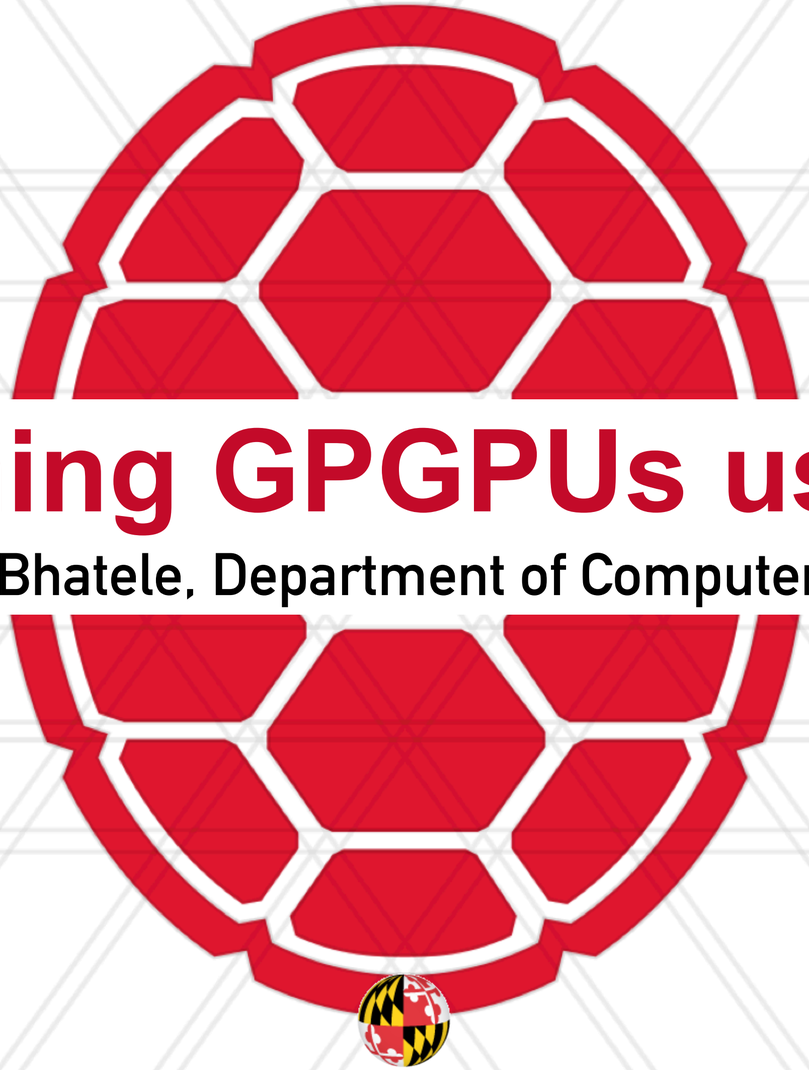
Systems for Machine Learning (CMSC828G)



