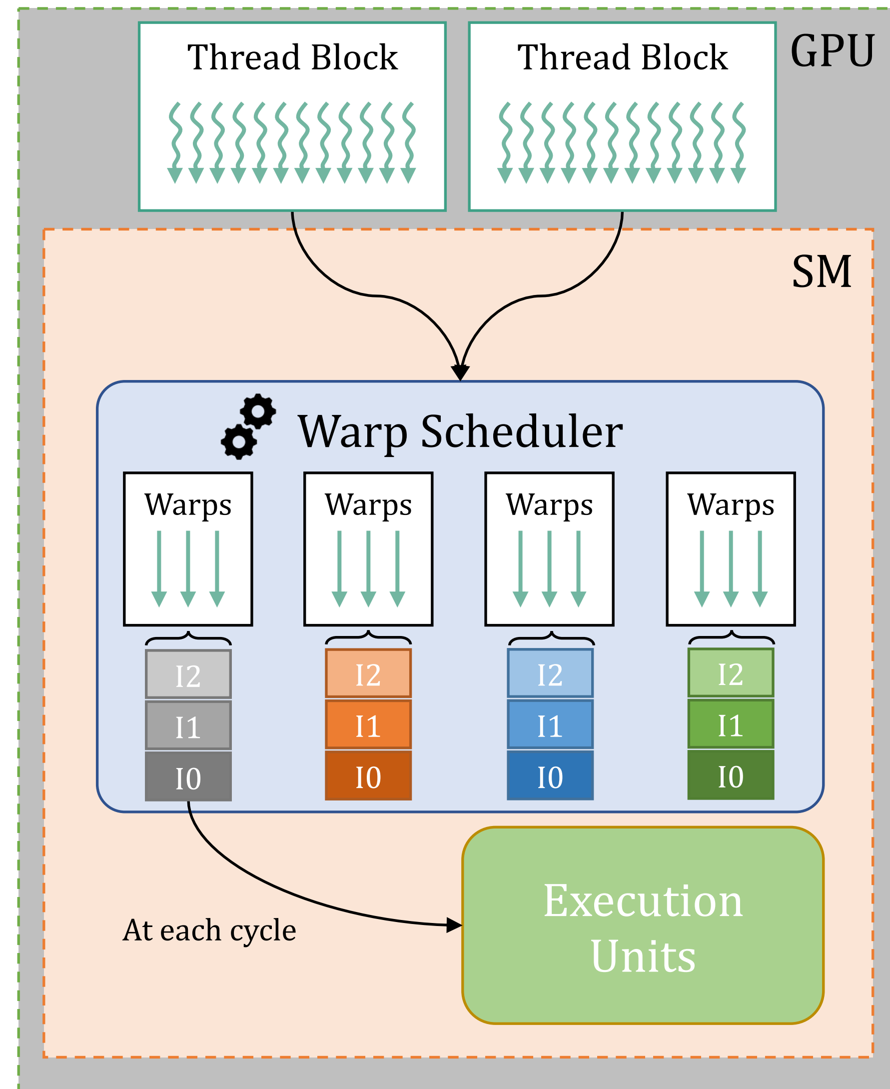
A GPU is designed to execute warps:

- A warp is the GPU execution unit

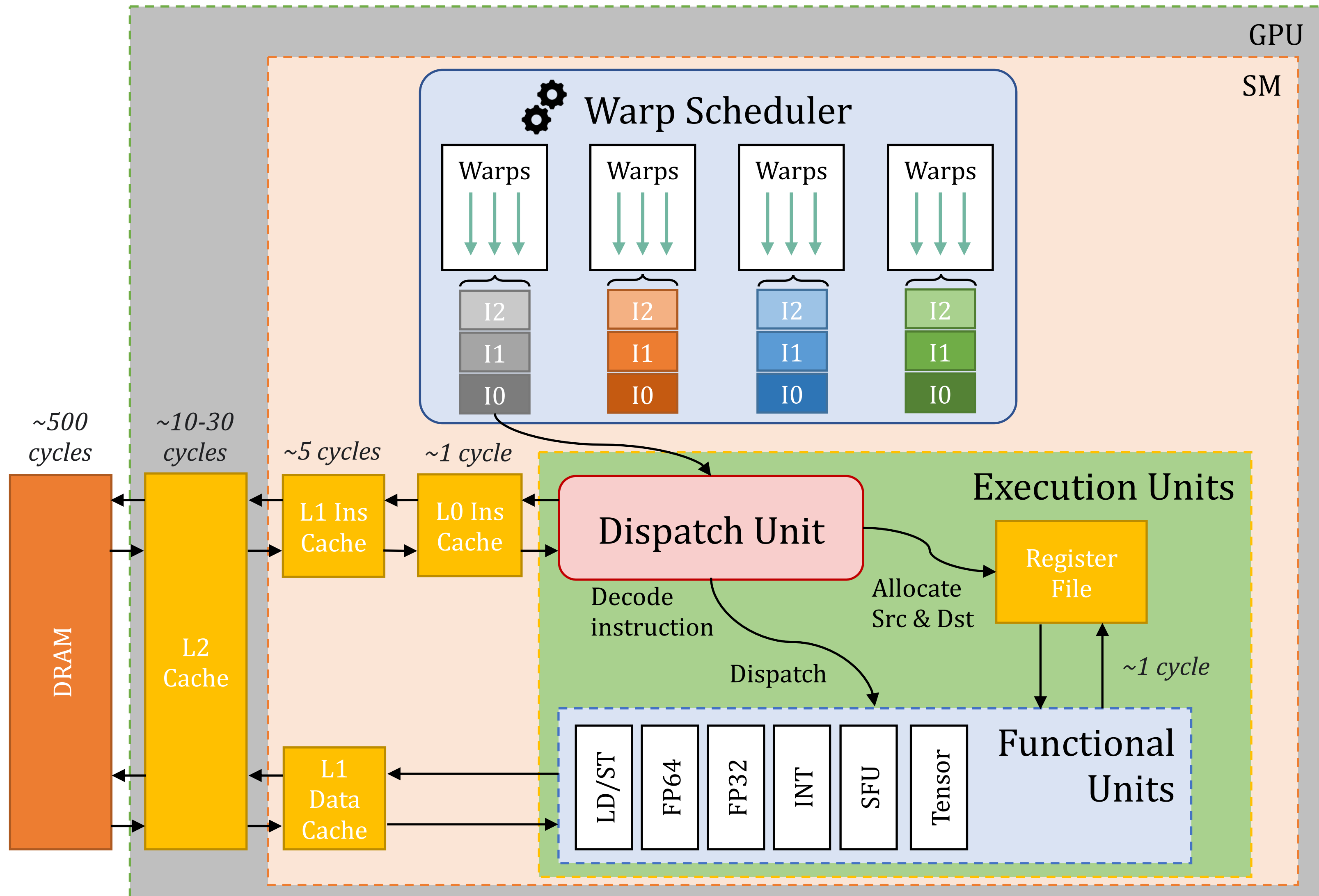


SMs and warps

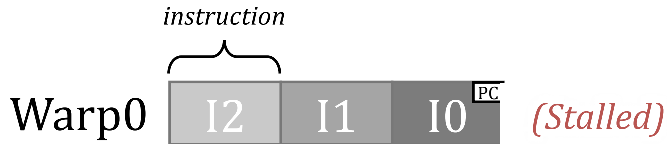
- Thread blocks issued to an SM are sent to **warp schedulers**
- Warp schedulers divide the thread blocks into **warps**
  - The warp size is **NOT user configurable**
- A warp includes a list of **instructions that are shared across multiple (32) threads**
- At every cycle, the warp schedulers **issue the next instruction of a warp** to the corresponding execution units



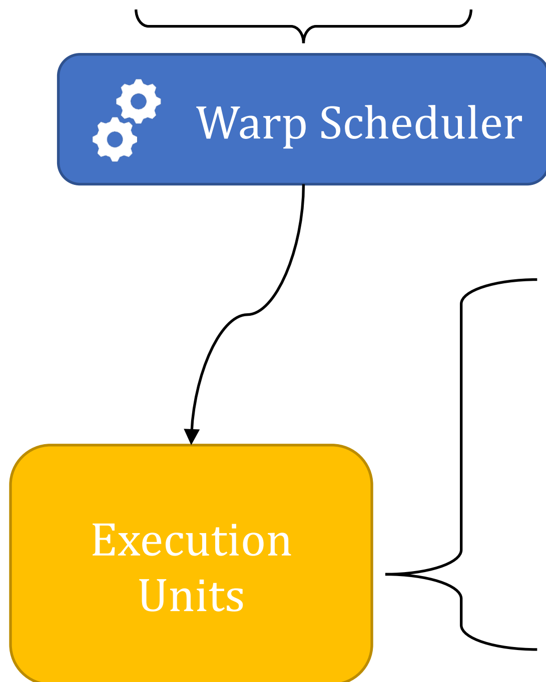
SM execution cycle



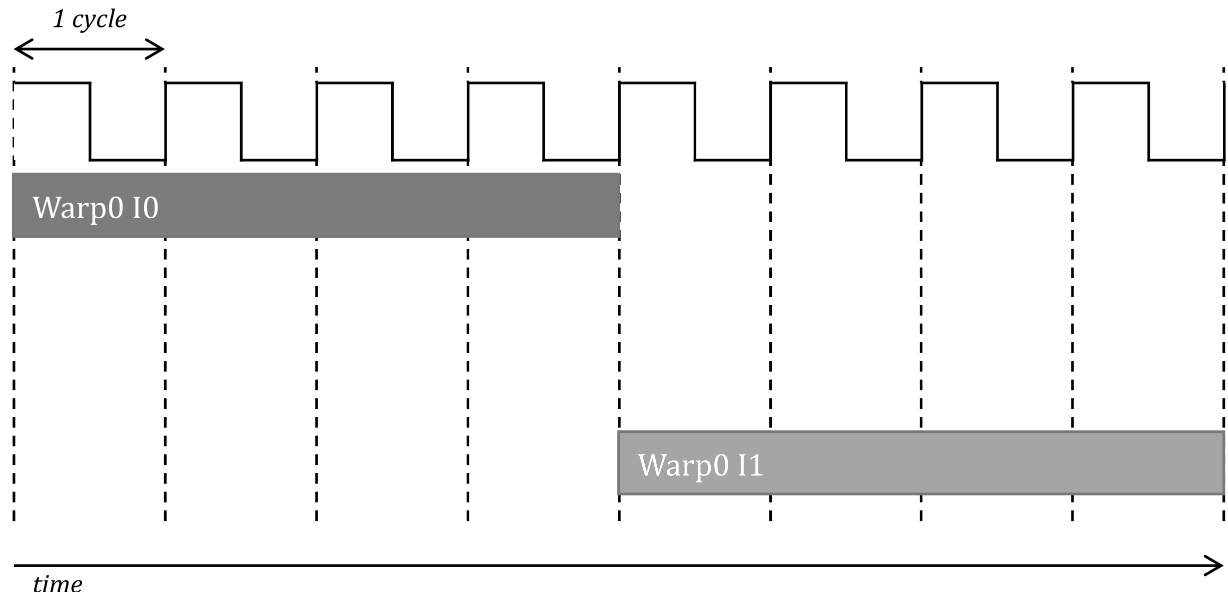
Warp scheduling example



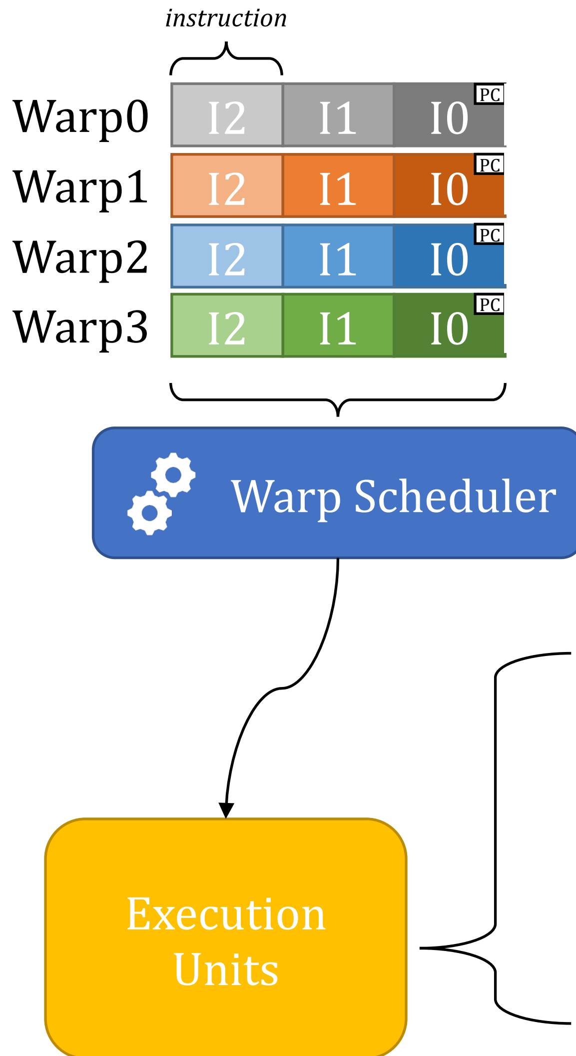
- GPUs include “simple” throughput-oriented in-order pipelines
  - No data forwarding, branch prediction, etc.



- GPUs experience long latency instruction execution

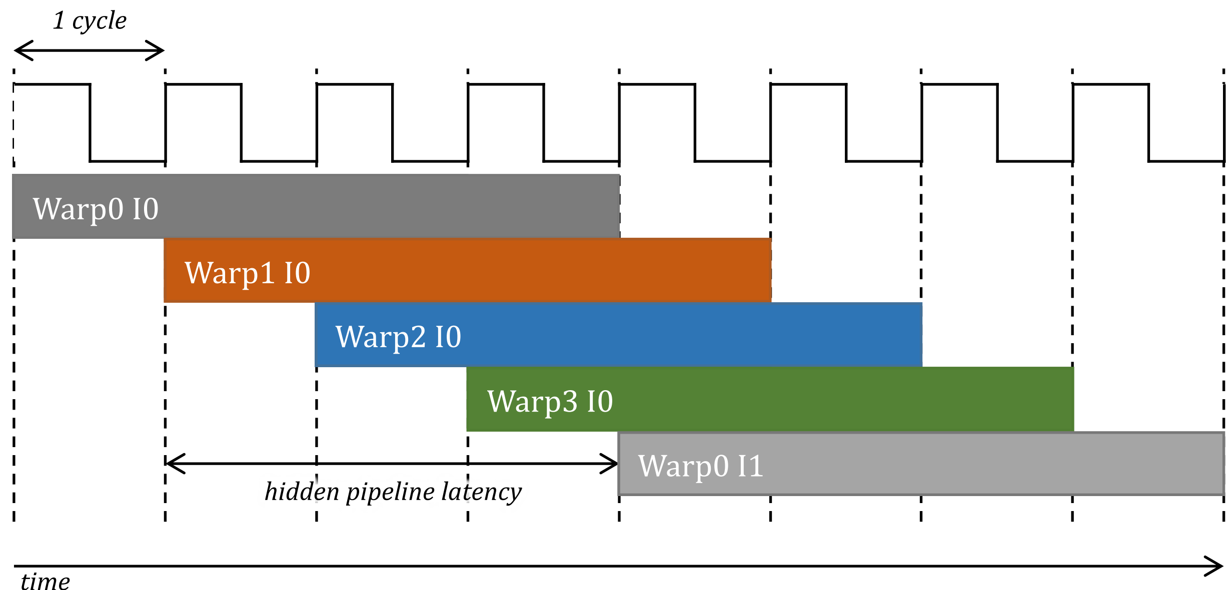


Fine grained multithreading



- Interleaved execution of multiple warps skipping stalled warps

- Fine grained multithreading enables long latency tolerance by running multiple independent warps



Warp scheduling

Limited by the maximum warps supported *(device dependent)*

Device Limit

Can be stalled by:

- An instruction fetch
- A memory dependency (result of memory instruction)
- An execution dependency (result of previous instruction)
- A synchronization barrier

Limited by the kernel invocation configuration *(user-defined)*

```
VecAdd<<<...,...>>>(A, B, C);
```

Theorical Occupancy

each cycle

Active Warps

Stalled Warps

Eligible Warps

Limited by the internal load balancing of the GPU *(thread block scheduler)*

Limited by the GPU core's capabilities

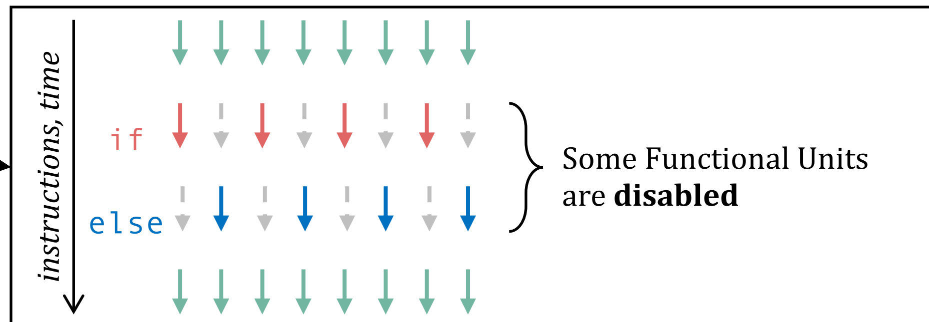
Selected Warps

Warp divergence

Threads in a warp can **diverge** because of **branching**

Simple example: condition on thread ID

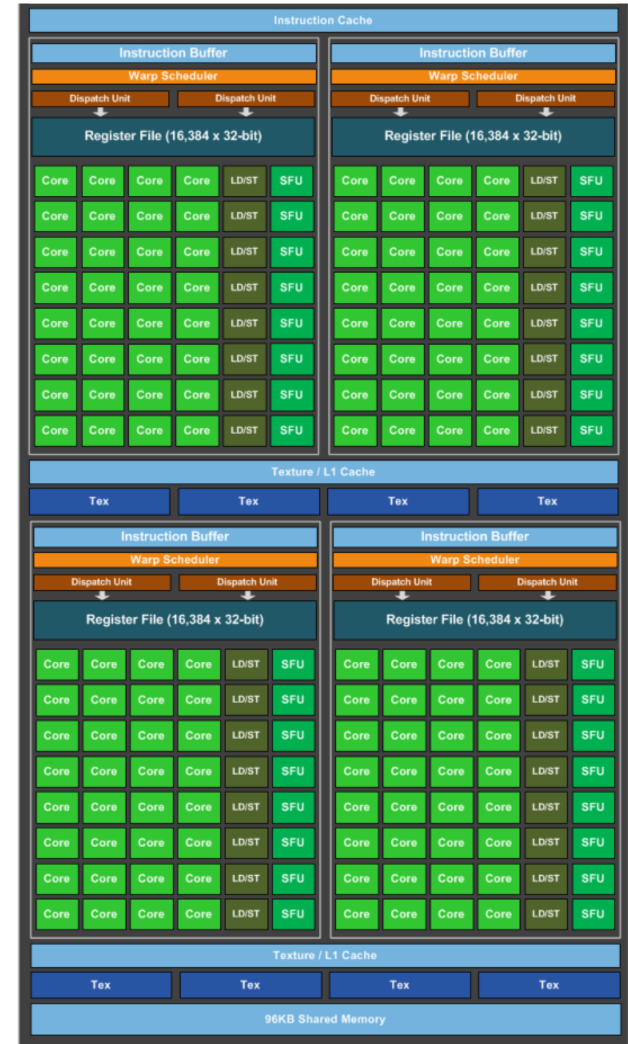
```
// Kernel definition
__global__ void VecOp(float* A, float* B, float* C) {
  int tid = threadIdx.x;

  if (tid % 2 == 0)
  {
    C[i] = A[i] + 1 * B[i];
  }
  else
  {
    C[i] = A[i] + 2 * B[i]
  }
}
```



The NVIDIA Jetson Nano (2019)

NVIDIA family	Maxwell
Compute capability	5.3
# Streaming Multiprocessors	1
# CUDA cores	128
Max clock rate	922 MHz
L2 cache size	256 kB
Warps dispatched per cycle	2x4
Warp size	32

TSMC 22nm



Max raw performance of Jetson Nano GPU

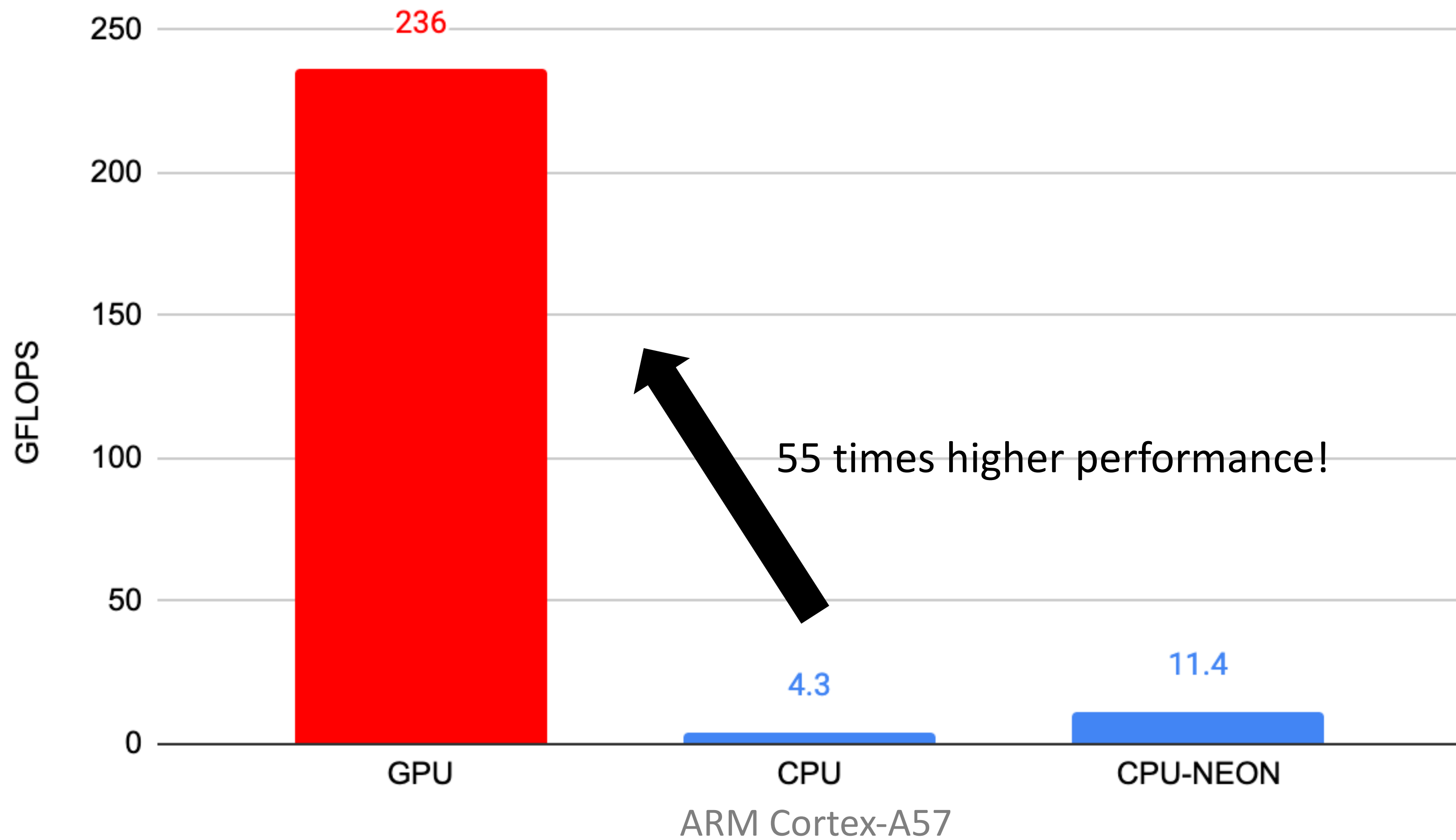
Max. perf = WarpSize x WarpsDispPerCycle x Freq

