CMSC828G: Special Topics in Systems for Machine Learning

CUDA and Triton

Instructor: Yafan Huang

DEPARTMENT OF
COMPUTER SCIENCE

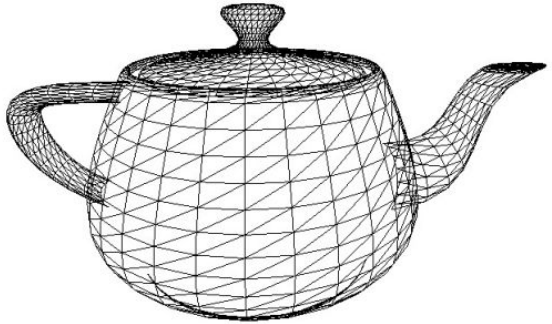
Before This Lecture

- Why still CUDA today?
  - Hardware changes. Principles persist.
  - AI makes systems intuition more valuable, not less.
  - Higher-level abstractions (DSL like Triton) still rest on GPU fundamentals.
  - For many scientific workloads, CUDA is still the portal to GPU.
  - And yes... people still hire kernel engineers. 😂

Outline

- Background
- CUDA Basics
- Optimizing CUDA Programs
- Triton Basics

What GPUs were Original Designed to do: 3D Rendering



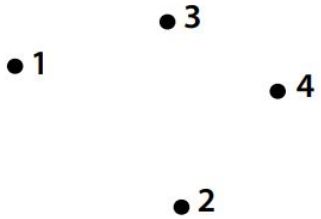
Input: description of a scene:
3D surface geometry

Output: Image of the scene

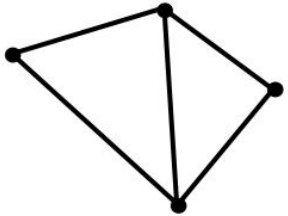
Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

3D Rendering Task with GPU

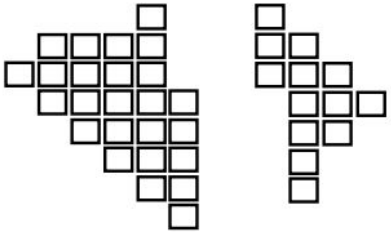
- Real-time graphic primitives (entities)



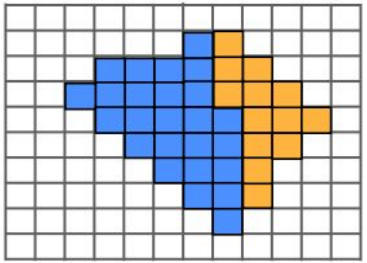
Vertices
(points in space)



Primitives
(e.g., triangles, points, lines)



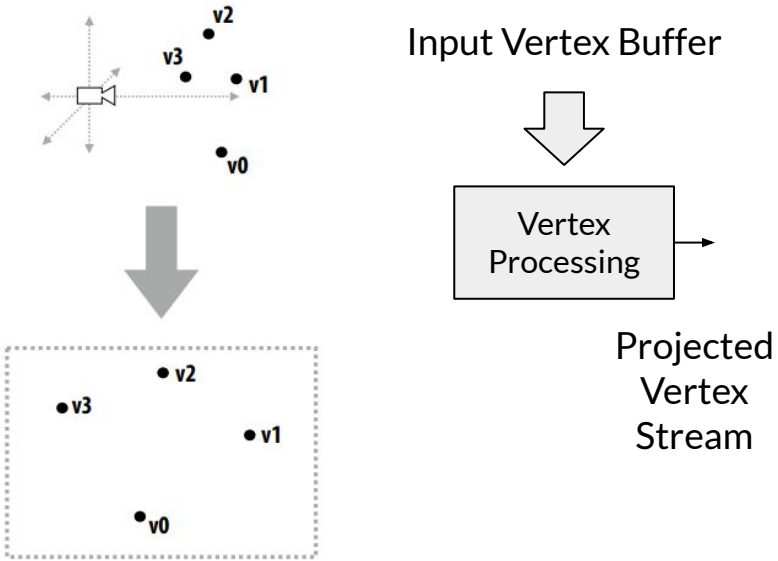
Fragments



Pixels (in an image)