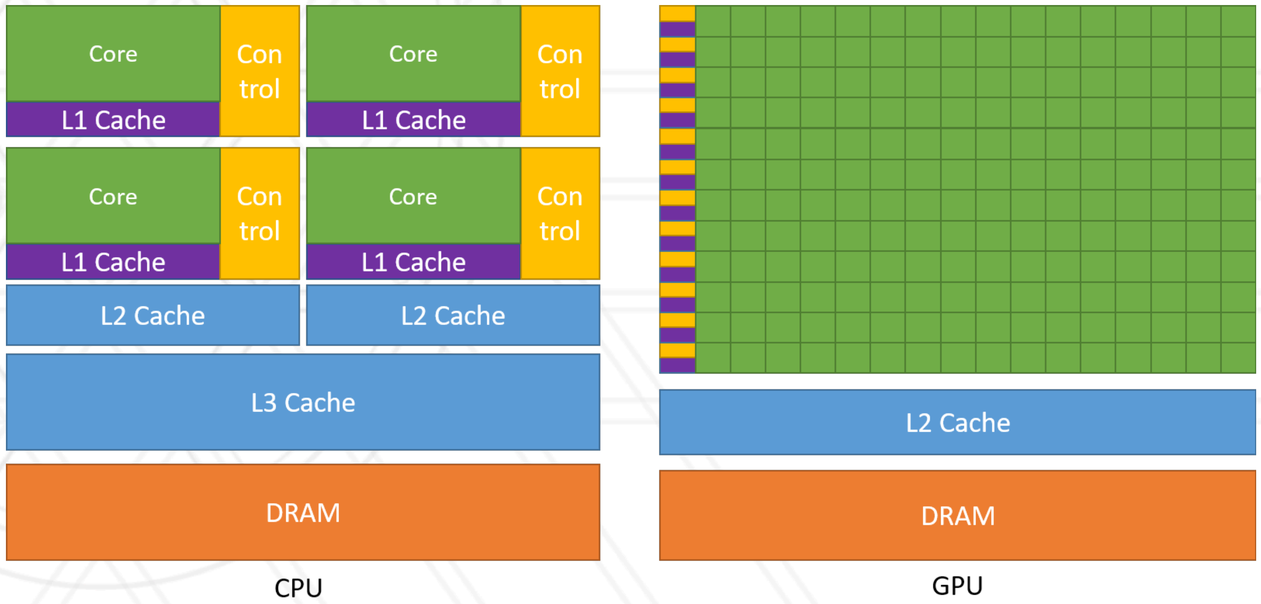
Programming GPGPUs using Triton

Abhinav Bhatele, Department of Computer Science

UNIVERSITY OF MARYLAND

CPUs vs GPUs

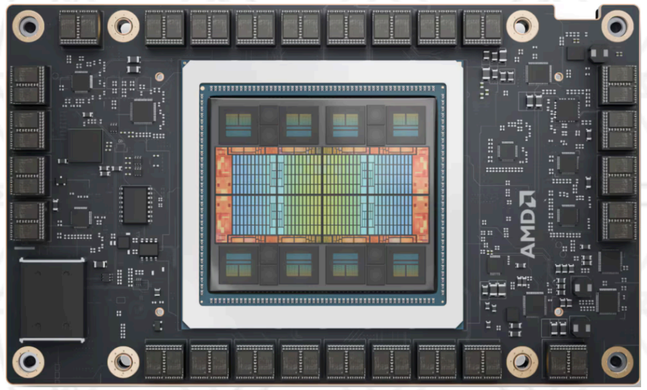
- GPUs: Many more "slower" cores
- Much higher instruction throughput

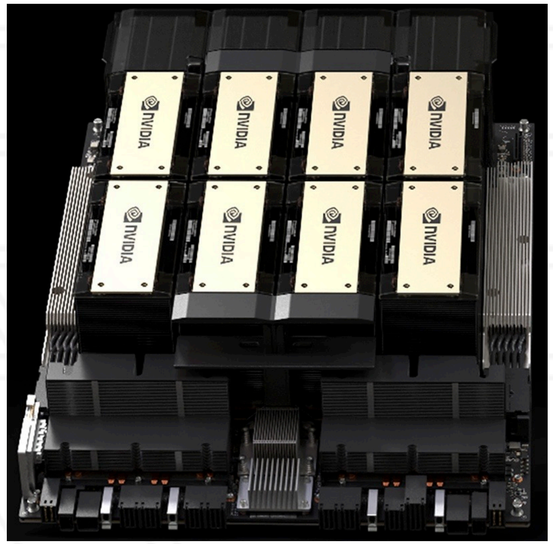


https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Some performance numbers

- AMD EPYC 9965
  - 192 cores (384 threads)
  - BF16 TFlop/s: 91
- AMD MI325X
  - 19,456 cores
  - BF16 TFlop/s: 1307
- NVIDIA H200
  - 18,000+ cores
  - BF16 TFlop/s: 989