= 32 x 8 x 922M

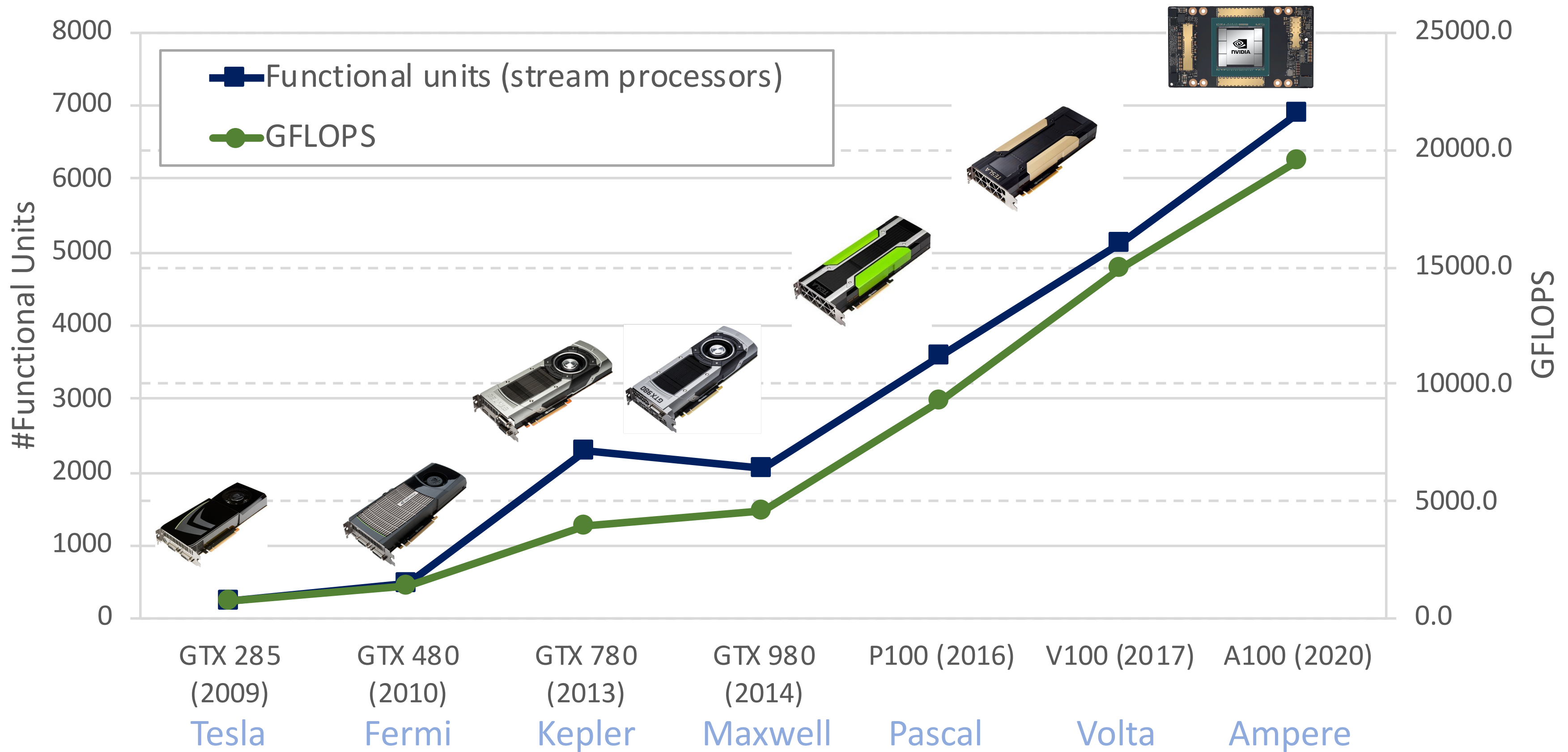
= 236 GFLOPS* (FP32)

* GFLOPS = Giga Floating Point Operations Per Second

Theoretical performance comparison

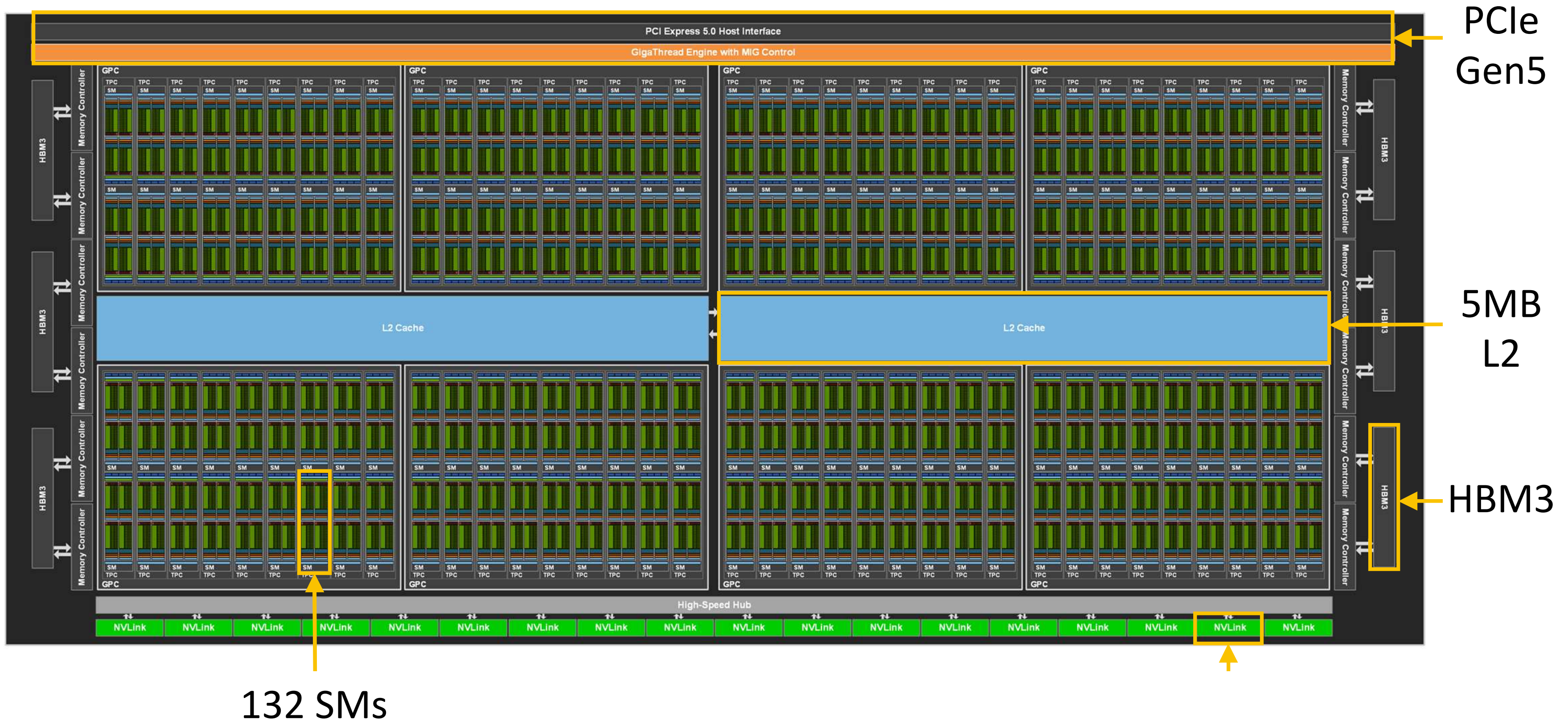


Evolution of NVIDIA GPUs



NVIDIA H100 (latest GPU, 2022)

80B Transistors, TSMC 4N (5nm)

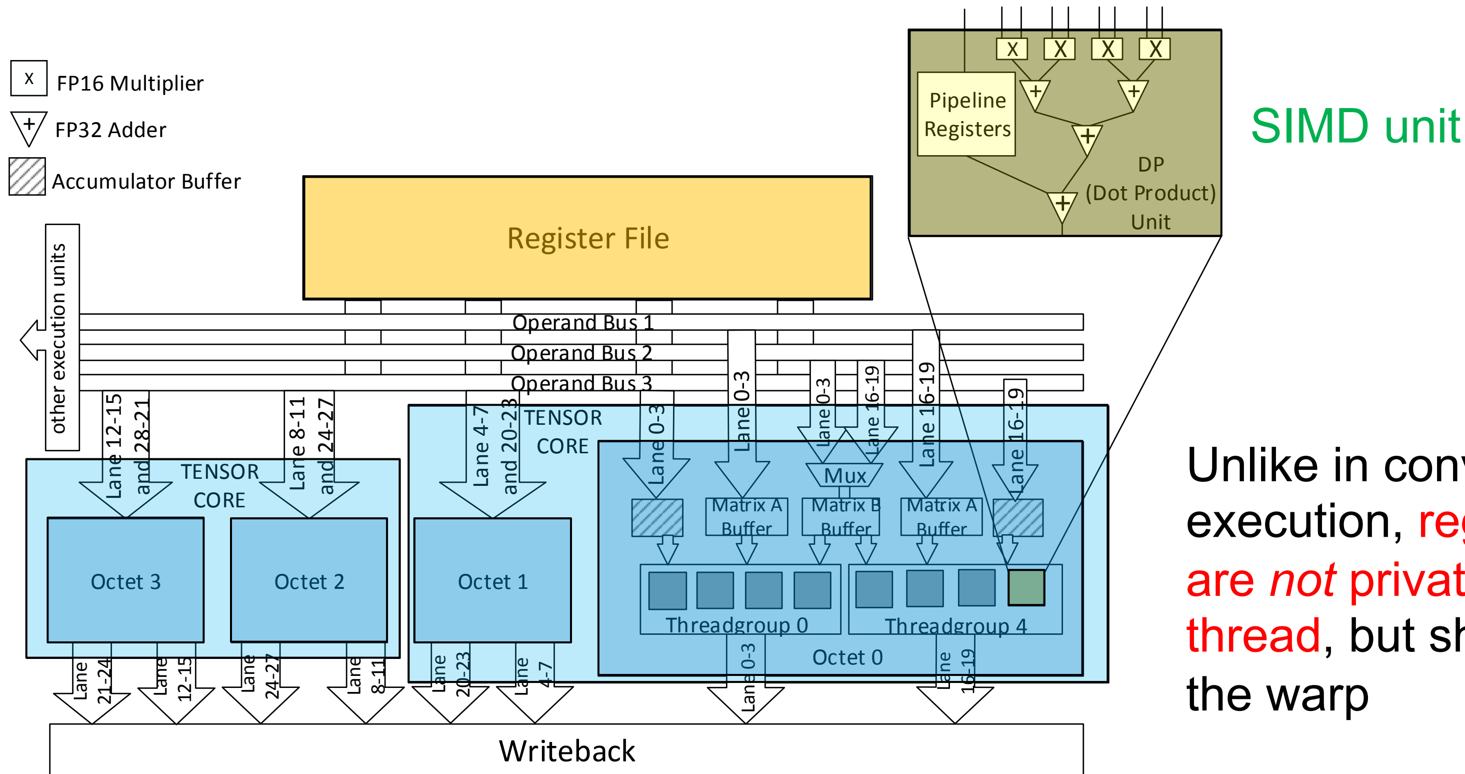


H100 SM architecture

FP32 cores/SM	128
FP64 cores/SM	64
INT32 cores/SM	64
Tensor cores/SM	4
Max clock rate	1.755 GHz
L1 cache size	256 kB
Warps dispatched per cycle	4

Tensor core microarchitecture (Volta)

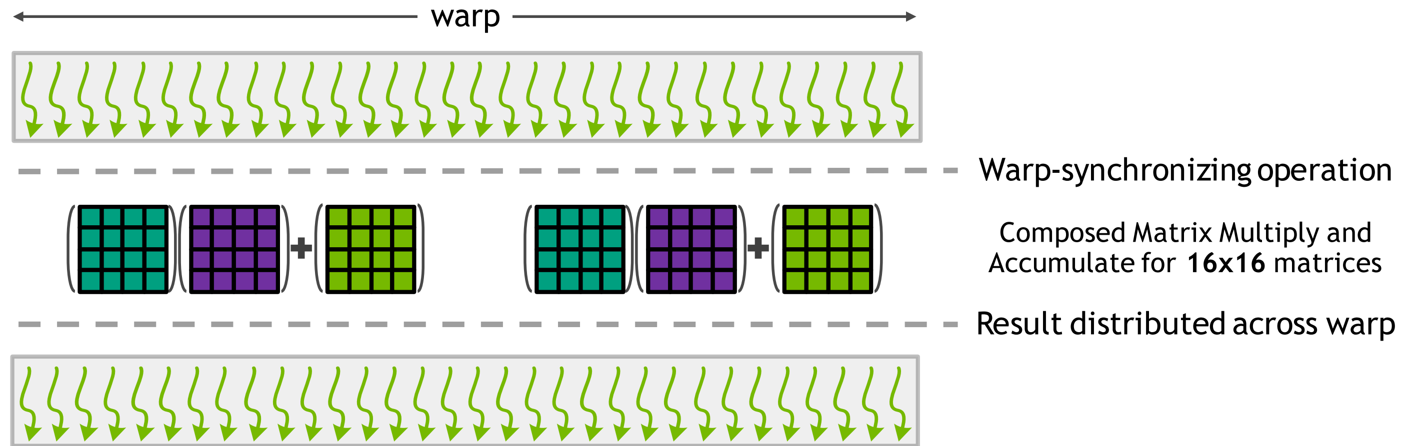
- Each warp utilizes two tensor cores
- Each tensor core contains two “octets”
  - 16 SIMD units per tensor core (8 per octet)
  - 4x4 matrix-multiply and accumulate each cycle per tensor core



Unlike in conventional GPU execution, register contents are *not* private to each thread, but shared inside the warp

M. A. Raihan et al."Modeling Deep Learning Accelerator Enabled GPUs." ISPASS. 2019.

Tensor core example



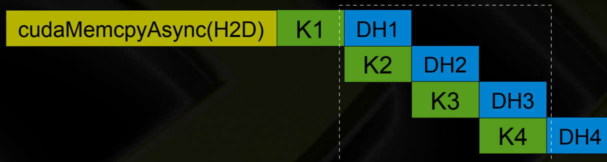

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++
Warp-Level Matrix Operations

CUDA streams and concurrency

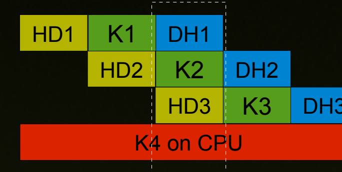
- **Serial** (1x)

- **2-way concurrency** (up to 2x)

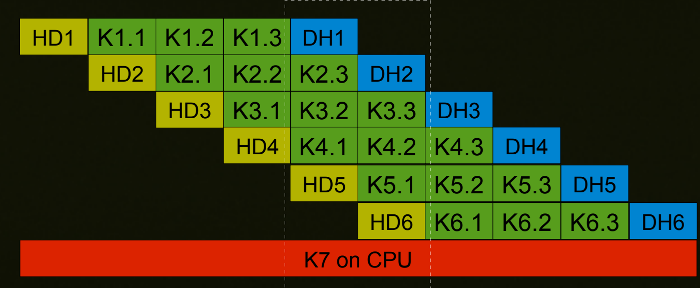


- **3-way concurrency** (up to 3x)

- **4-way concurrency** (3x+)



- **4+ way concurrency**

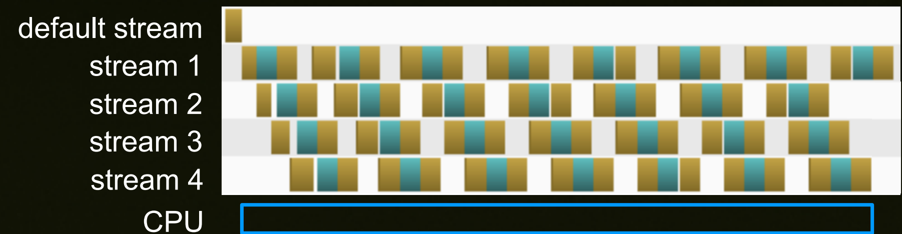


Streams on a real example

- **CPU** (4core Westmere x5670 @2.93 GHz, MKL)
  - **43 Gflops**
- **GPU** (C2070)
  - **Serial : 125 Gflops (2.9x)**
  - **2-way : 177 Gflops (4.1x)**
  - **3-way : 262 Gfllops (6.1x)**
- **GPU + CPU**
  - **4-way con.: 282 Gflops (6.6x)**
  - **Up to 330 Gflops for larger rank**

DGEMM: m=n=8192, k=288

Nvidia Visual Profiler (nvvp)

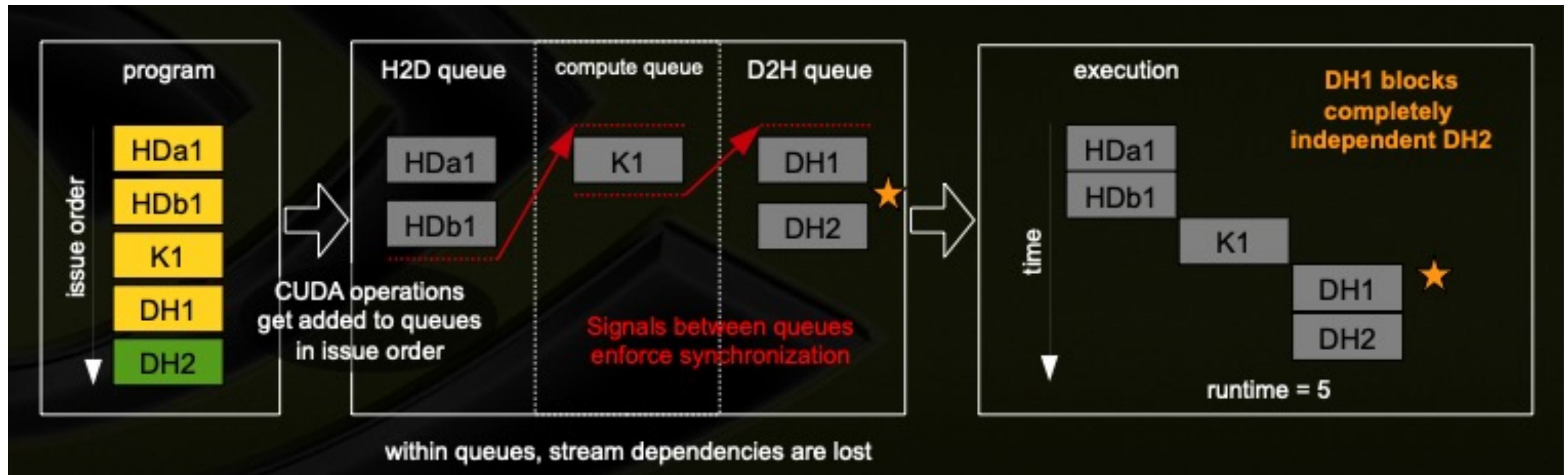


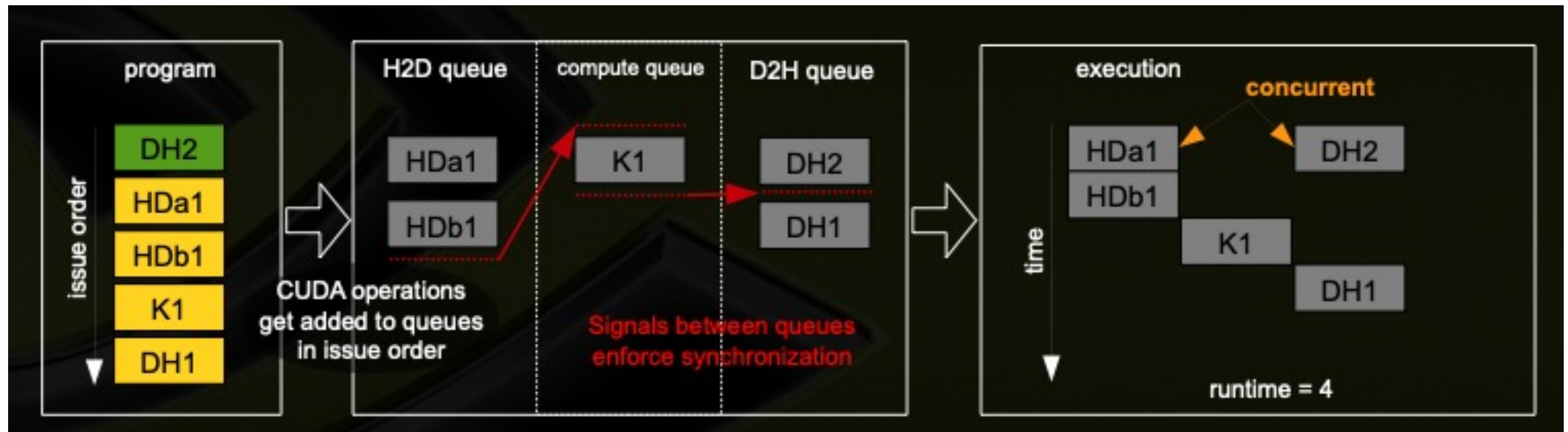
Use of streams in CUDA

```
…
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, …  ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( …, dev3, …  ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
```

potentially overlapped

Issue order matters





Exploiting high performance is not easy

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

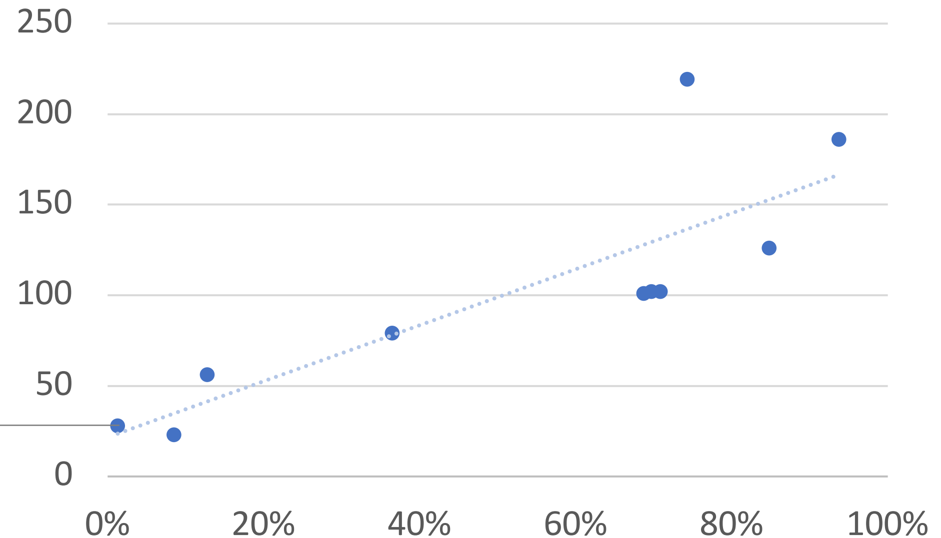
1.3% of the performance of the sgemm kernel in the CUDA Basic Linear Algebra Subroutine (cuBLAS) library