Rendering a Picture

- Step 1: given a scene position, compute where the vertices lie on screen



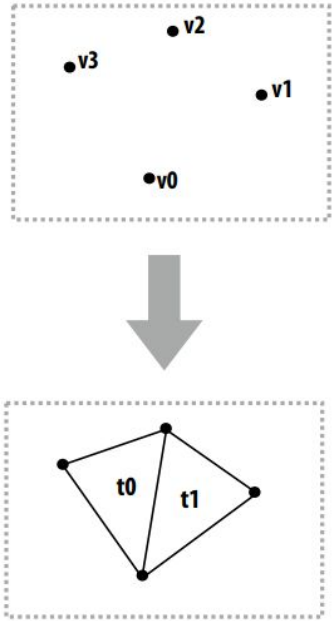
Input Vertex Buffer

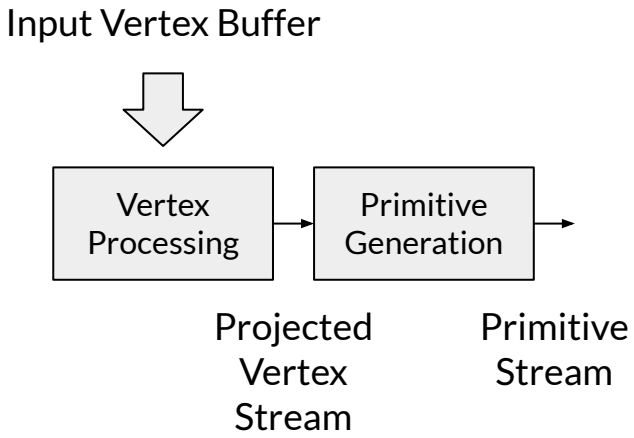
Vertex Processing

Projected Vertex Stream

Rendering a Picture

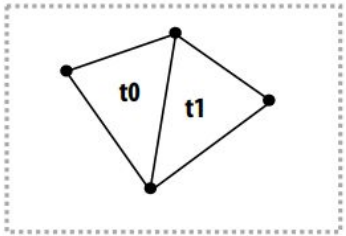
- Step 2: group vertices into primitives