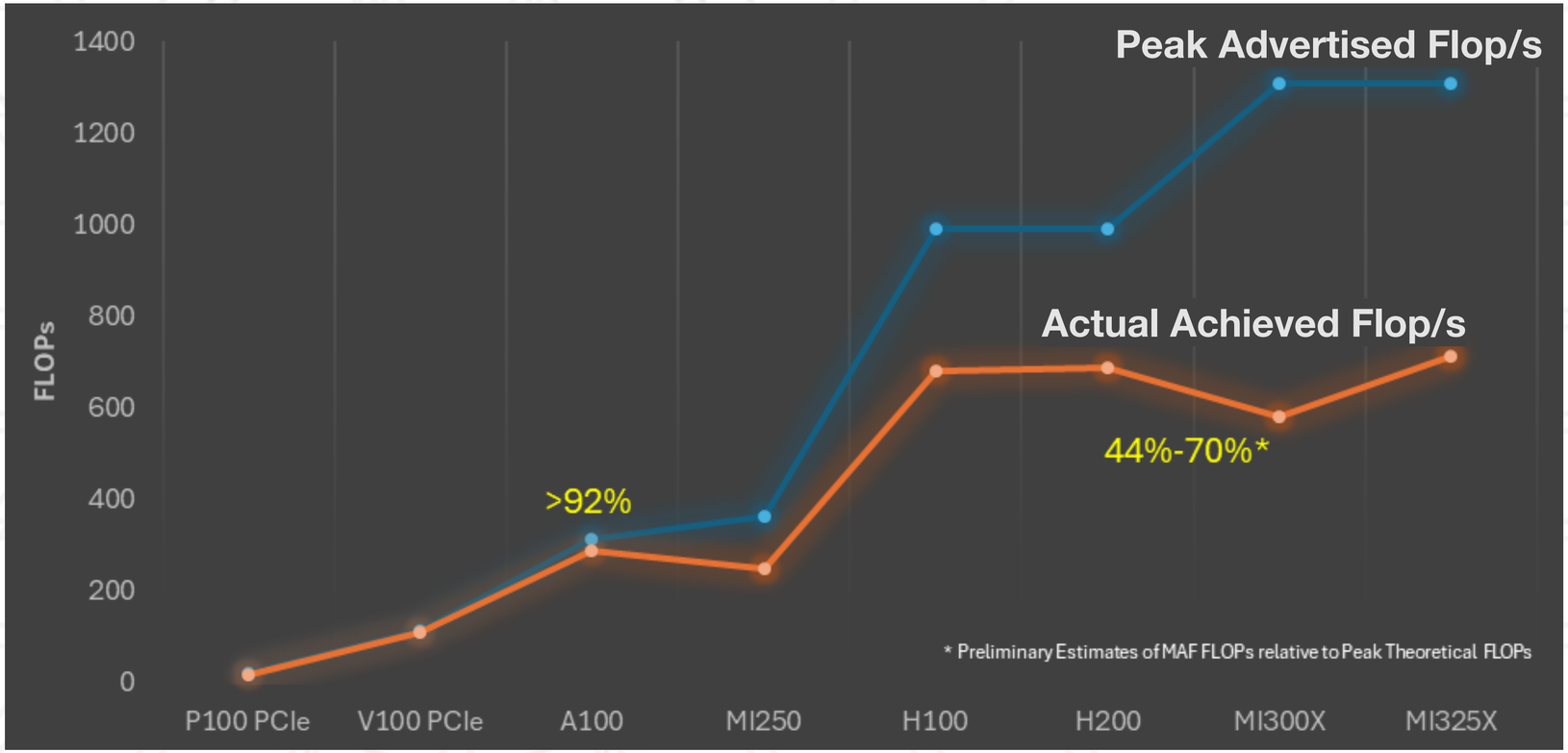




https://rocm.blogs.amd.com/software-tools-optimization/Understanding_Peak_and_Max-Achievable_FLOPS/README.html