Exploiting high performance is not easy

GPUs promise high raw performance but…

Only **complex programs** can use GPU's computational power efficiently

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                            const float *B, float beta, float *C) {
  // compute position in C that this thread is responsible for
  const uint x = blockIdx.x * blockDim.x + threadIdx.x;
  const uint y = blockIdx.y * blockDim.y + threadIdx.y;

  // `if` condition is necessary for when M or N aren't multiples of 32.
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y];
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```



Performance relative to the NVIDIA library (cuBLAS)

https://siboehm.com/articles/22/CUDA-MMM

Performance tuning

For optimal performance, the programmer has to juggle

- Finding enough parallelism to use all SMs
- Finding enough parallelism to keep all cores in an SM busy
- Optimizing use of registers and shared memory
  - E.g., Avoid bank conflicts
- Optimizing device memory access for contiguous memory
  - E.g., Use shared memory to improve data reuse
- Minimizing bandwidth requirements
- Maximizing streams cuncurrency
- Minimizing warp divergence
- Minimizing thread synchronization
- Minimizing the number of registers used per thread
  - Avoid register spilling
- Etc.

Outline

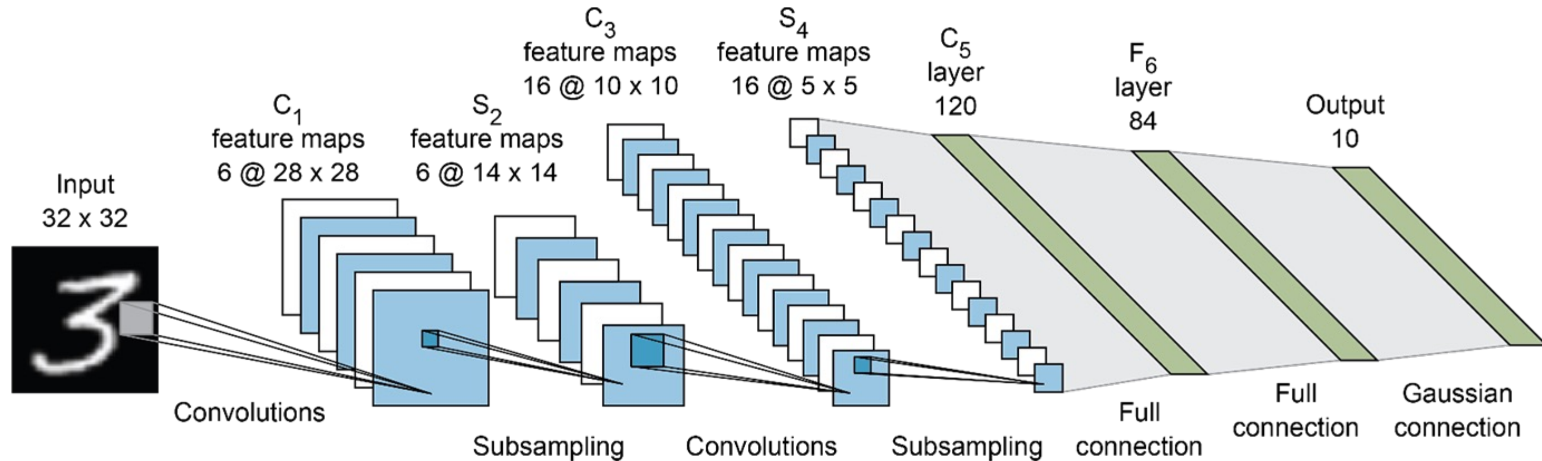
1. What is a GPU?

2. How ML frameworks use GPUs

The ML compilation problem

- Application designers write ML applications in high-level languages (mostly Python)
  - Optimized for productivity
  - Implementation agnostic

LeNet-5 in 9 lines of code

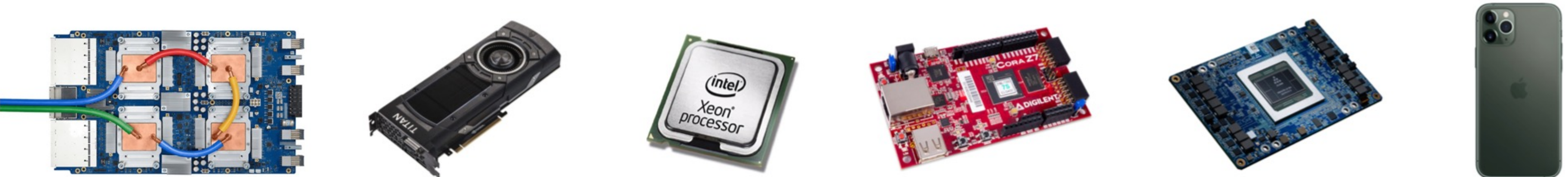



```
model = keras.Sequential()

model.add(layers.Conv2D(filters=6, kernel_size=(3,3), activation='relu', input_shape=(32,32,1)))
model.add(layers.AveragePooling2D())

model.add(layers.Conv2D(filters=16, kernel_size=(3,3), activation='relu'))
model.add(layers.AveragePooling2D())

model.add(layers.Flatten())

model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation = 'softmax'))
```

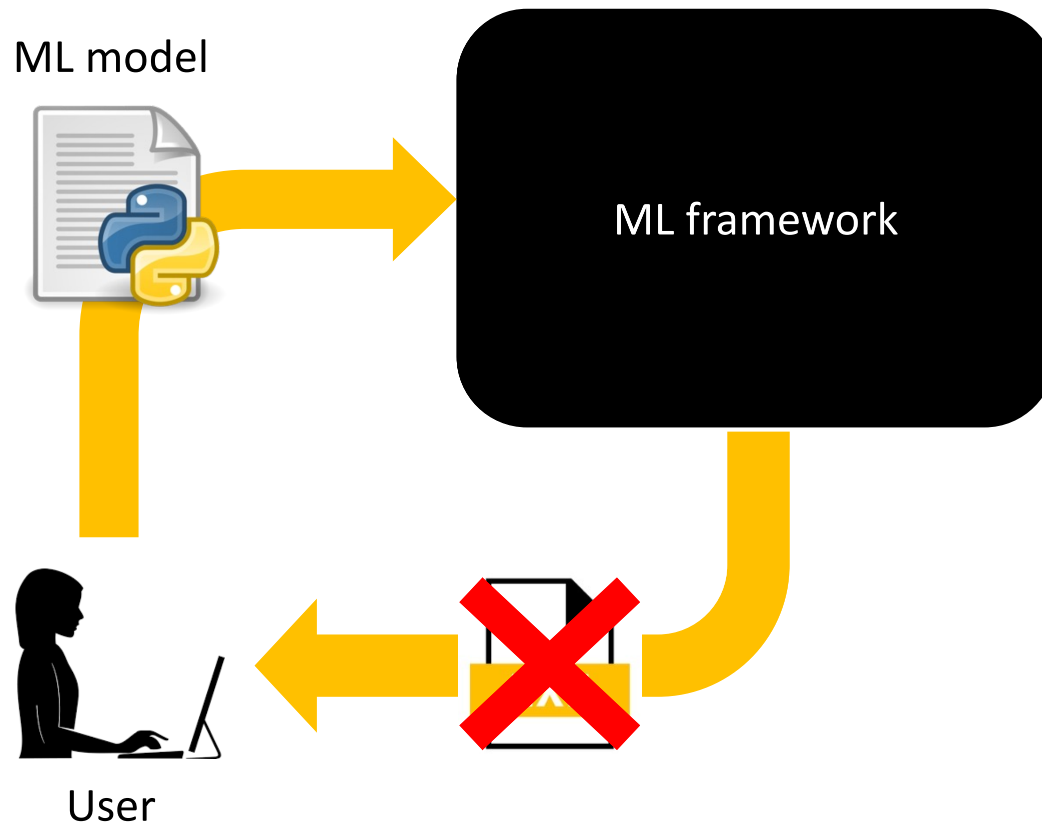
The ML compilation problem

- Application designers write ML applications in high-level languages (mostly Python)
  - Optimized for productivity
  - Implementation agnostic

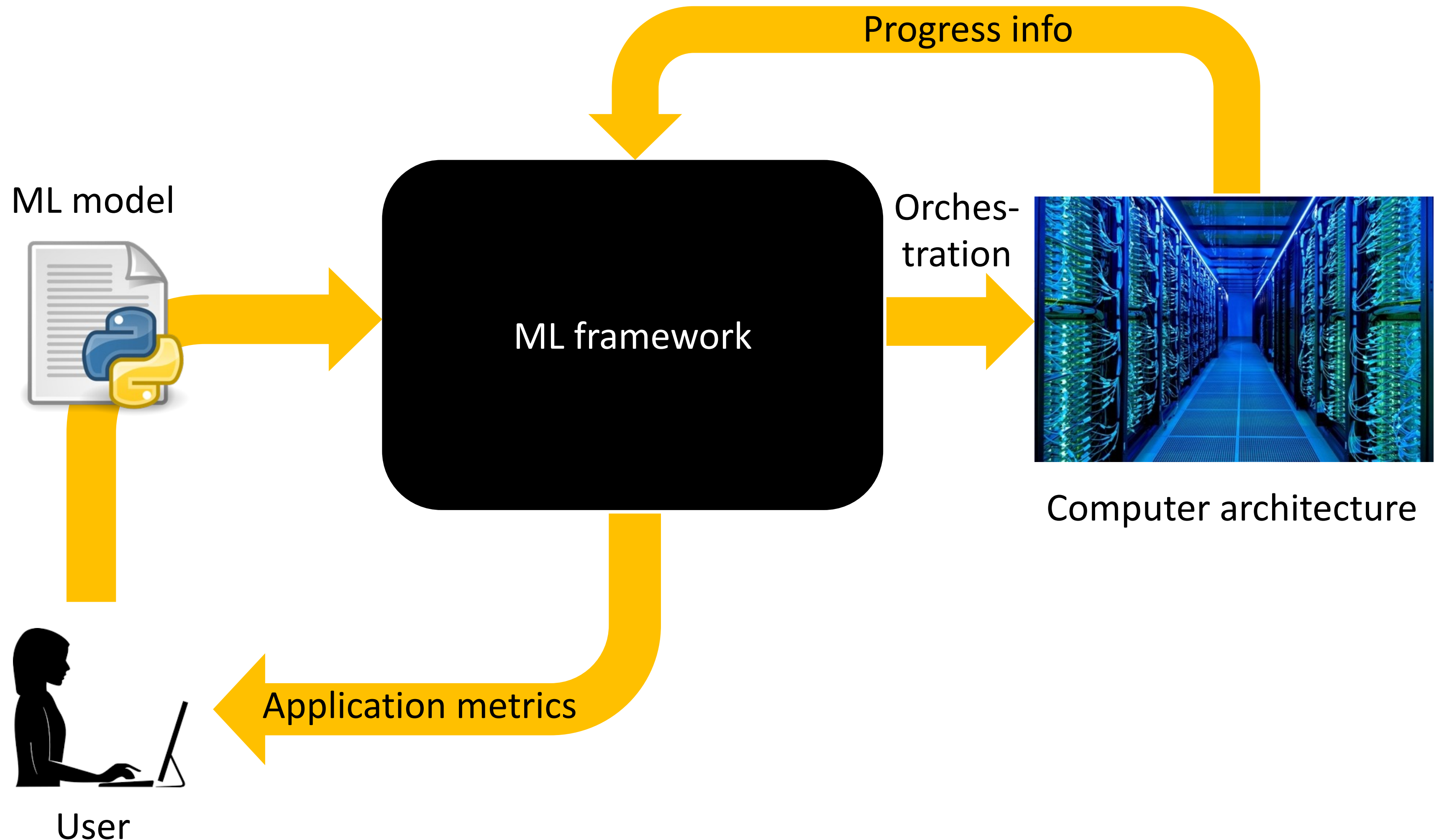
- The same code has to run efficiently across a plethora of hardware platforms



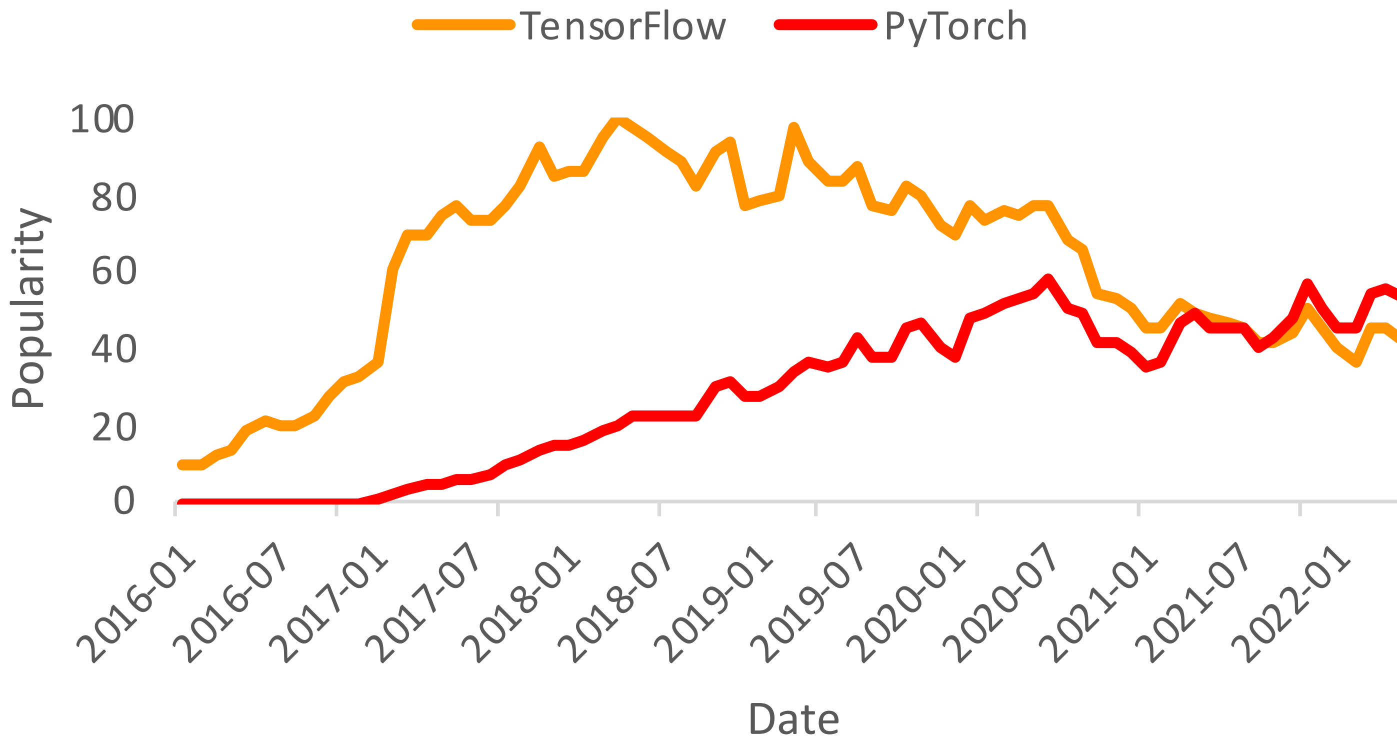
An ML framework



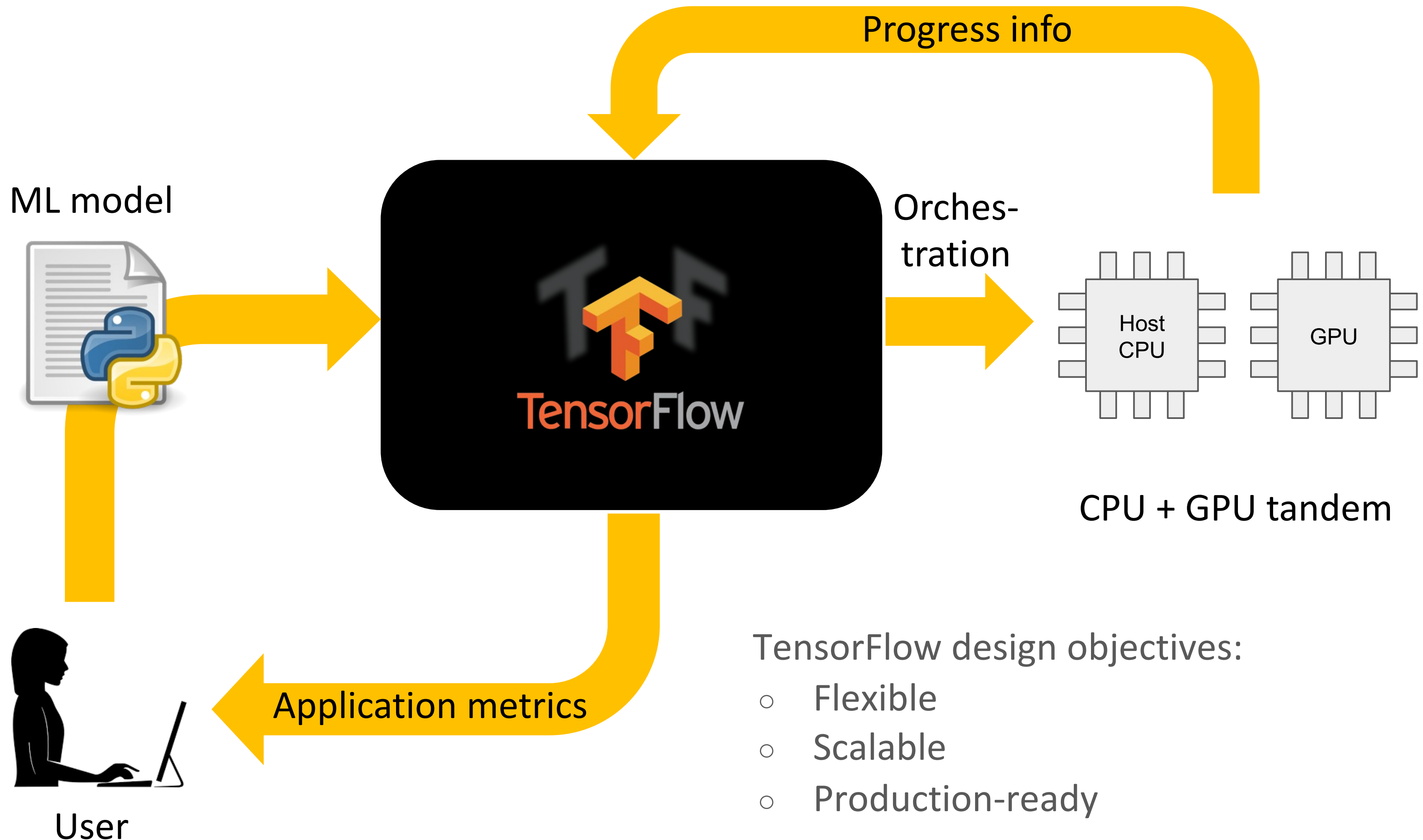
An ML framework



Most popular ML frameworks



TF as a representative ML framework



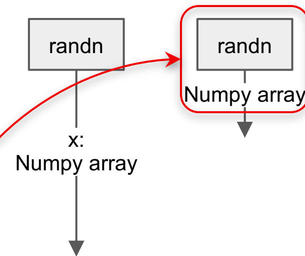
A TensorFlow example program

```python
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
   x = np.random.randn(1024,1024)
   y = tf.constant(np.random.randn(1024,1024))

   z = tf.matmul(x, y, transpose_b=True)

   s = tf.nn.relu(z)

   print(s)
```