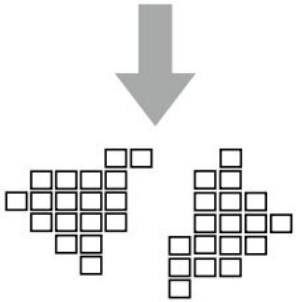


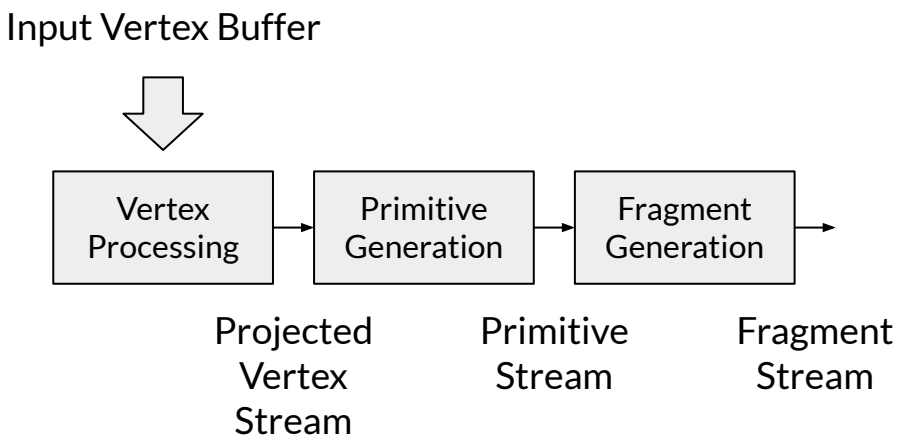


Rendering a Picture

- Step 3: generate one fragment for each pixel based on the primitive

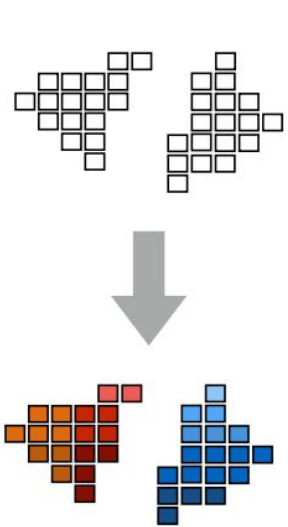


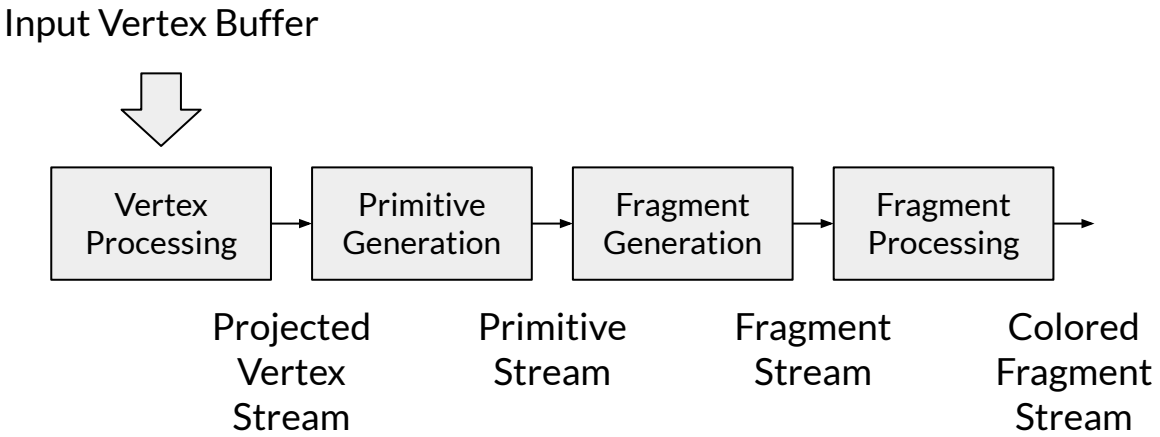




Rendering a Picture

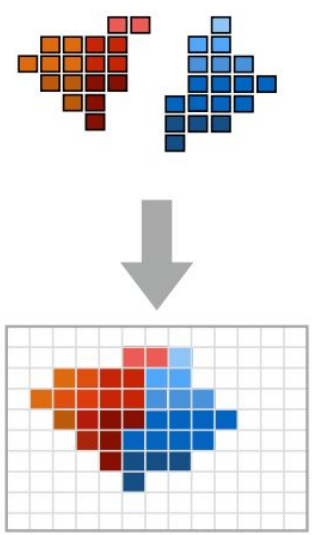
- Step 4: compute color of primitives for each fragment (based on scene lighting, material properties, entity interactions, etc)

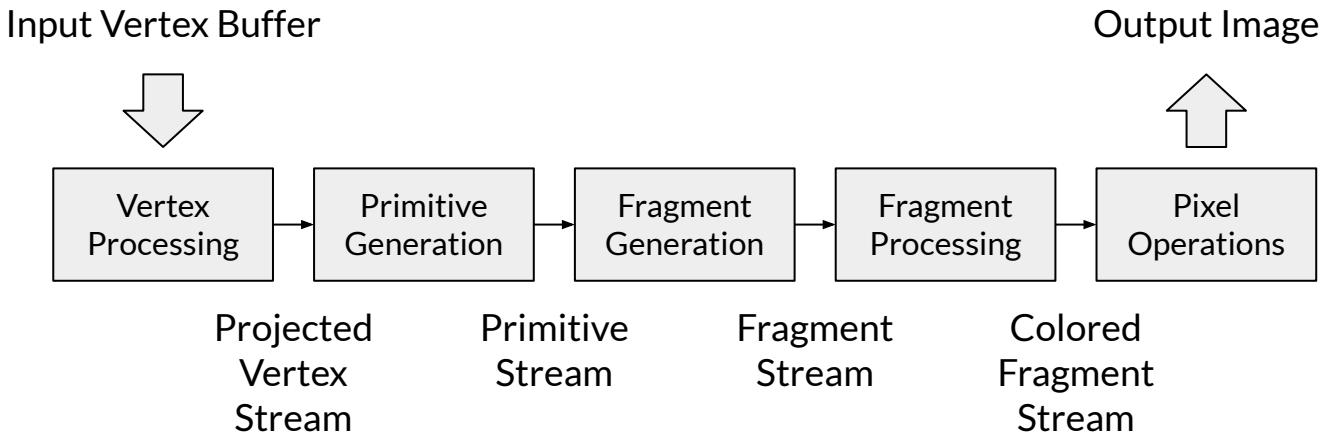




Rendering a Picture

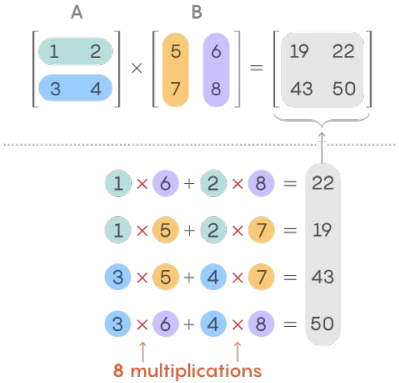
- Step 5: post-processing for each pixel and generate output image



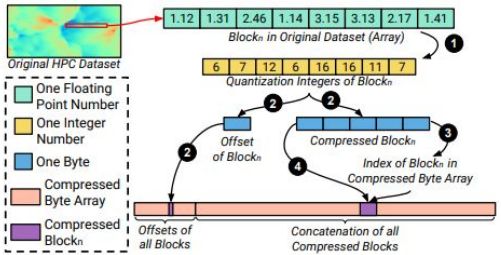


GPGPU: General Purpose Computations on GPU

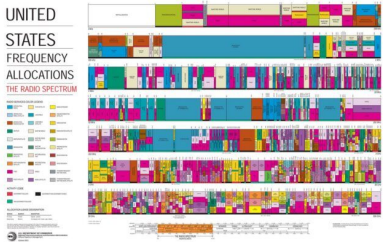
- GPGPU, short for General-purpose computing on graphics processing units, refers to the use of GPUs to perform computations beyond their traditional role of rendering graphics.