A TensorFlow example program

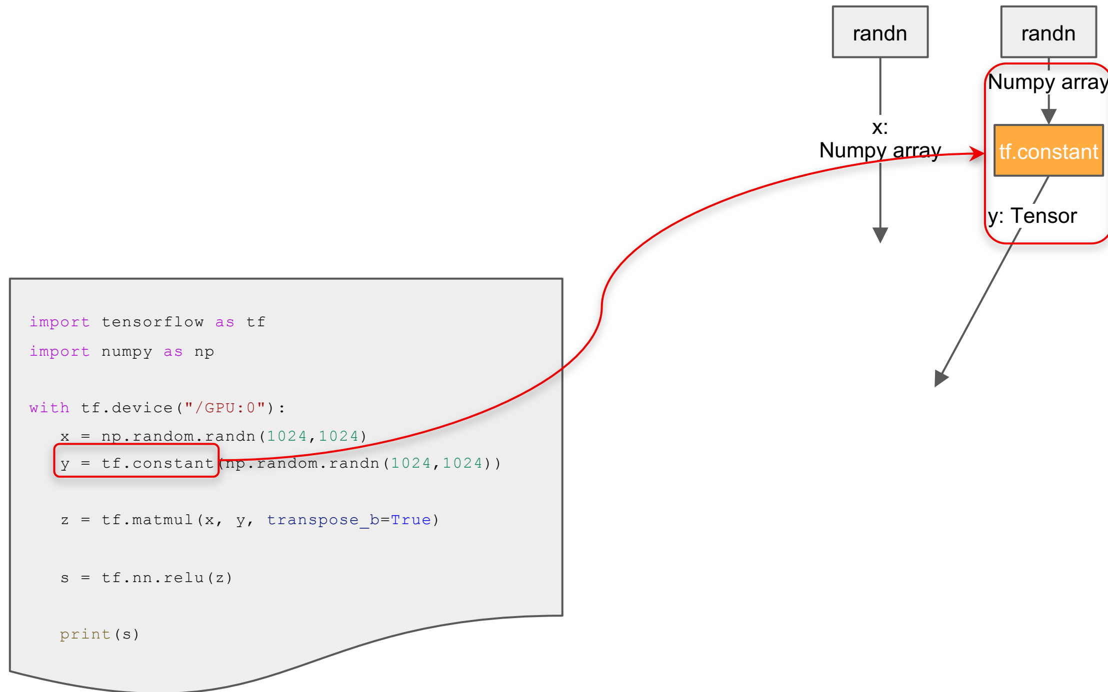
```
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
   x = np.random.randn(1024,1024)
   y = tf.constant(np.random.randn(1024,1024))

   z = tf.matmul(x, y, transpose_b=True)

   s = tf.nn.relu(z)

   print(s)
```

A TensorFlow example program



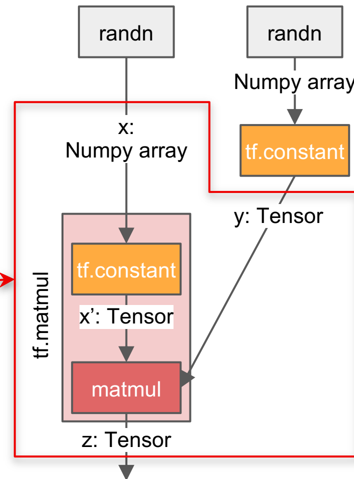

```
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
    x = np.random.randn(1024,1024)
    y = tf.constant(np.random.randn(1024,1024))

    z = tf.matmul(x, y, transpose_b=True)

    s = tf.nn.relu(z)

    print(s)
```

A TensorFlow example program



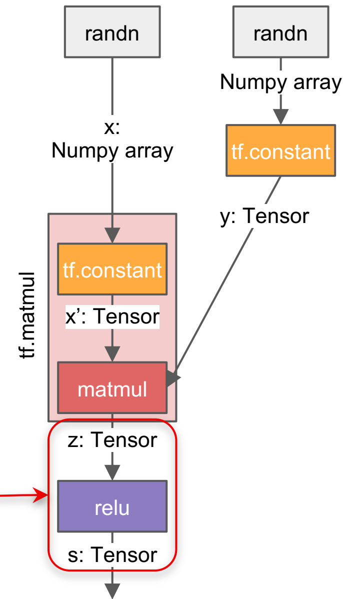
A TensorFlow example program

```
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
    x = np.random.randn(1024,1024)
    y = tf.constant(np.random.randn(1024,1024))

    z = tf.matmul(x, y, transpose_b=True)

    s = tf.nn.relu(z)

    print(s)
```



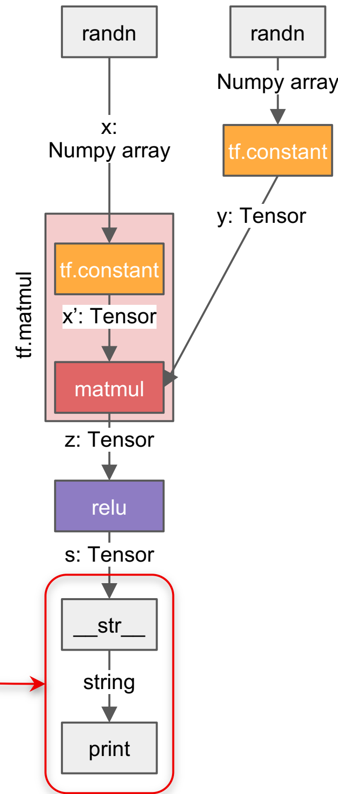
A TensorFlow example program

```python
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
    x = np.random.randn(1024,1024)
    y = tf.constant(np.random.randn(1024,1024))

    z = tf.matmul(x, y, transpose_b=True)

    s = tf.nn.relu(z)

    print(s)
```

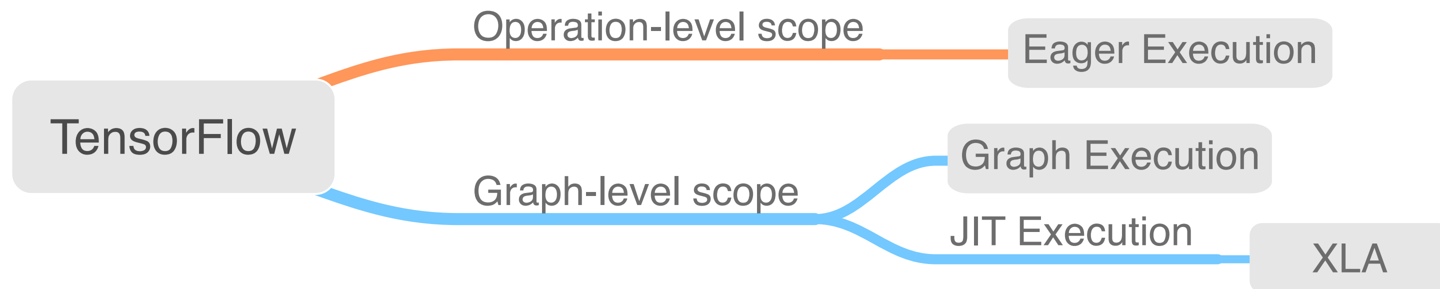


A TensorFlow example program

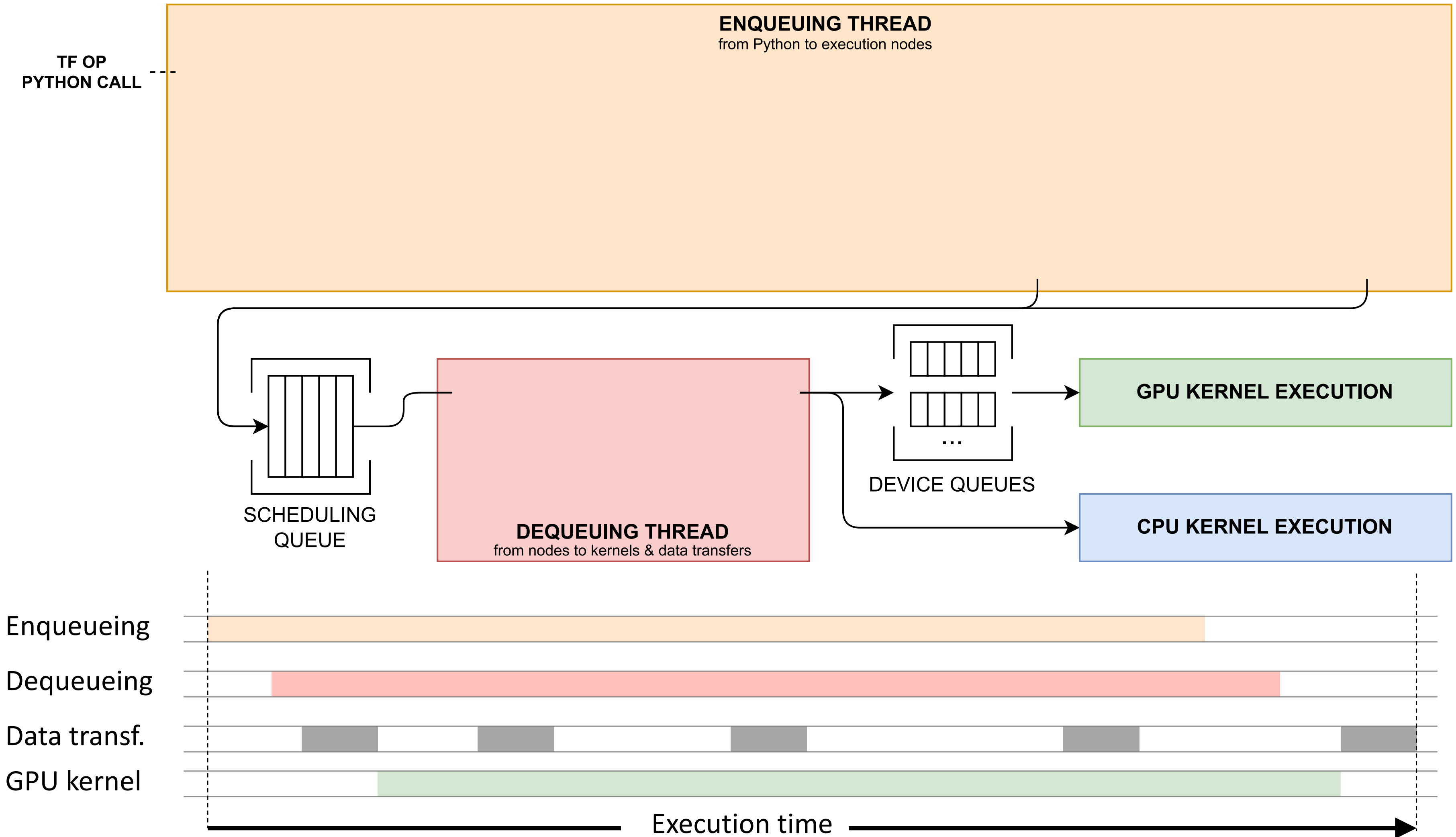
```
import tensorflow as tf
import numpy as np

with tf.device("/GPU:0"):
   x = np.random.randn(1024,1024)
   y = tf.constant(np.random.randn(1024,1024))

   z = tf.matmul(x, y, transpose_b=True)

   s = tf.nn.relu(z)

   print(s)
```



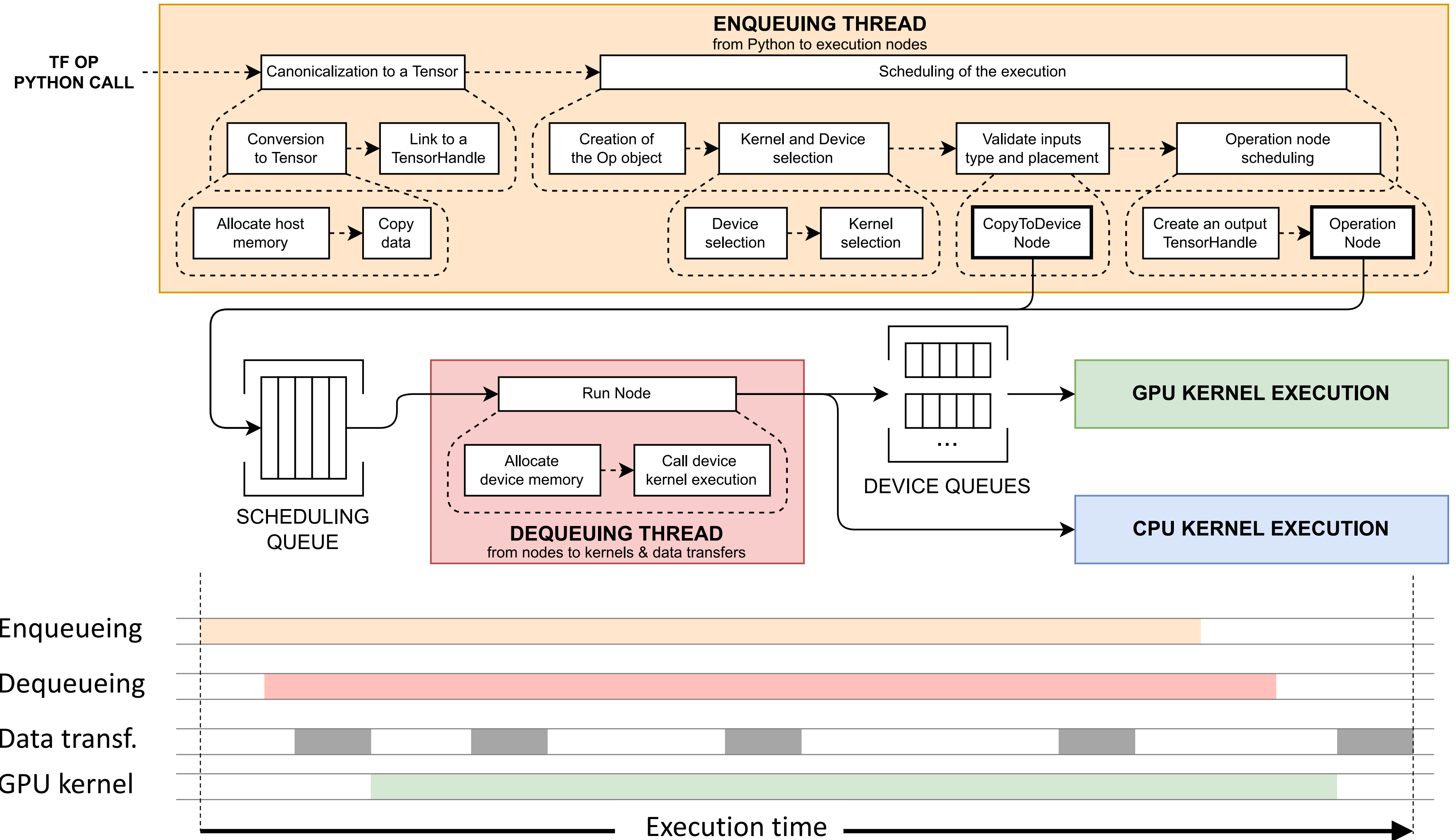
TensorFlow execution modes



Eager Execution: internal architecture



Eager Execution: internal architecture



TF execution: Enqueueing thread

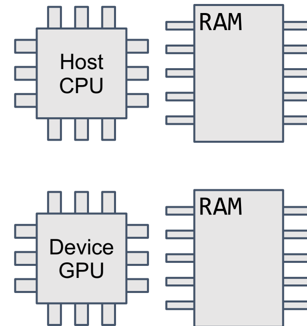
1 Context initialization

line operations

context

```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```

context lifetime



- Initialized when starting the Python program, and destroyed when program ends
- Stores details about the execution (mode of exec., **target device**, etc.)
- Thread-local ⇒ allows for multiple TF Python programs in parallel

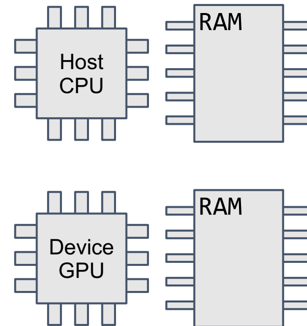
TF execution: Enqueueing thread

1	3
Context initialization	Set Device

line operations

context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



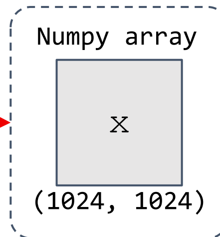
TF execution: Enqueueing thread

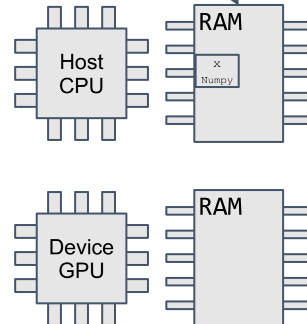
1	3	4
Context initialization	Set Device	malloc x (numpy)

line operations

context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```





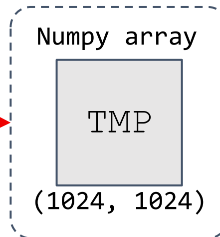
TF execution: Enqueueing thread

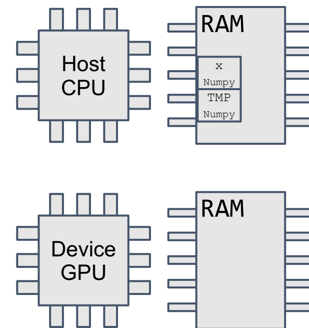
1	3	4	5
Context initialization	Set Device	malloc x (numpy)	malloc y (numpy)

line operations

context device=GPU0

```python
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

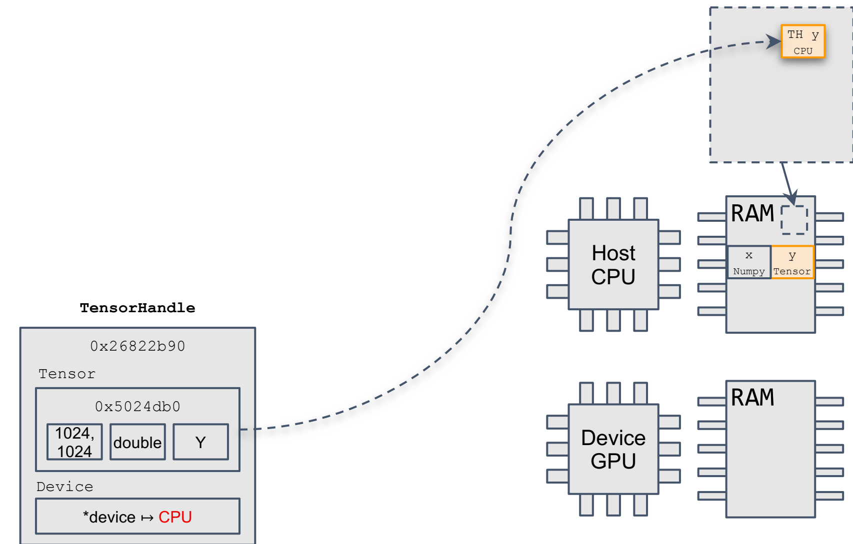




TF execution: Enqueueing thread

context device=GPU0

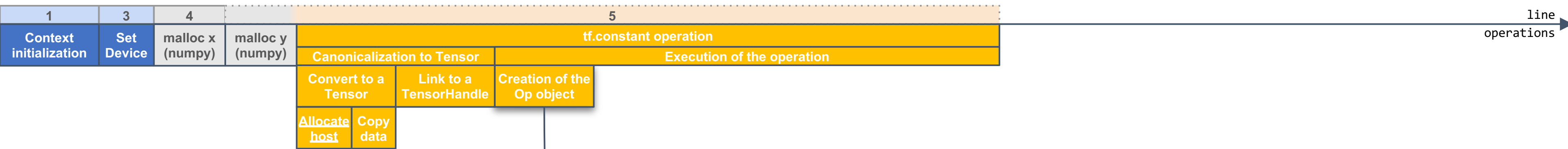
```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```



TensorHandle

- Represents a tensor (or a not-yet-computed FutureTensor) which lives (or will live) on a device
- Allows the runtime to race ahead and continue parsing Python operations

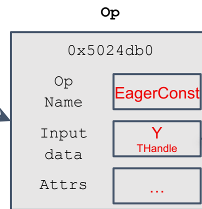
TF execution: Enqueueing thread

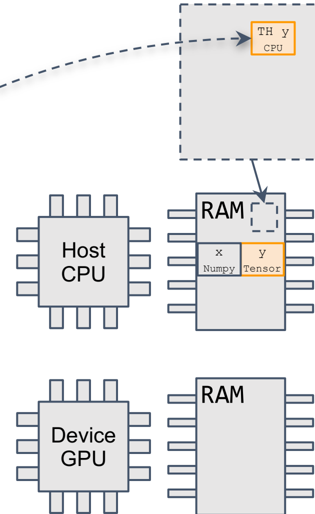


context device=GPU0

```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```

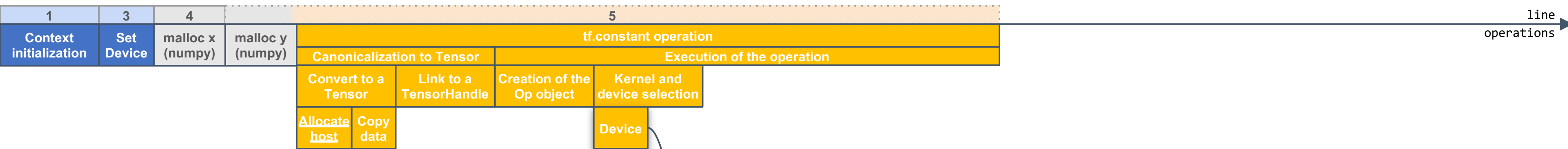
Op object





- Gathers all the information needed to select a kernel and a device
    - Input data (TensorHandle)
    - Attributes of the operation
    - Operation name

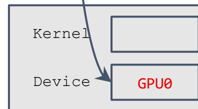
TF execution: Enqueueing thread



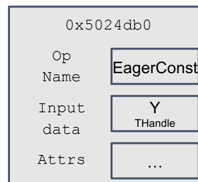
context device=GPU0

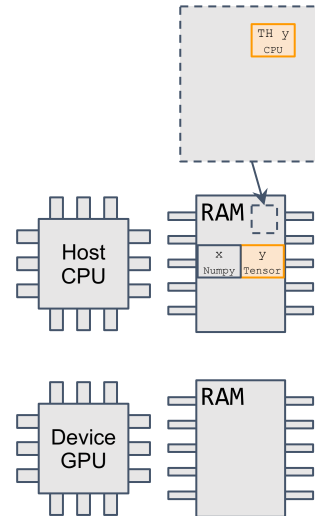
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

Kernel and Device



Op

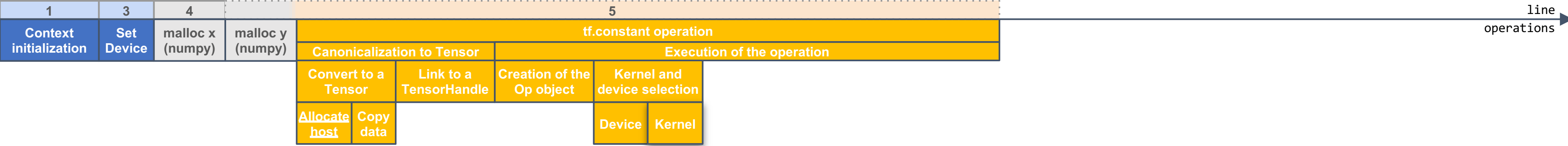




Kernel & device

- Op will execute on the user-specified device
  - If no device is specified, the runtime tries to choose the fastest device available

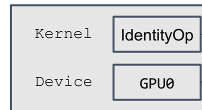
TF execution: Enqueueing thread



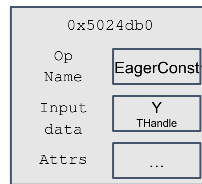
context device=GPU0

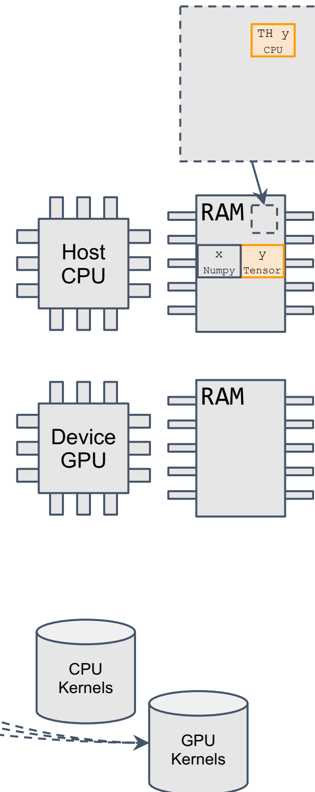
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

Kernel and Device



Op

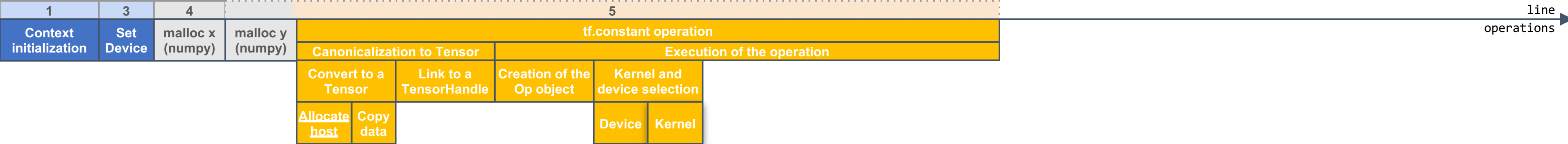




Kernel & device

- Selecting the kernel to execute based on the Op name
  - Based on the kernels registered in TF system

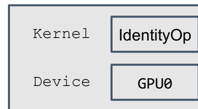
TF execution: Enqueueing thread



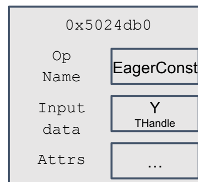
context device=GPU0

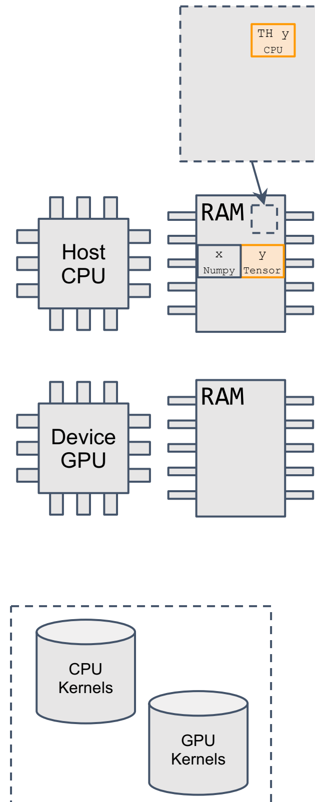
```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```

Kernel and Device

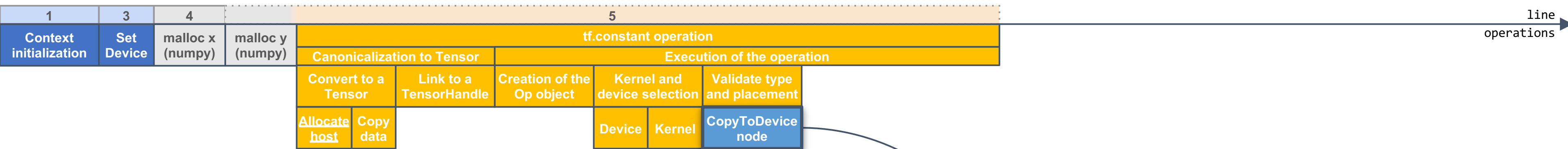


Op





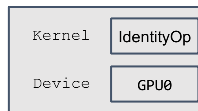
TF execution: Enqueueing thread



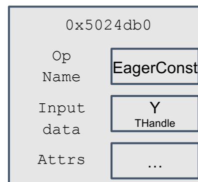
context device=GPU0

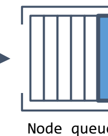
```python
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```

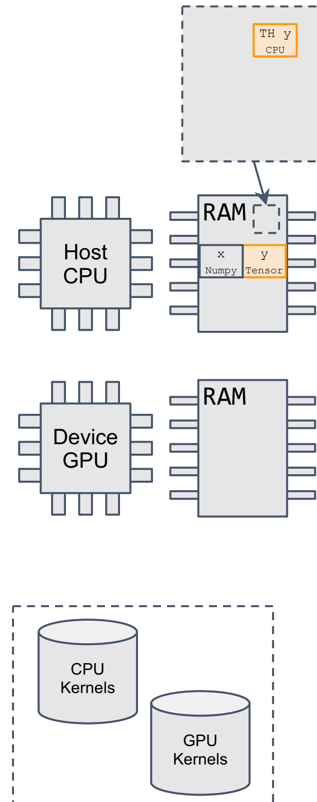
Kernel and Device



Op

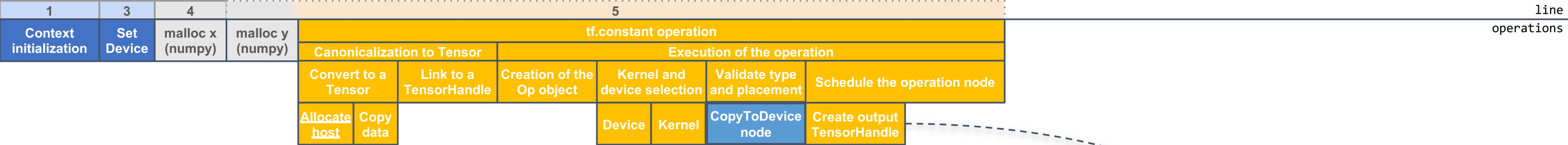






- Copy the input of the operation to the expected device
  - Launches a CopyToDevice **node**

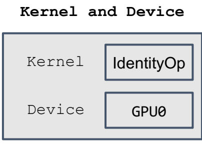
TF execution: Enqueueing thread



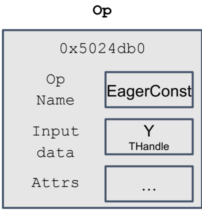
context device=GPU0

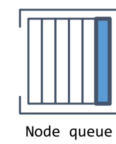
```python
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```

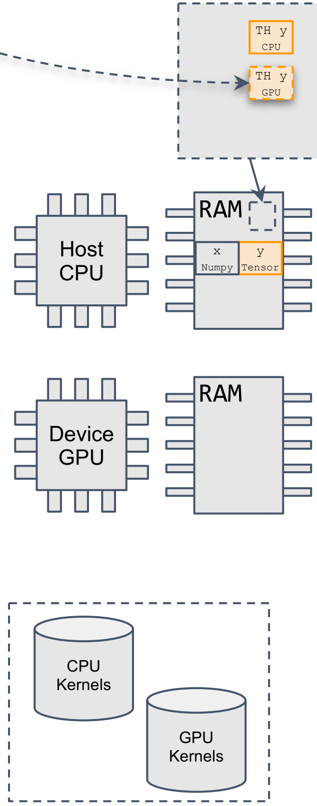
Kernel and Device



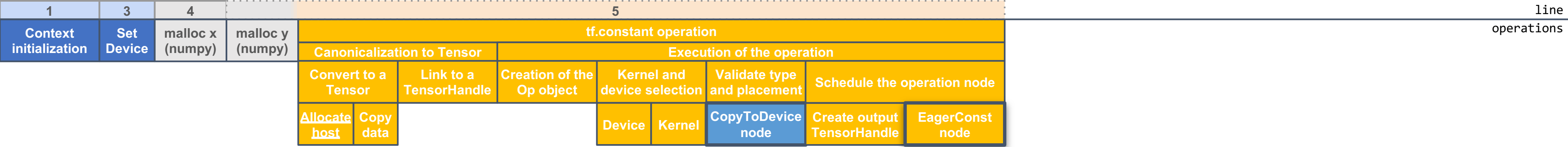
Op





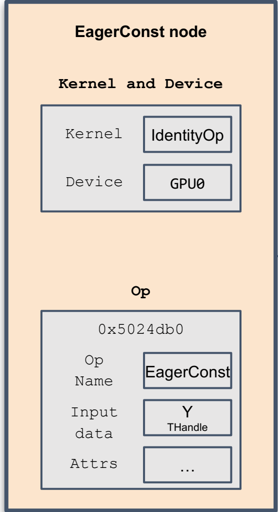


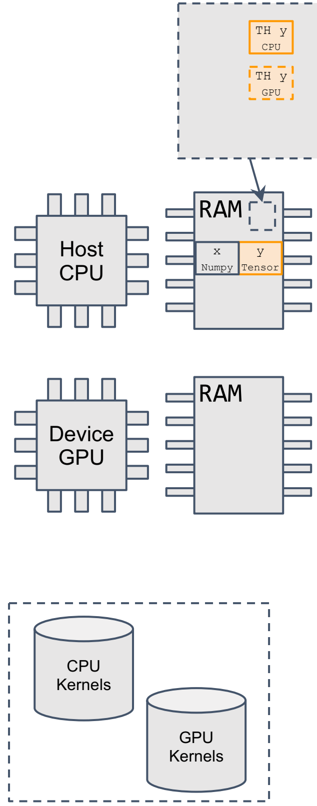
TF execution: Enqueueing thread



context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

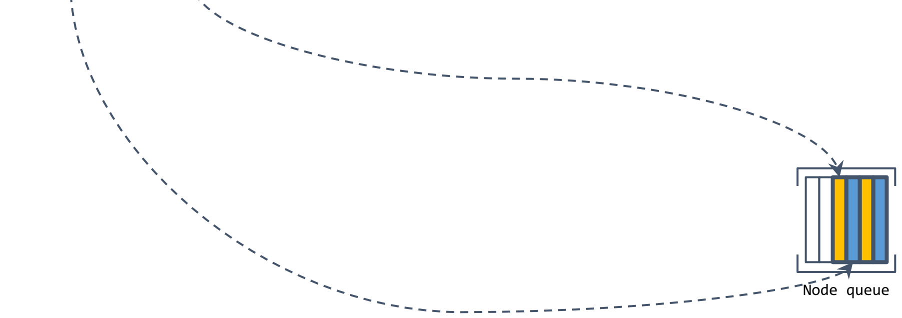


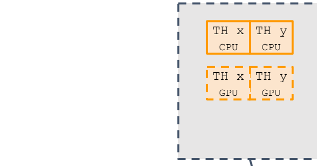


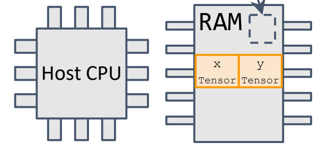
TF execution: Enqueueing thread

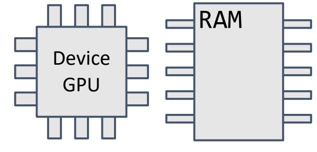
context device=GPU0

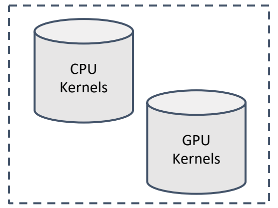
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



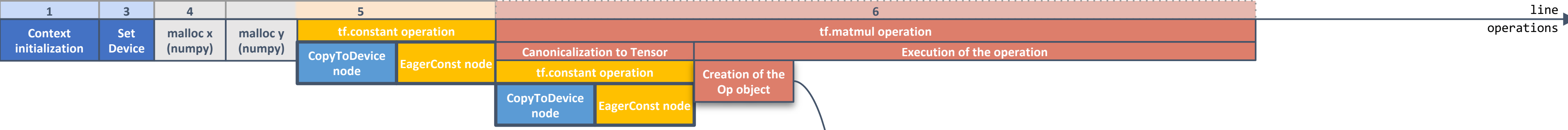






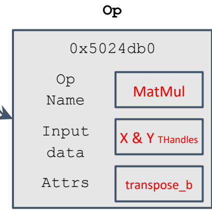


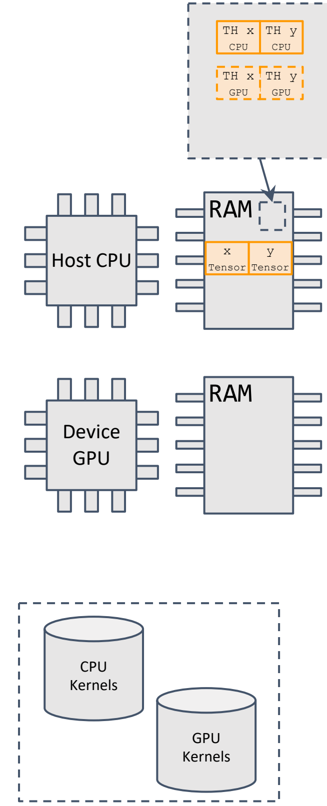
TF execution: Enqueueing thread



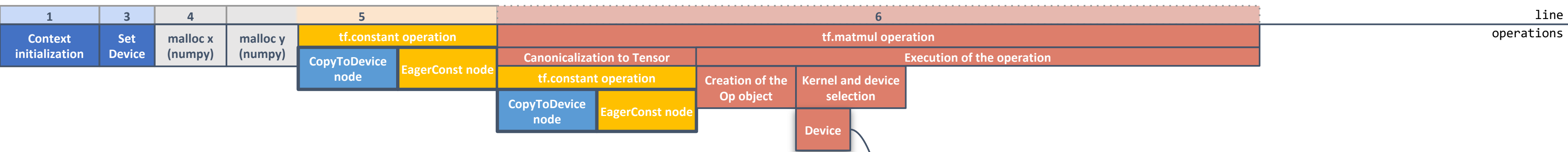
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



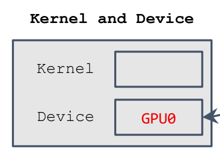


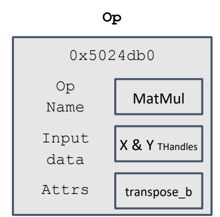
TF execution: Enqueueing thread

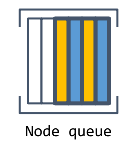


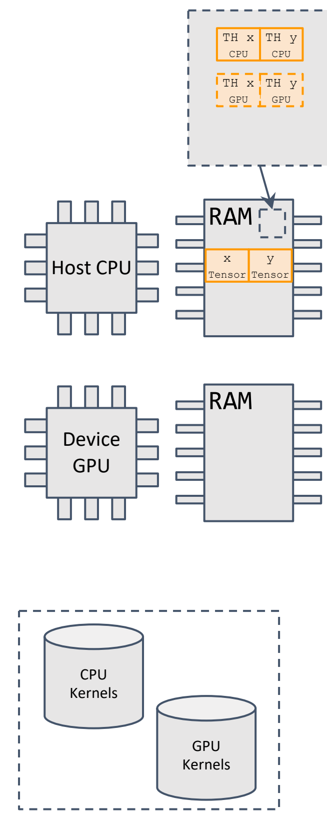
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

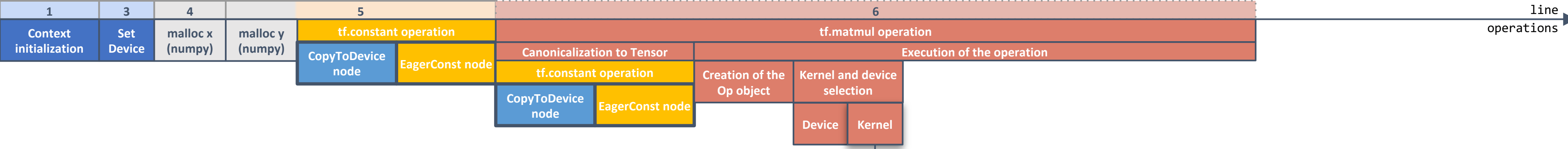






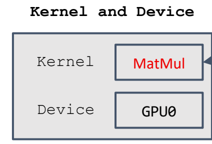


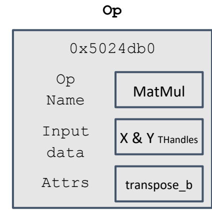
TF execution: Enqueueing thread

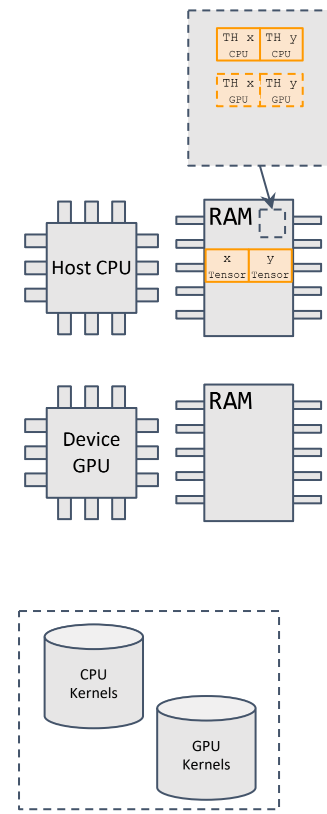


context device=GPU0

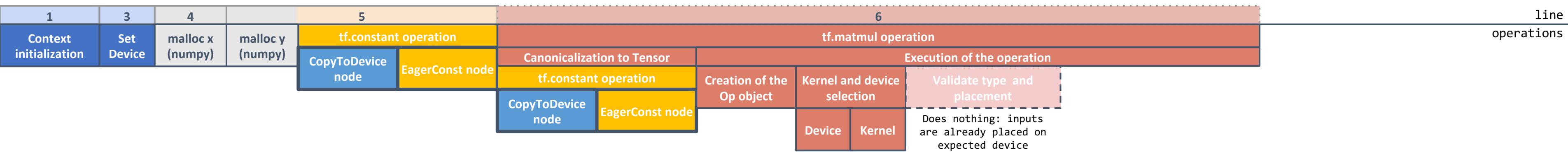
```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```







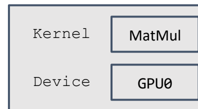
TF execution: Enqueueing thread



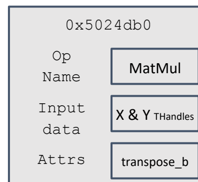
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

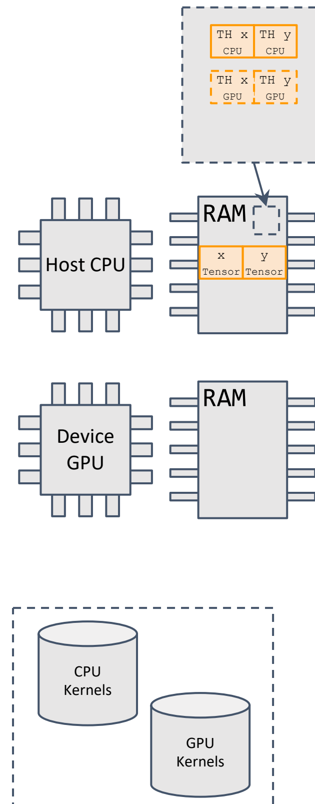
Kernel and Device



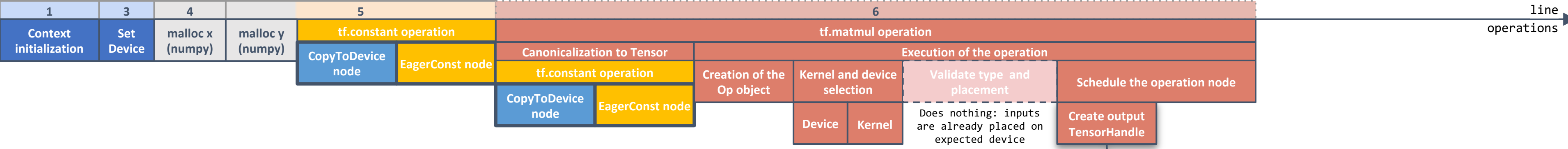
Op



Node queue

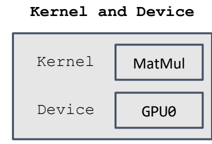


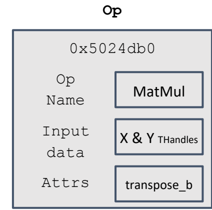
TF execution: Enqueueing thread

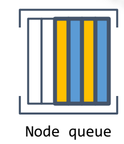


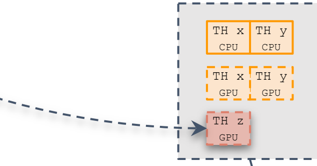
context device=GPU0

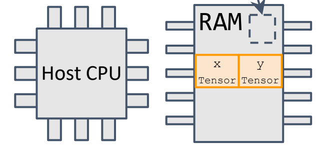
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