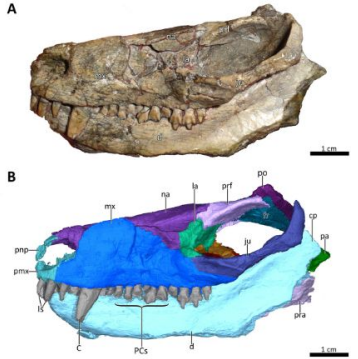
Matrix-Matrix Multiplication



Data Reduction



Audio Processing

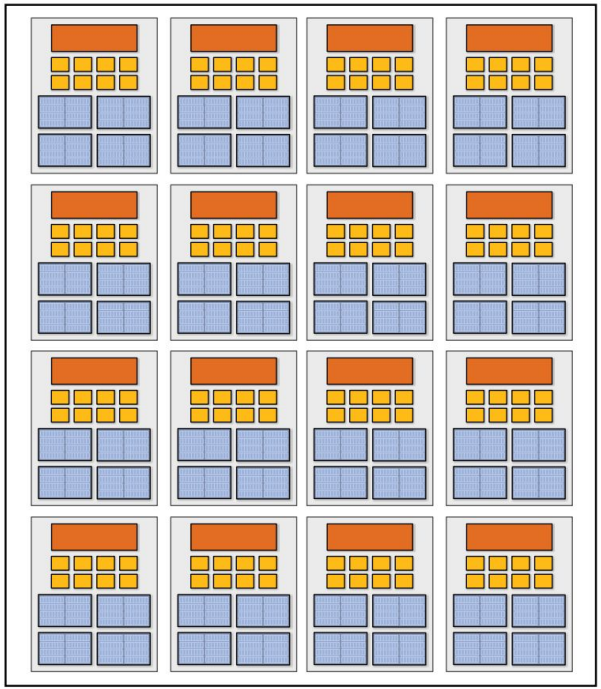


CT (Computed Tomography) Scan

CUDA Programming Language

- Introduced in 2007 with NVIDIA Tesla architecture.
- “C-like” language to express programs that run on GPUs using compute-mode hardware interface.
- Relatively low-level: CUDA’s abstractions closely match the capacities/performance characteristics of modern GPUs.
- Note: OpenCL is an open standards version of CUDA
    - CUDA only runs on NVIDIA GPUs.
    - OpenCL runs on CPUs and GPUs from many vendors.
    - Poor documentations, unfriendly developing environments, poor performance, etc. CUDA is much more popular now.

Basic GPU Architecture



~1500 GB/s

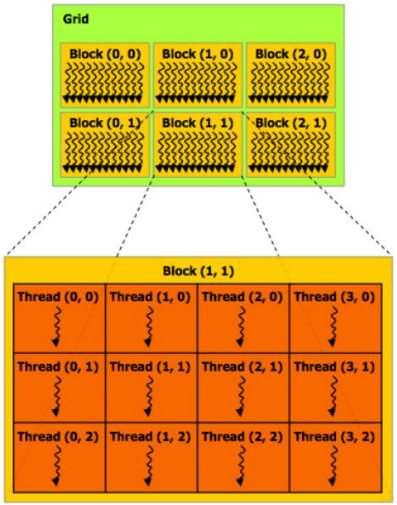
(for high end GPUs, e.g., A100)

Global Memory

HBM DRAM

CUDA Program Consists of Hierachy of Concurrent Threads

- Thread IDs can be up to 3-dimensional (2D example below)
- Multi-dimensional thread IDs are convenient for problems that are naturally N-D




```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Regular application thread running on CPU (the “host”)

Basic CUDA Syntax

Host code: serial execution
Running as part of normal C/C++ Application on CPU

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Regular application thread running on CPU (the “host”)

CUDA device code: kernel function (__global__ denotes a CUDA kernel function) runs on GPU

Function call returns when all threads have terminated