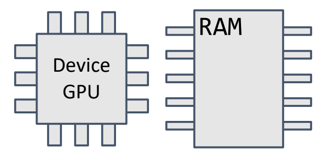


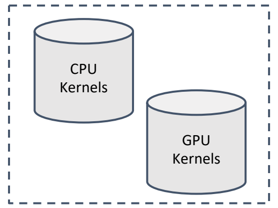




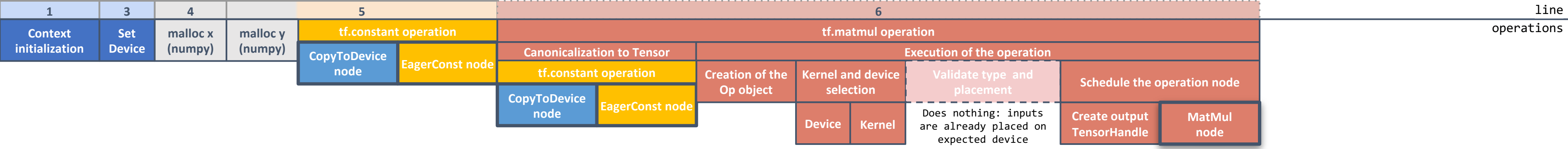






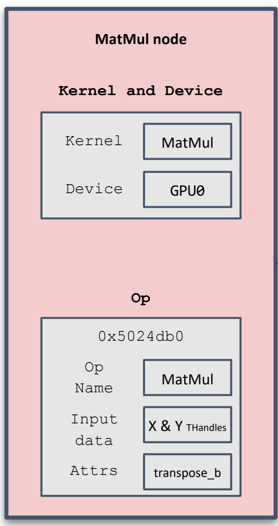


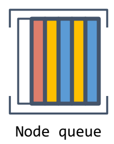
TF execution: Enqueueing thread

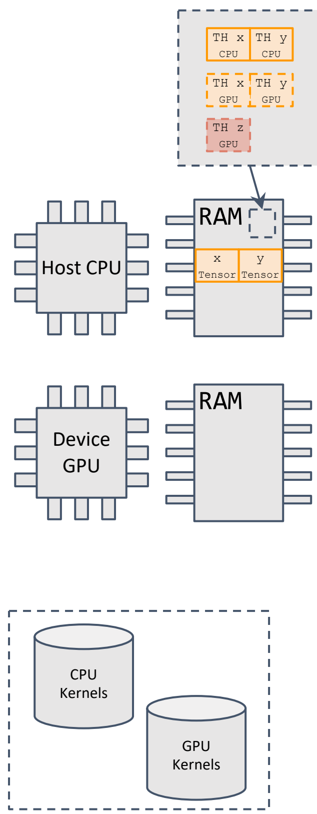


context device=GPU0

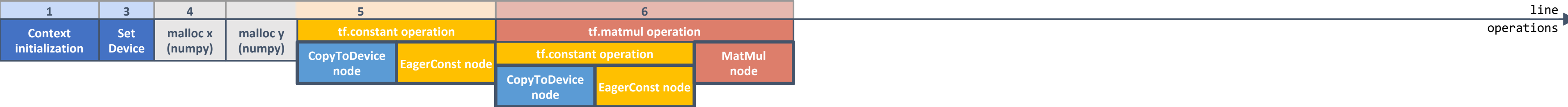
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```





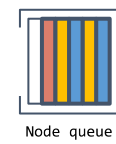


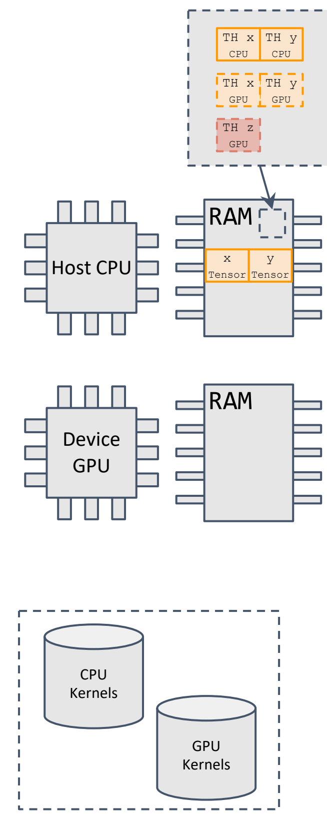
TF execution: Enqueueing thread



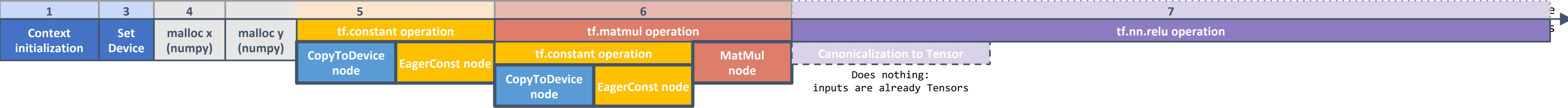
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



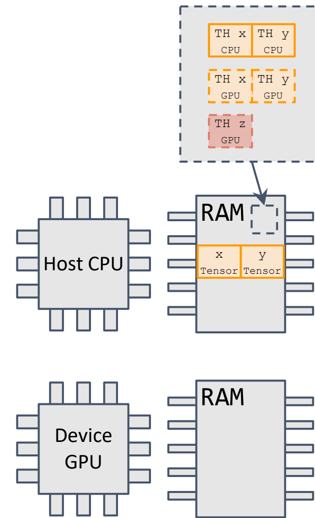


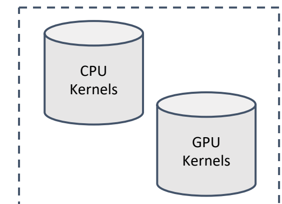
TF execution: Enqueueing thread



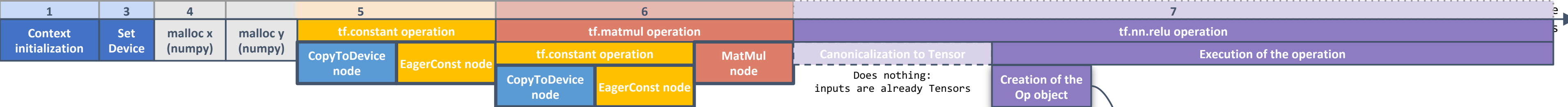
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```





TF execution: Enqueueing thread

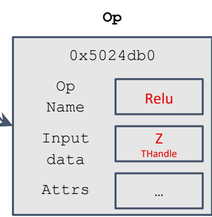


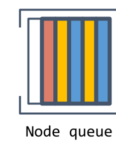
Does nothing:
inputs are already Tensors

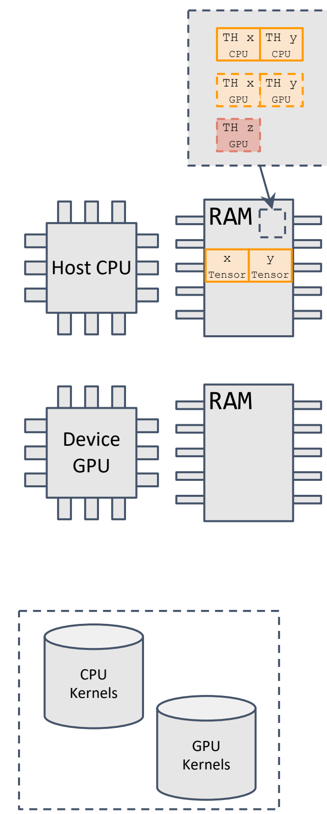
context device=GPU0

```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

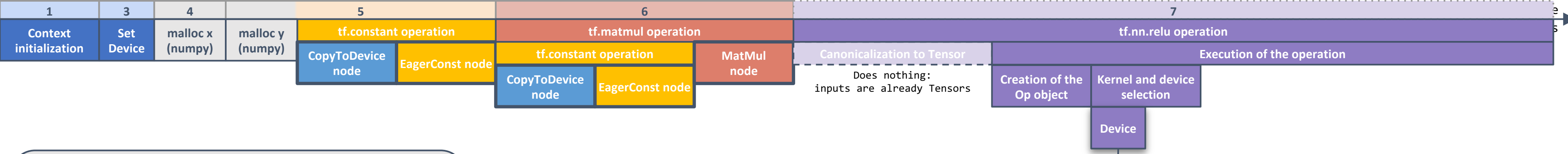
Op





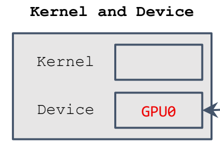


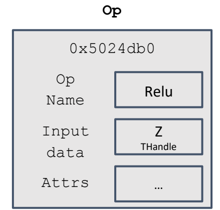
TF execution: Enqueueing thread

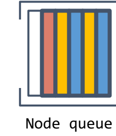


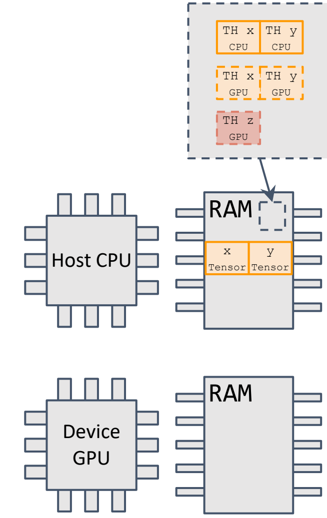
context device=GPU0

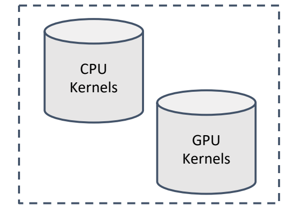
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



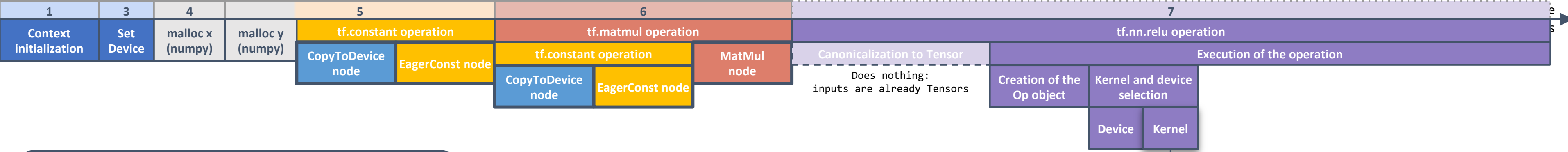






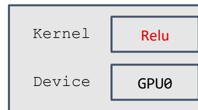


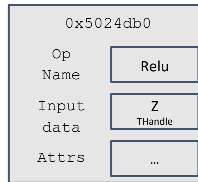
TF execution: Enqueueing thread

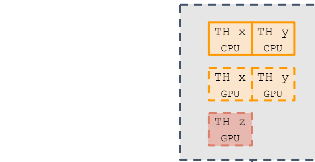


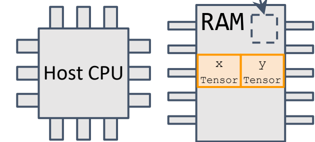
context device=GPU0

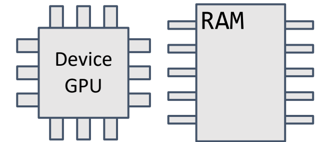
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

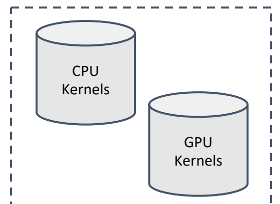




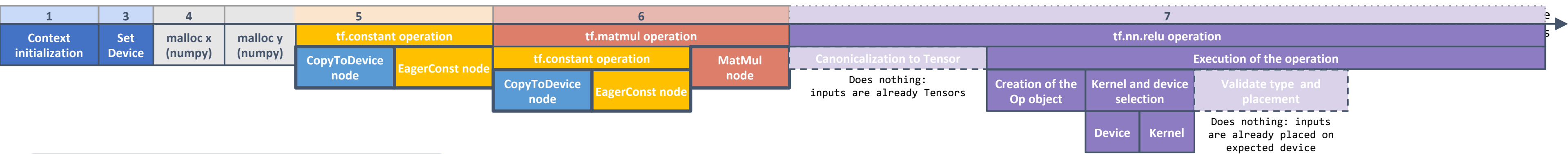








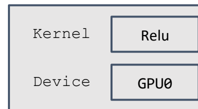
TF execution: Enqueueing thread



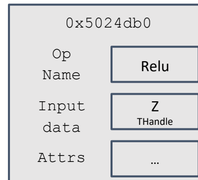
context device=GPU0

```python
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

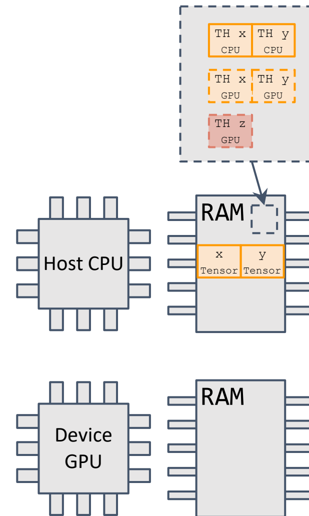
Kernel and Device

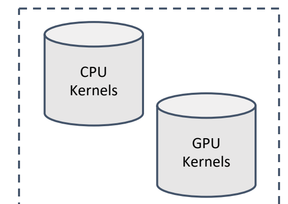


Op

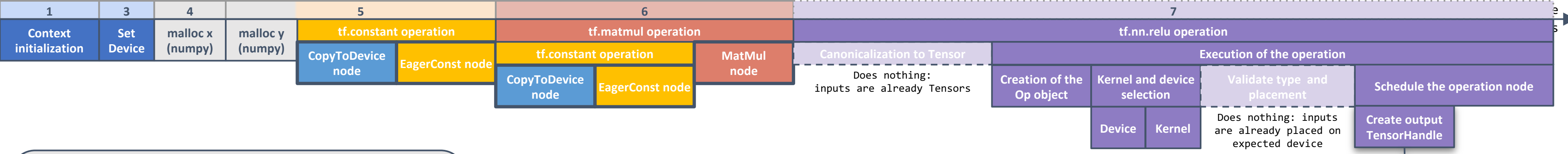


Node queue





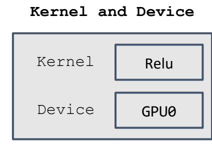
TF execution: Enqueueing thread



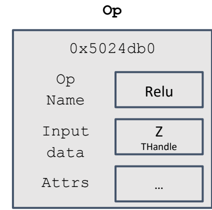
context device=GPU0

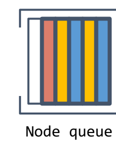
```
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

Kernel and Device

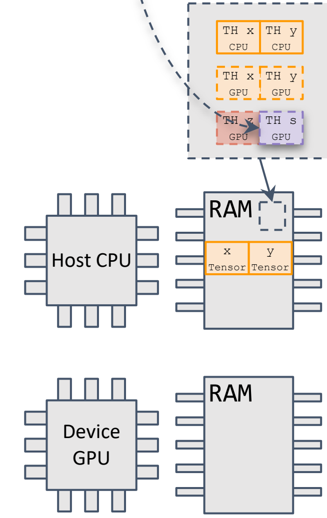


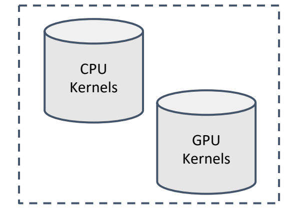
Op



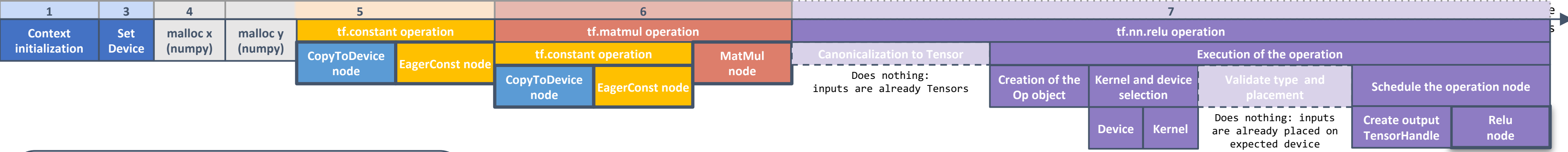


Node queue



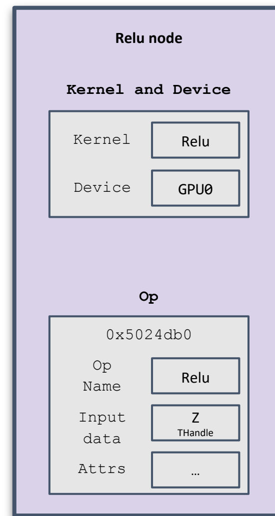


TF execution: Enqueueing thread

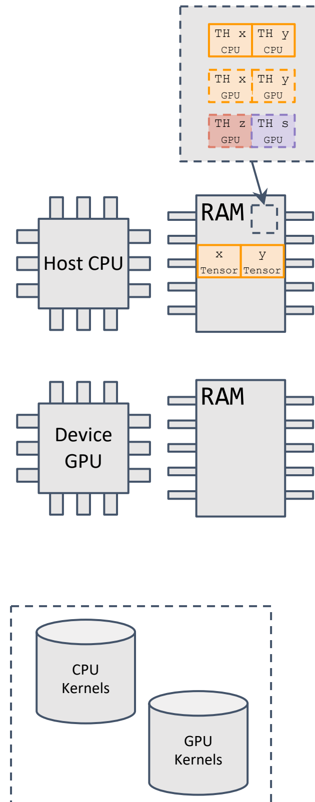


context device=GPU0

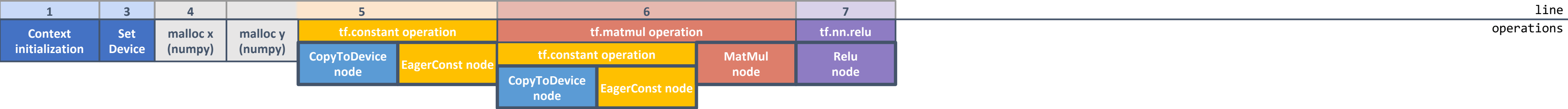
```python
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```



Node queue

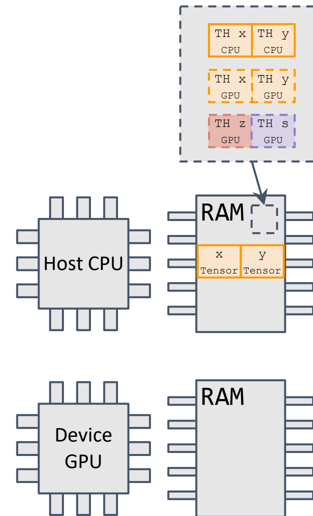


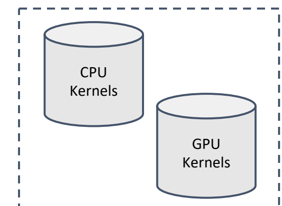
TF execution: Enqueueing thread



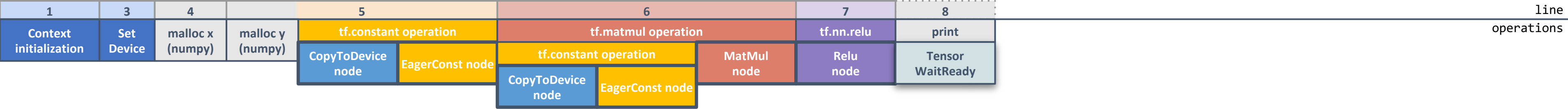
context device=GPU0

```
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```



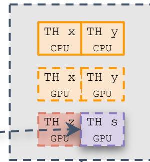


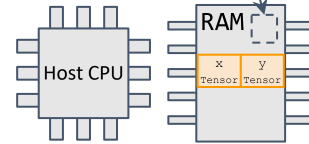
TF execution: Enqueueing thread

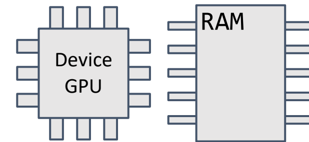


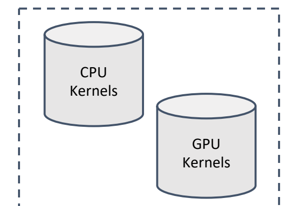
context device=GPU0

```python
1  import tensorflow as tf
2  import numpy as np

3  with tf.device("/GPU:0"):
4      x = np.random.randn(1024,1024)
5      y = tf.constant(np.random.randn(1024,1024))

6      z = tf.matmul(x, y, transpose_b=True)

7      s = tf.nn.relu(z)