```
// kernel definition

__global__ void matrixAdd(
        float A[Ny][Nx], float B[Ny][Nx], float C[Ny][Nx])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    C[j][i] = A[j][i] + B[j][i];
}
```

CUDA kernel function

CUDA Execution Model

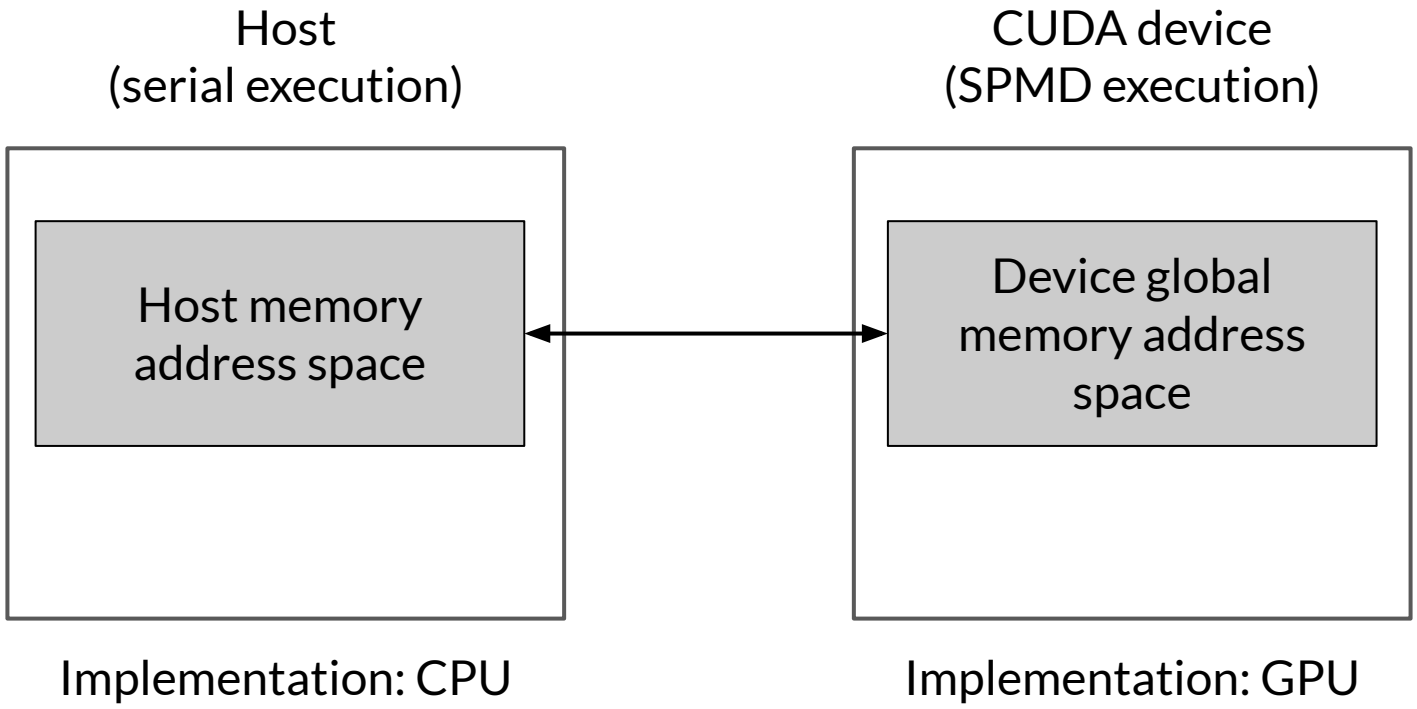
Host
(serial execution)

Implementation: CPU

CUDA device
(SPMD execution)

Implementation: GPU

CUDA Execution Model



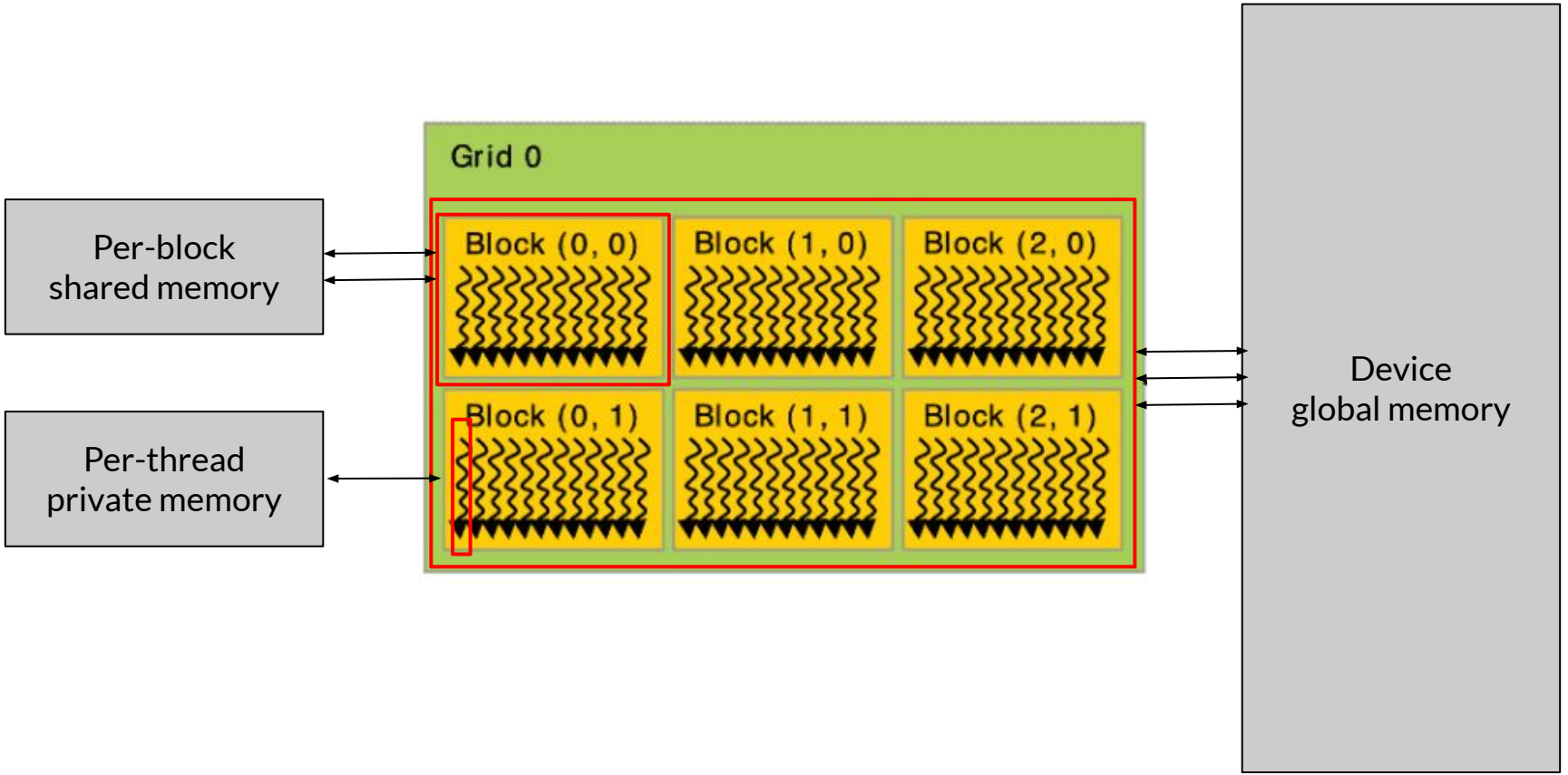
Memcpy Primitive

```
float* A = new float[N]; // allocate buffer in host mem

// populate host address space pointer A
for (int i=0 i<N; i++)
 A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA; // allocate buffer in
cudaMalloc(&deviceA, bytes); // device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

After kernel finishes execution, copy data back to CPU by

`cudaMemcpy(new_A, deviceA, bytes, cudaMemcpyDeviceToHost)`

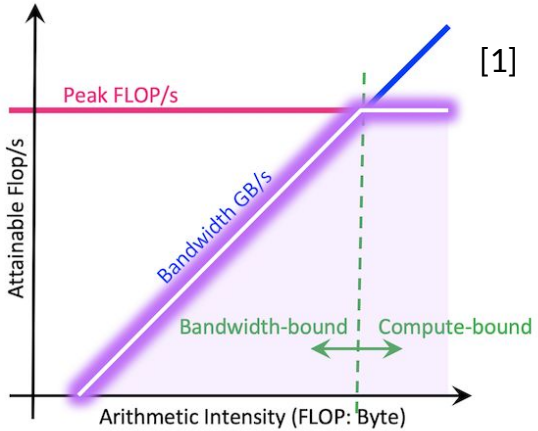
CUDA Device Memory Model



CUDA Synchronization Constructs

- __syncthreads()
  - Barrier: wait for all threads in the block to arrive at this point
- Atomic operations
  - e.g., float atomic add(float* addr, float amount)
  - Atomic operations on both global memory and shared memory var
- Host/device synchronization
  - Implicit barrier across all threads at return of kernel

Optimizing CUDA Programs



[1]