8      print(s)
```

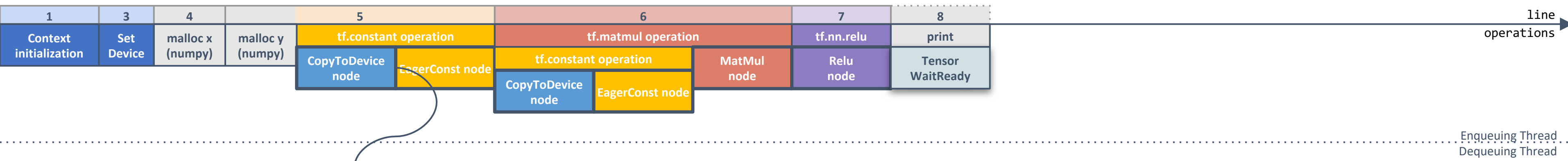


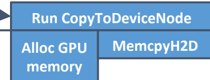




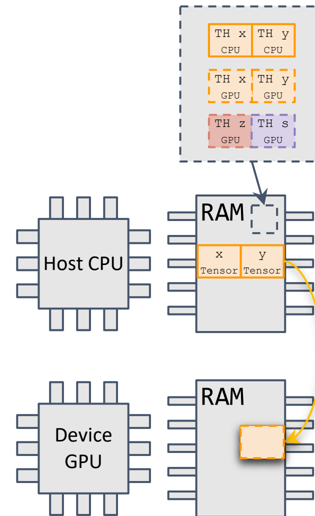


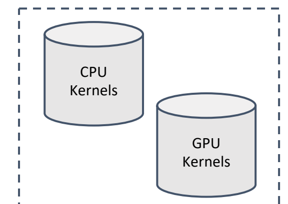
TF execution: Dequeueing thread



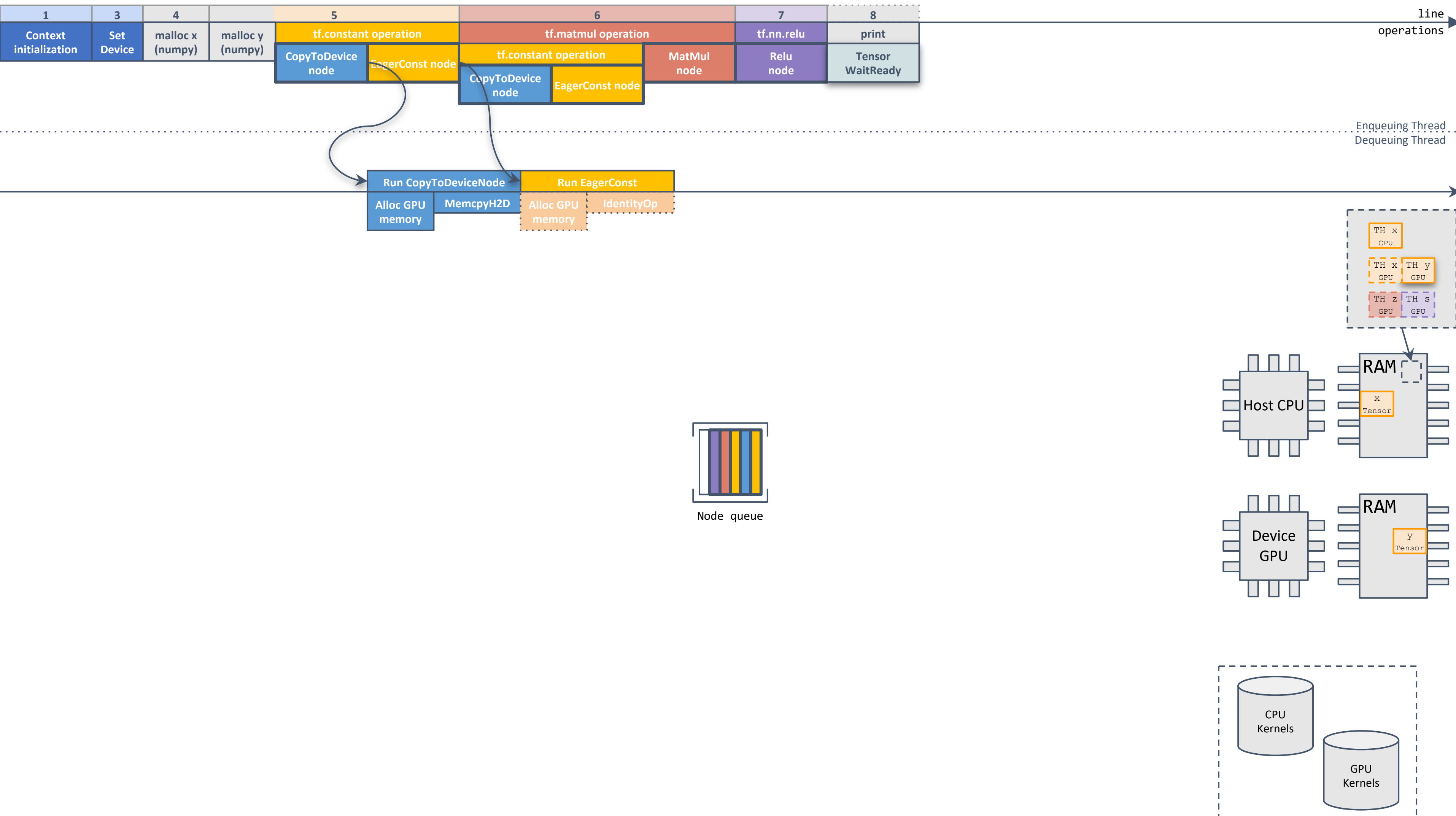


+ Delete TensorHandle
→ Deallocates memory

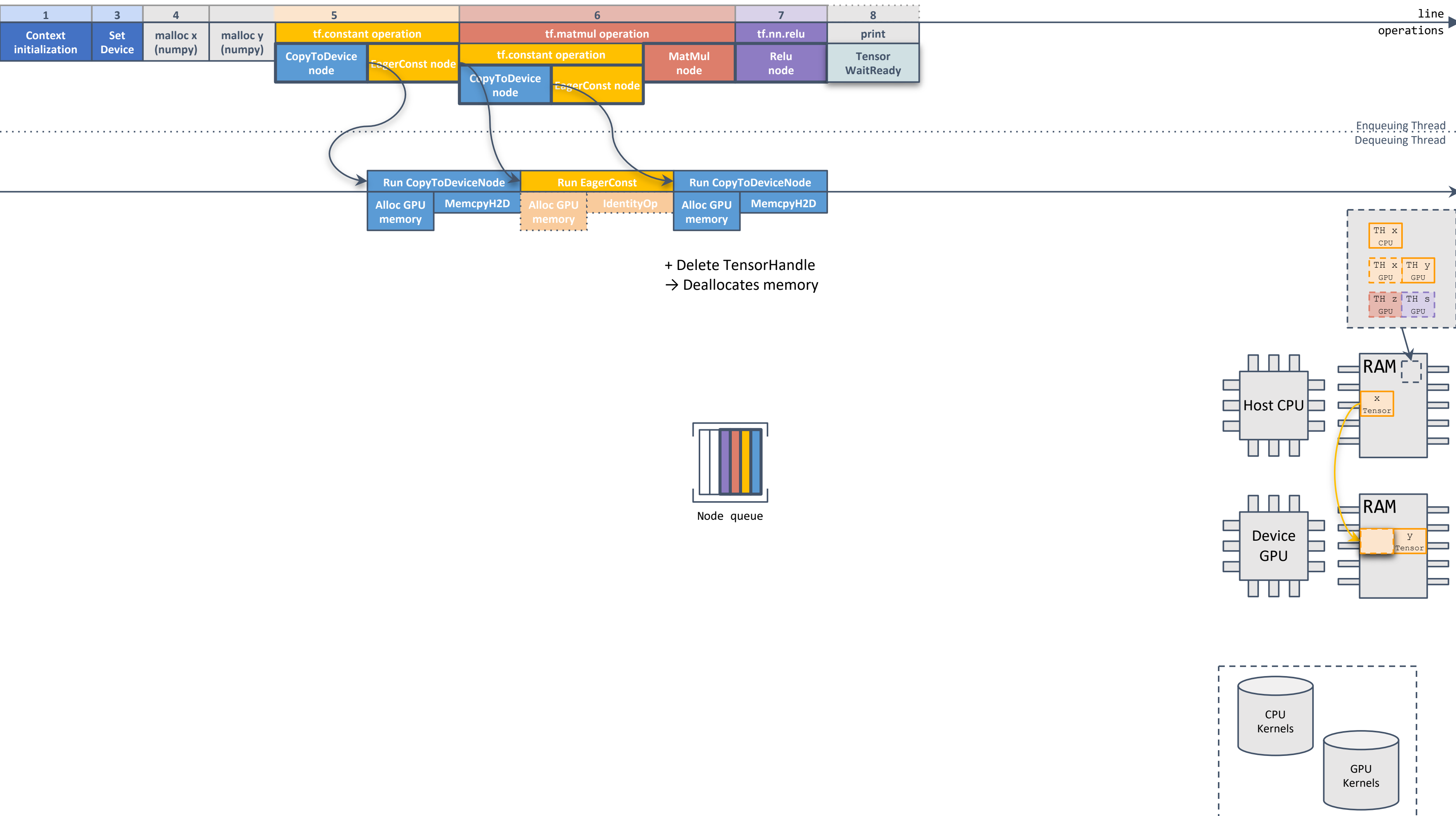




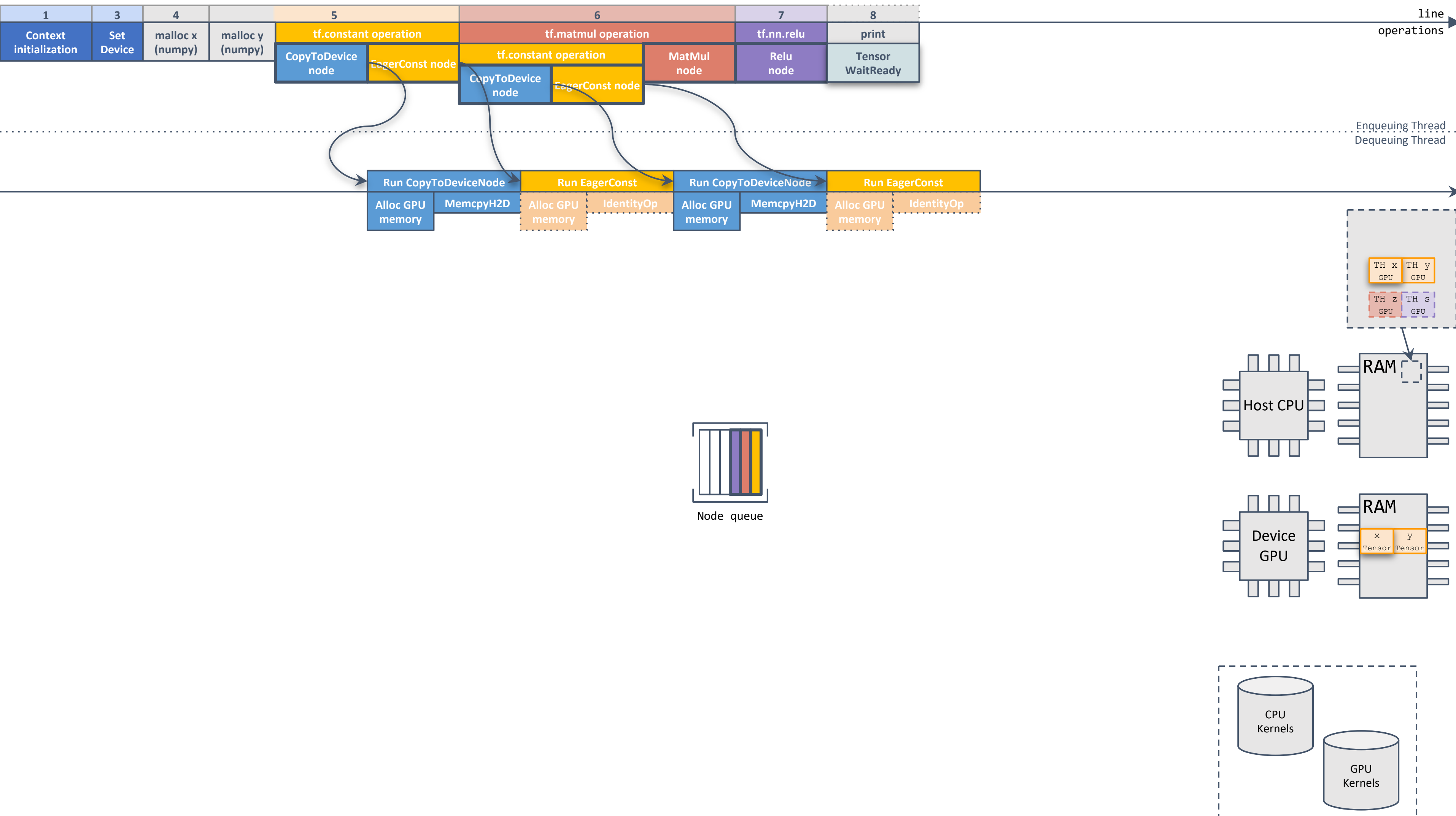
TF execution: Dequeueing thread



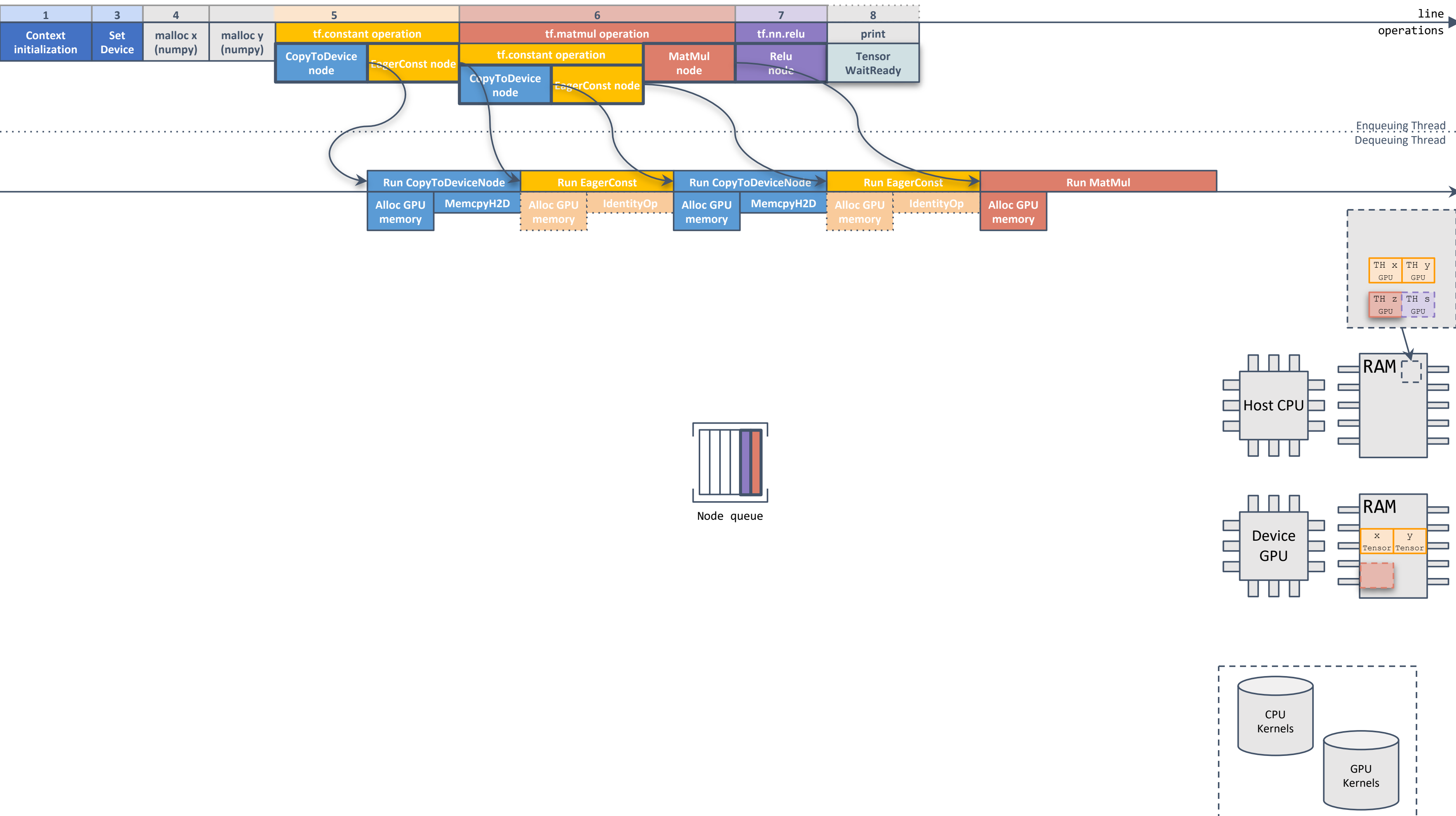
TF execution: Dequeueing thread



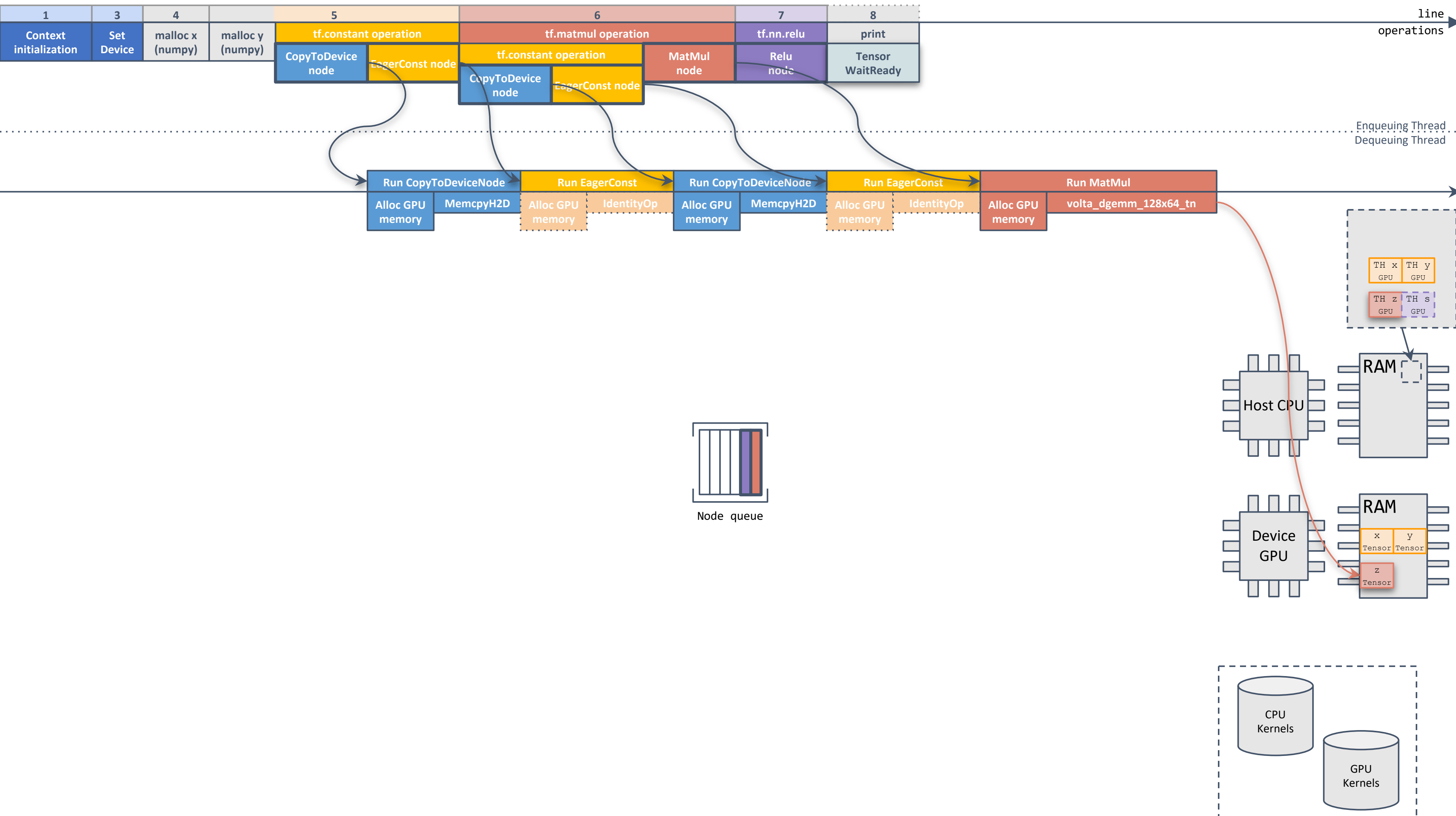
TF execution: Dequeueing thread



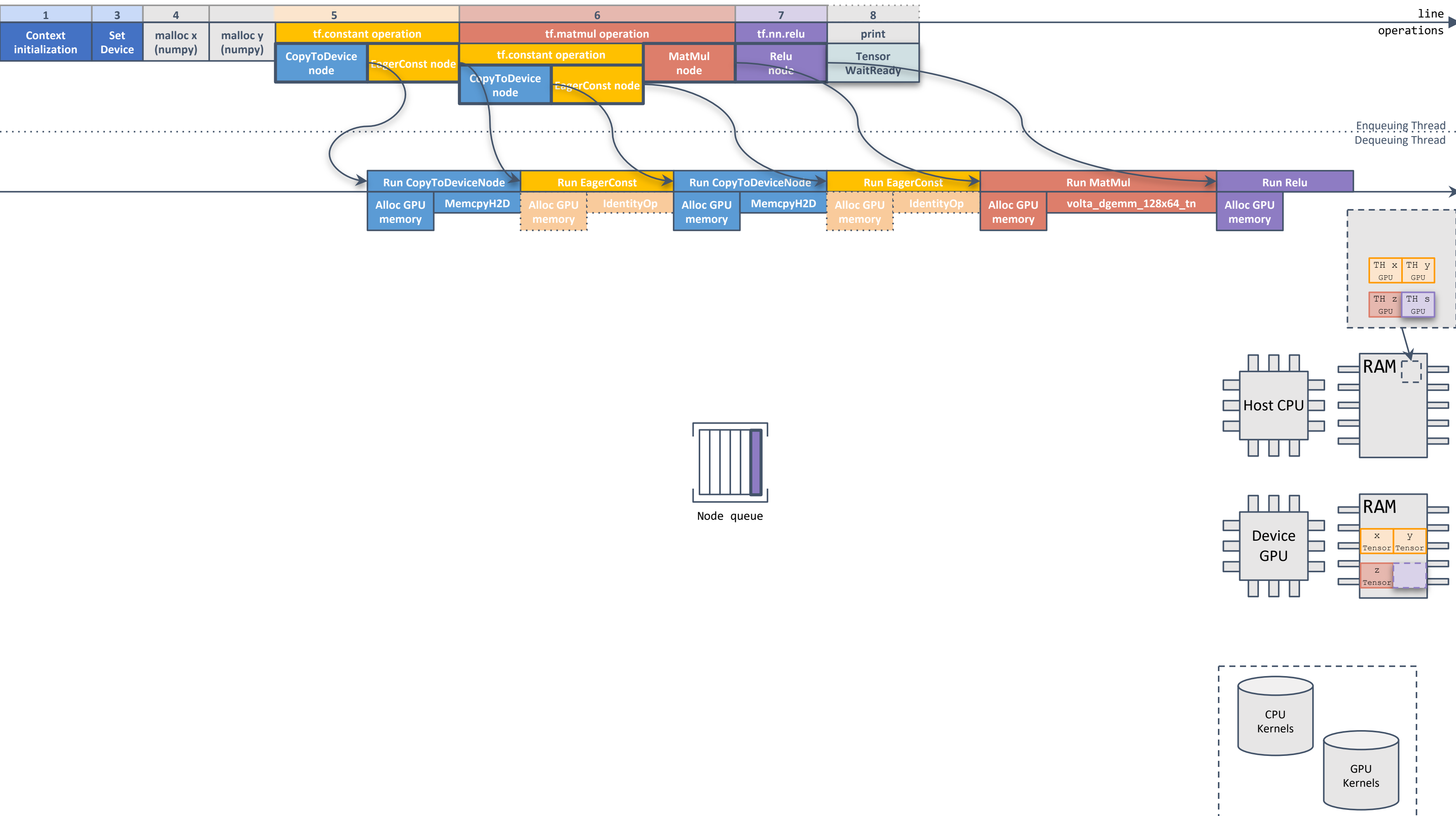
TF execution: Dequeueing thread



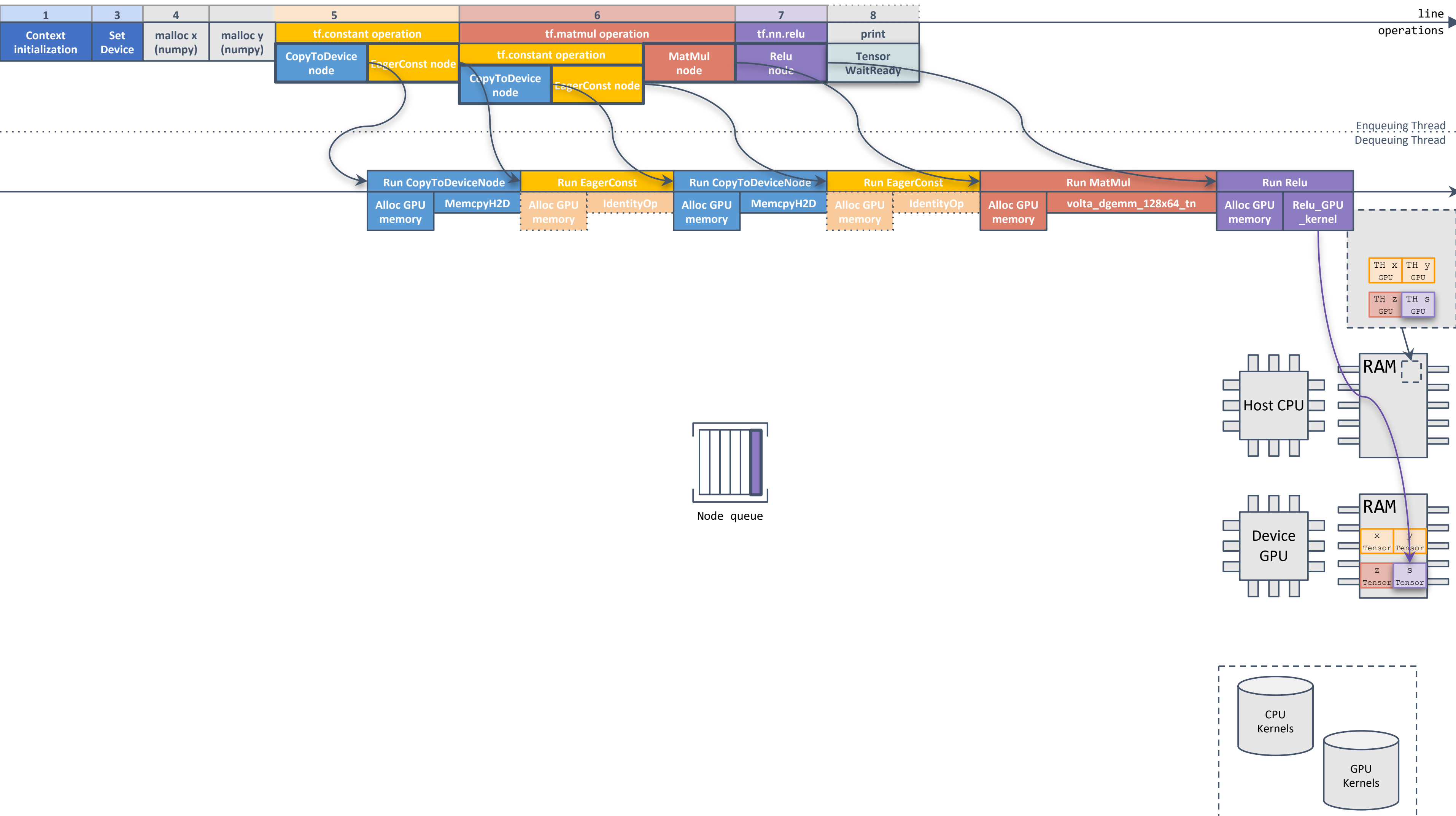
TF execution: Dequeueing thread



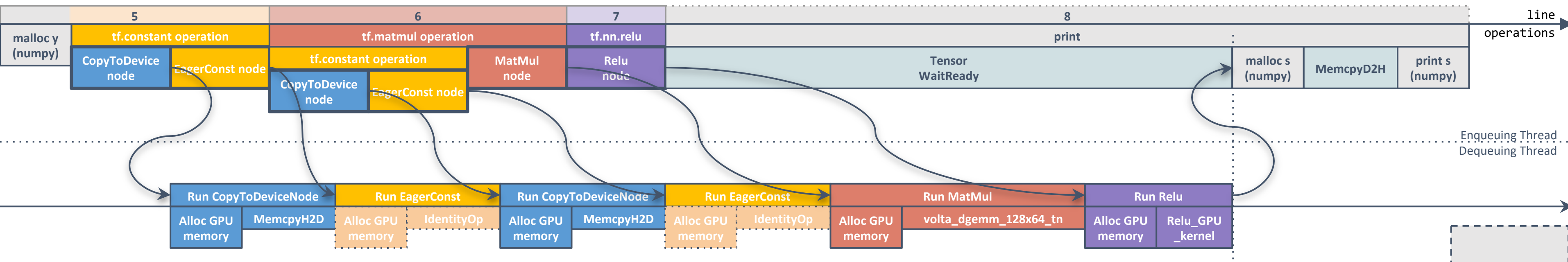
TF execution: Dequeueing thread



TF execution: Dequeueing thread

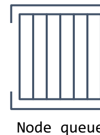


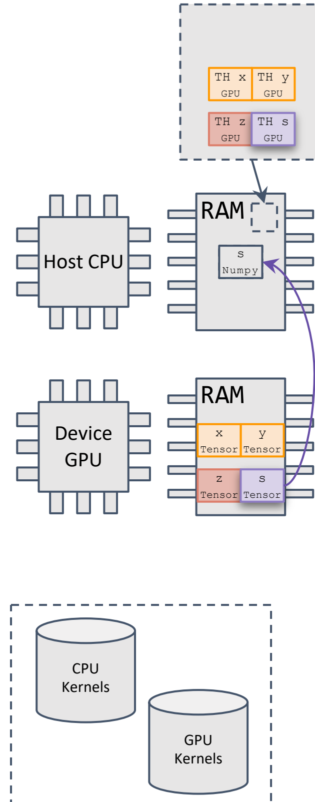
TF execution: Back to enqueueing thread



context device=GPU0