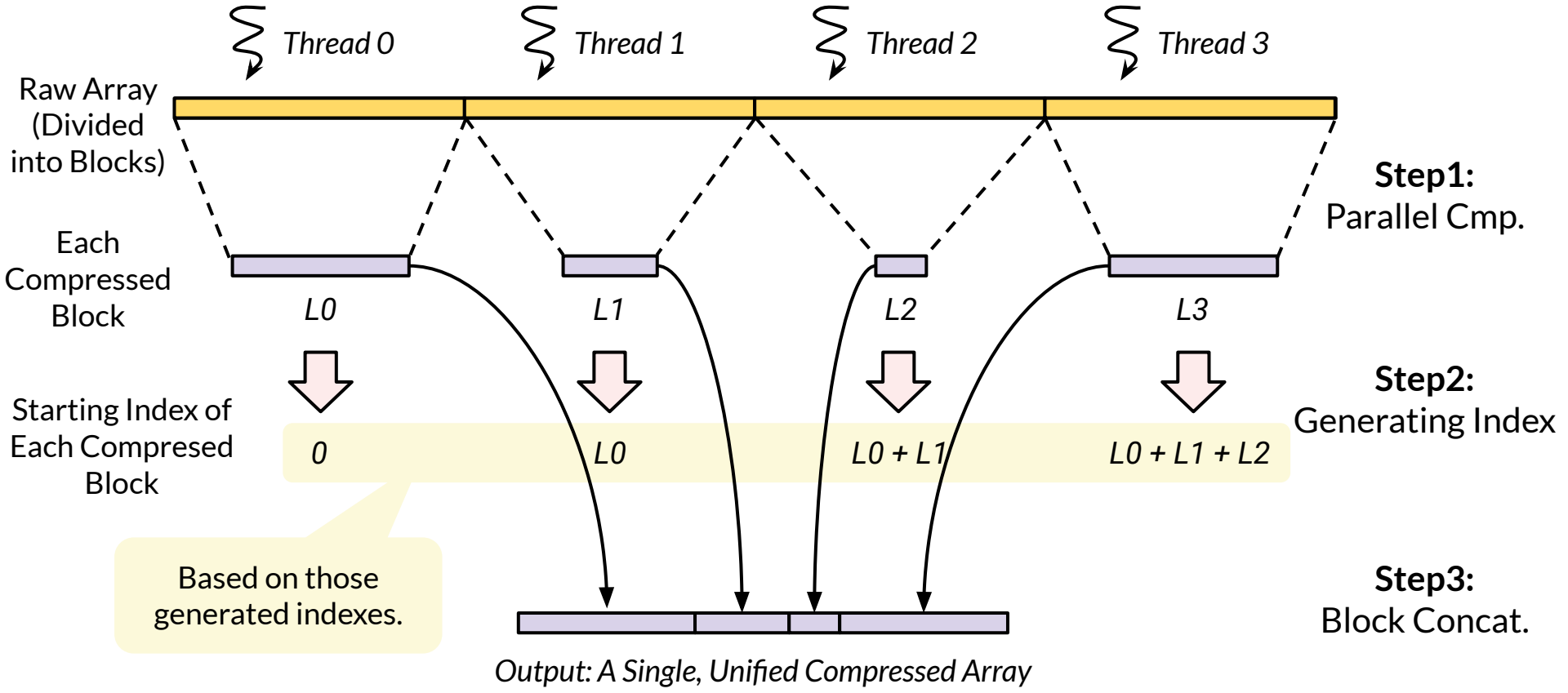
- Two major computation patterns for CUDA programs.
- Computation-bound Workloads
  - Dominated by arithmetic operations.
  - Limited by GPU's compute throughput.
  - Example: General Matrix Multiplication (GEMM).
- Memory-bound Workloads
  - Dominated by memory accesses or synchronization overhead.
  - Limited by HBM bandwidth or inter-thread coordination.
  - Example: Prefix-Sum (Scan Operation).

[1] https://docs.nersc.gov/tools/performance/roofline/

Optimizing a GPU Compression Kernel

- A memory-bound workload (irregular memory access, prefix-sum, etc).

Why GPU Compression Requires Prefix-Sum



Optimizing a GPU Compression Kernel

- A memory-bound workload (irregular memory access, prefix-sum, etc).
- Algorithm / Implementation:
  - Quantization
  - Fixed-length Encoding
  - Prefix-sum
  - Concatenation
- Let’s play with the code together!

CUDA Toolkit

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **Compiler Directives** — Ease of use
- **Programming Languages** — Maximum Flexibility

Development Environment

Nsight Systems

Nsight Compute

CUDA Profiling Tools Interface

CUDA-GDB Debugger

CUDA MEMCHECK

Language Support

C C++ Fortran python

Compile new languages to CUDA

CUDA Library

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library.

Learn More >

cuFFT

GPU-accelerated library for Fast Fourier Transform implementations.

Learn More >

cuRAND

GPU-accelerated random number generation.

Learn More >

cuSOLVER

GPU-accelerated dense and sparse direct solvers.

Learn More >

cuSPARSE

GPU-accelerated BLAS for sparse matrices.

Learn More >

cuTENSOR

GPU-accelerated tensor linear algebra library.

Learn More >

cuDSS

GPU-accelerated direct sparse solver library.

Learn More >

CUDA Math API

GPU-accelerated standard mathematical function APIs.

Learn More >

AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods.

Learn More >

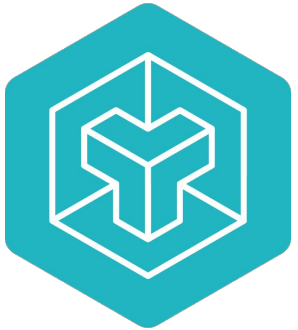
Useful Links

- Memory-bound Workload: Optimizations See Course Materials
- Computation-bound Workload: Optimizing GEMM from Scratch [Link]
- Modern GPU Programming For MLSys [Link]
- CUTLASS and CuTe DSL [Link] and [Code]

- For fun: https://www.cgdirector.com/nvidia-graphics-cards-order-performance/

Triton Basics

- Triton: A Python-based language and compiler for GPU programming.
  - Open-source, developed and popularized by OpenAI.
  - Specifically targeted for deep learning kernels.
  - Python-like syntax designed for high-performance.



Why Triton?

- Suppose PyTorch does not have exactly the kernel we want.
- One option is to write CUDA.
    - But CUDA exposes threads, warps, shared memory, synchronization, etc.
- Triton asks: can we express the computation at the **tile/block level**, and let the compiler handle more low-level details?

- Triton does not remove GPU optimization; it changes the level at which programmers express it.

vectorAdd: CUDA vs Triton

- CUDA: program one thread at a time. (left)
- Triton: program one tile at a time. (right)

n = 1000
BLOCK_SIZE = 256

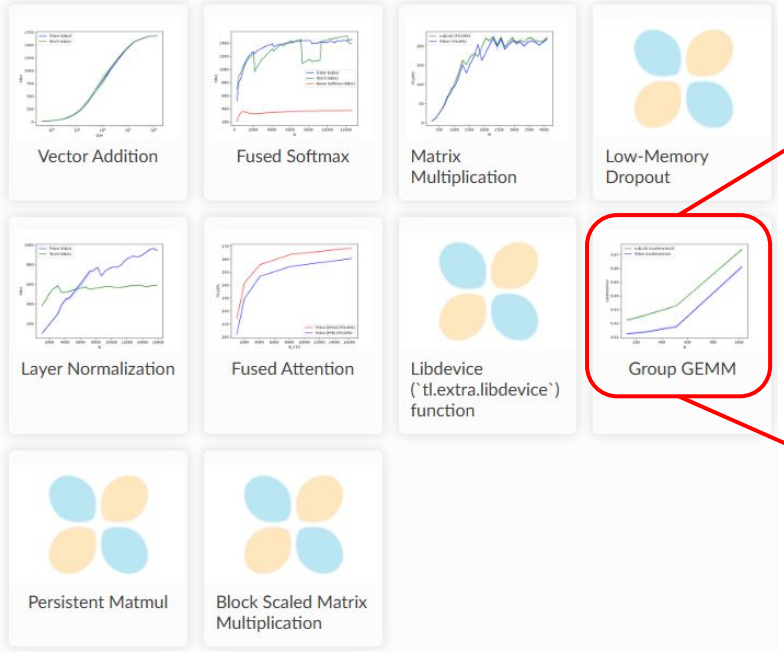
Program 0: elements 0 ~ 255
Program 1: elements 256 ~ 511
Program 2: elements 512 ~ 767
Program 3: elements 768 ~ 999

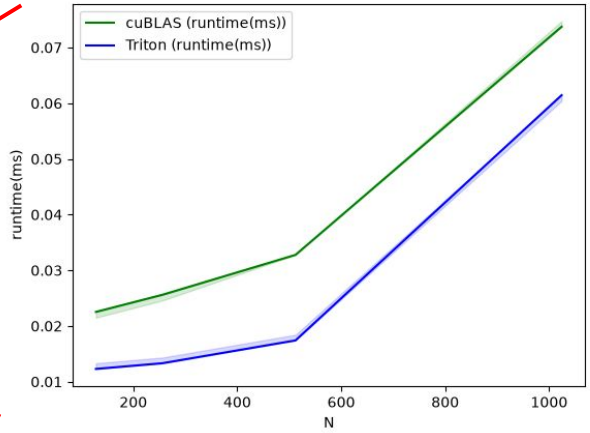
```
__global__ void vectorAdd(
      float* x, float* y, float* z, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
        z[i] = x[i] + y[i];
}
```

```
import triton.language as tl

pid = tl.program_id(0)

offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)

mask = offsets < n

x = tl.load(x_ptr + offsets, mask=mask)
y = tl.load(y_ptr + offsets, mask=mask)

z = x + y

tl.store(z_ptr + offsets, z, mask=mask)
```

Triton Tutorial

- https://triton-lang.org/main/getting-started/tutorials/index.html





Acknowledgement

- Parts of these slides were adapted from and inspired by materials from CMU 15-418/15-618: Parallel Computer Architecture and Programming, particularly the lecture on GPU architecture. I gratefully acknowledge the course instructors and authors of these materials.