```python
1 import tensorflow as tf
2 import numpy as np

3 with tf.device("/GPU:0"):
4     x = np.random.randn(1024,1024)
5     y = tf.constant(np.random.randn(1024,1024))

6     z = tf.matmul(x, y, transpose_b=True)

7     s = tf.nn.relu(z)

8     print(s)
```





Pros and Cons of TF Eager Execution

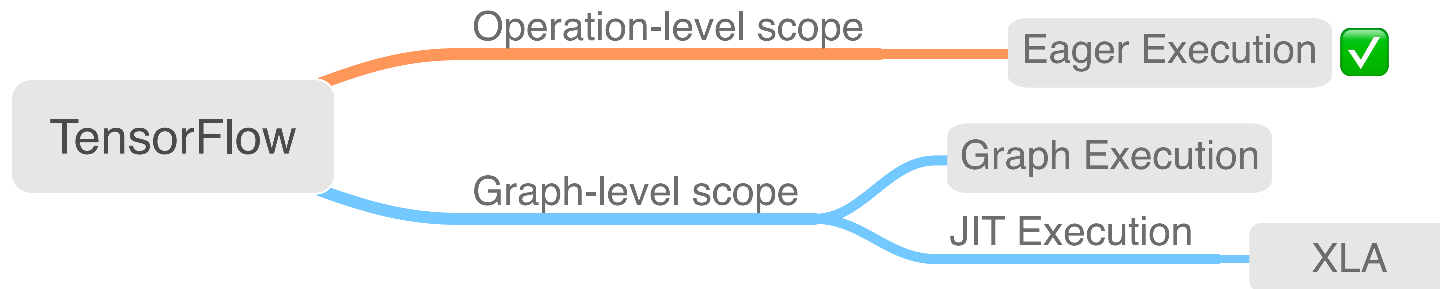
Strengths

- Easy debugging
  - Executes in program order
  - Immediate run-time errors
- Convenient library API to interface with hardware vendors
  - Few hundreds of operations but not too many

Weaknesses

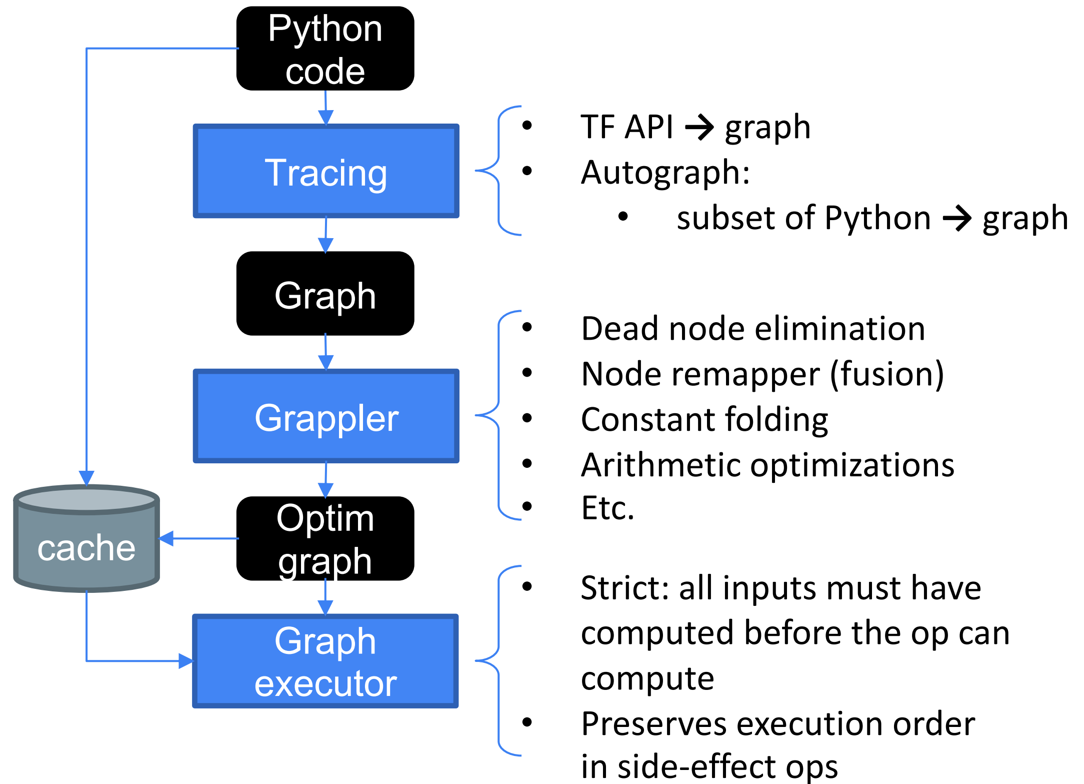
- Slow execution
  - Local optimizations: at the operation level
  - Large Python overhead
    - When running small kernels
    - When requiring frequent synchronization due to enqueueing thread needing tensor values

TensorFlow execution modes



Graph Execution

```python
import tensorflow as tf
import numpy as np

@tf.function
def example_func(x, y):
   z = tf.matmul(x, y, transpose_b=True)
   return tf.nn.relu(z)

with tf.device("/GPU:0"):
   x = np.random.randn(1024,1024)
   y = tf.constant(np.random.randn(1024,1024))
   s = example_func(x, y)
   print(s)
```



Pros and Cons of TF Graph Execution

Strengths

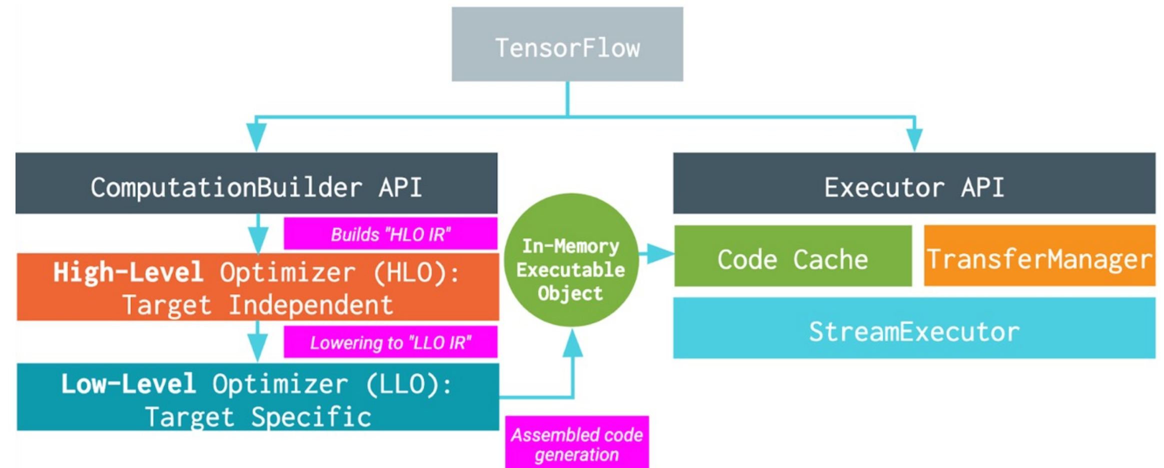
- Dynamic
  - Parametric graphs
    - Same graph for any shape: allows researchers to richly express themselves
  - Support for data-dependent shapes
- Higher performance than Eager Execution
  - Optimizations at the graph level: inter-operation optimizations

Weaknesses

- Difficult to debug
- Untaped optimization opportunities
  - Difficult to analyze & optimize the graph without information on shapes
  - Still have to compile to the same API: limited composability

XLA Execution

```python
import tensorflow as tf
import numpy as np

@tf.function(jit_compile=True)
def example_func(x, y):
    z = tf.matmul(x, y, transpose_b=True)
    return tf.nn.relu(z)

with tf.device("/GPU:0"):
    x = np.random.randn(1024,1024)
    y = tf.constant(np.random.randn(1024,1024))
    s = example_func(x, y)
    print(s)
```



XLA execution

XLA is a domain-specific compiler

- Eliminates op dispatch overhead
- Optimizes memory: memory reuse, in-place updates
- Static: more aggressive optimizations (e.g., op fusion, unrolls loops via known dimensions)
- Reduction of executable size

TF with XLA wrt Graph Execution

fp32 fp16

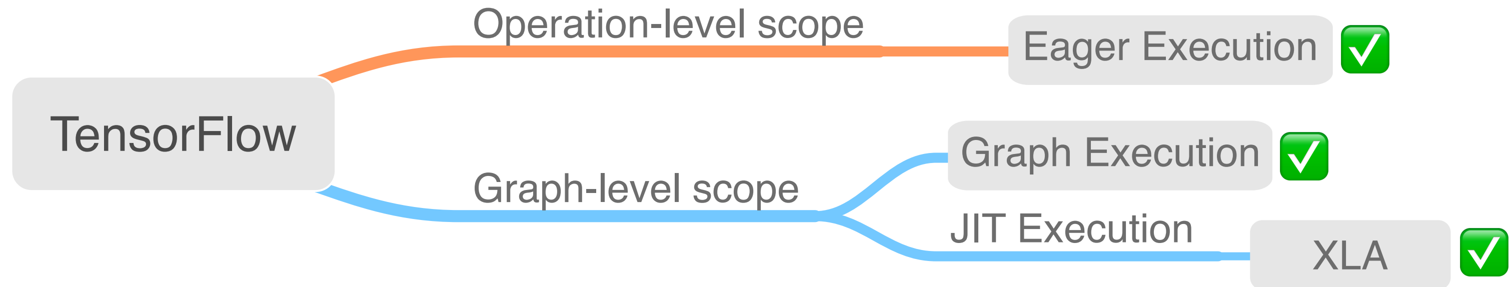
2.0
1.5
1.0
0.5
0.0

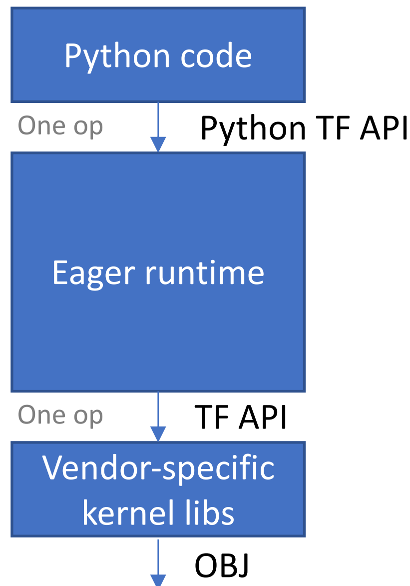
XLA is faster

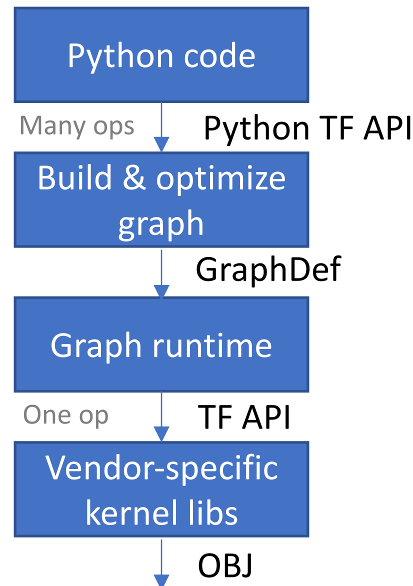
XLA is slower

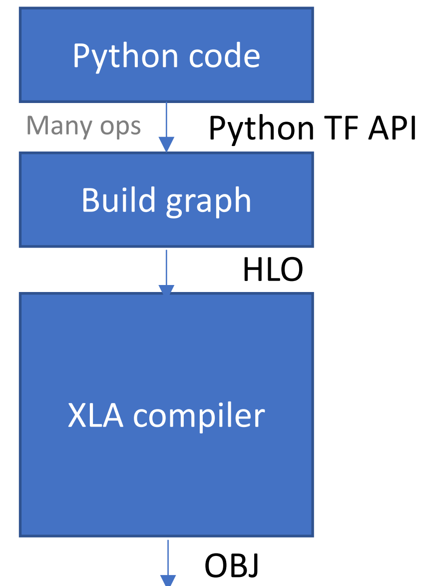
https://blog.tensorflow.org/2018/11/pushing-limits-of-gpu-performance-with-xla.html

TensorFlow execution modes

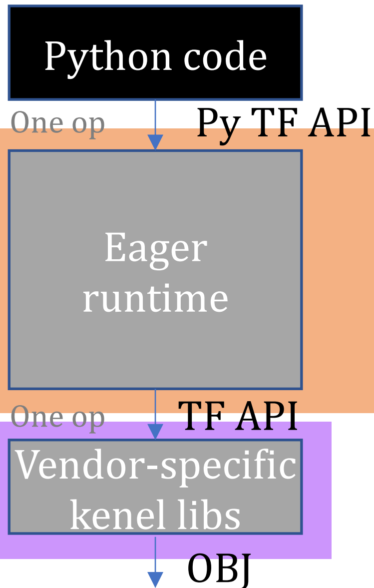


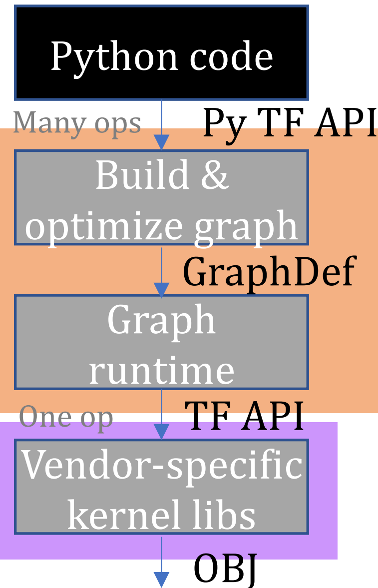


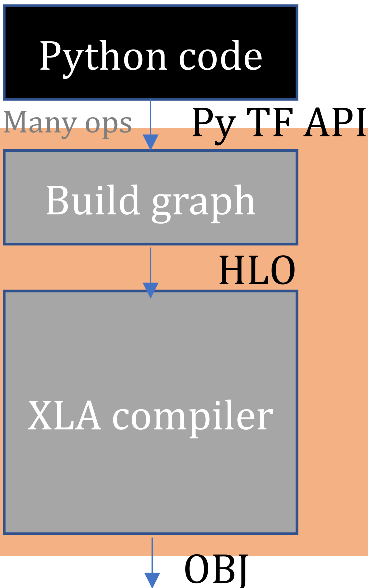


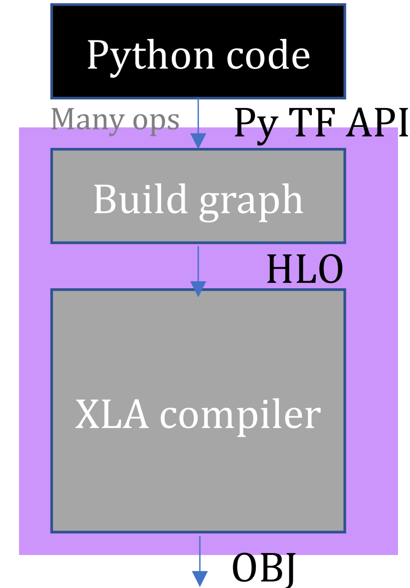


Execution vs. design time









Execution time   Design/compile time

Conclusions

- The incredible processing power of GPUs has been a major factor in the success of machine learning
- GPUs need carfully crafted low-level code to be able to achieve high performance
- SotA ML frameworks achieve high performance and scalability from a user-friendly programming interface
- Modern ML frameworks are becoming very complex
  - Different front-ends, different back-ends, different execution scenarios
    - E.g., model development vs. model deployment, inference vs. training, etc.
- The optimal split between execution and design time is not obvious and seems to be user-scenario dependent

Acknowledgements

Paul Delestrac
PhD Candidate at ADAC/LIRMM

Onur Mutlu
Professor at ETH Zurich

SAFARI

https://www.youtube.com/@OnurMutluLectures

We are hiring

If you're looking to further your research in computer architecture and systems, we have multiple great **PostDoc opportunities**

Email me at david.novo@lirmm.fr

Machine Learning & GPUs

ARCHI 2023

David Novo

Chargé de Recherche CNRS

https://www.lirmm.fr/david-novo/

cnrs

NVIDIA A100