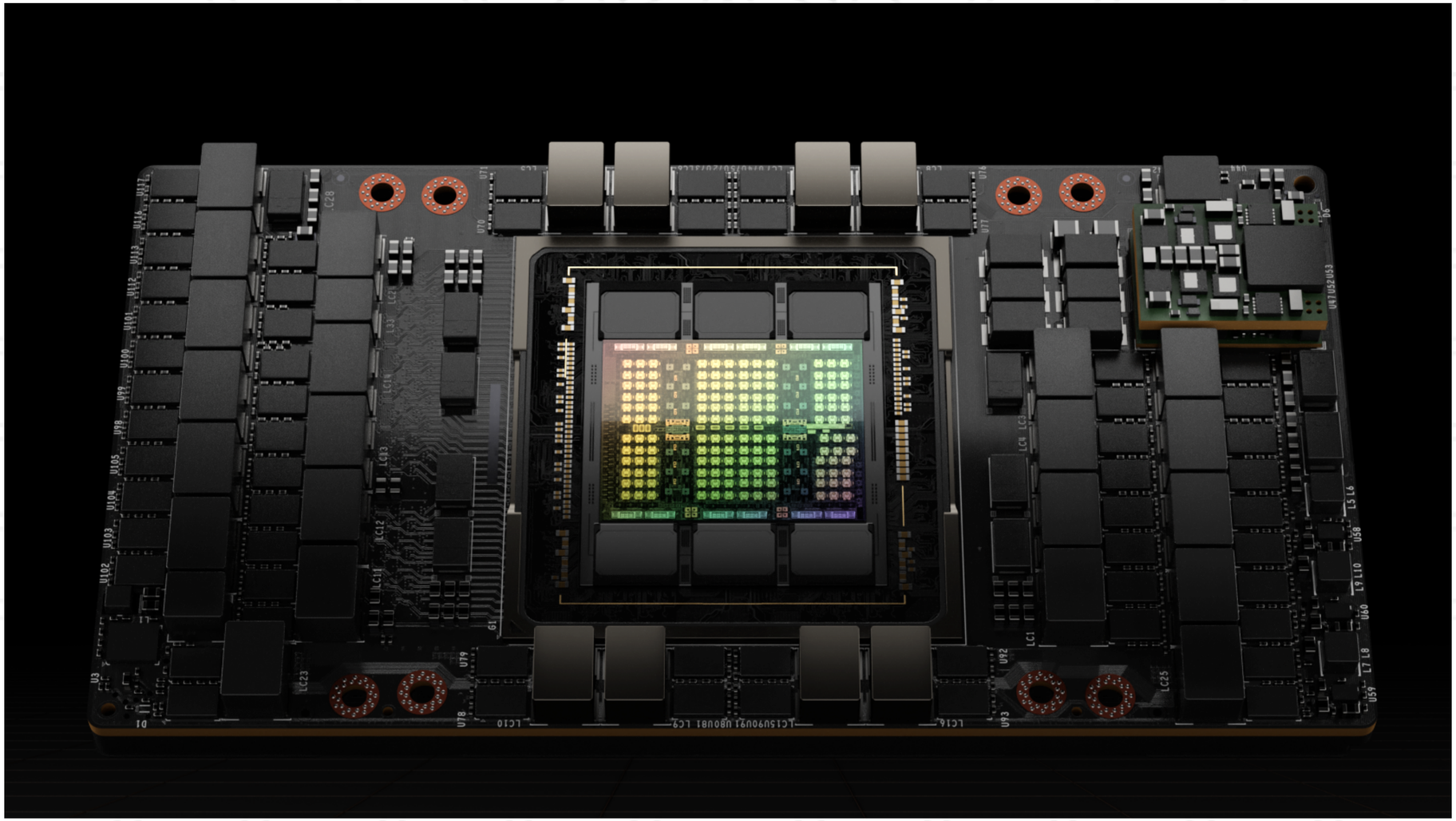
Some performance numbers

- AMD EPYC 9965
  - 192 cores (384 threads)
  - BF16 TFlop/s: 91
- AMD MI325X
  - 19,456 cores
  - BF16 TFlop/s: 1307
- NVIDIA H200
  - 18,000+ cores
  - BF16 TFlop/s: 989



https://rocm.blogs.amd.com/software-tools-optimization/Understanding_Peak_and_Max-Achievable_FLOPS/README.html

NVIDIA H100 chip



NVIDIA H100 chip

NVIDIA H100 chip

Hopper H100 SM

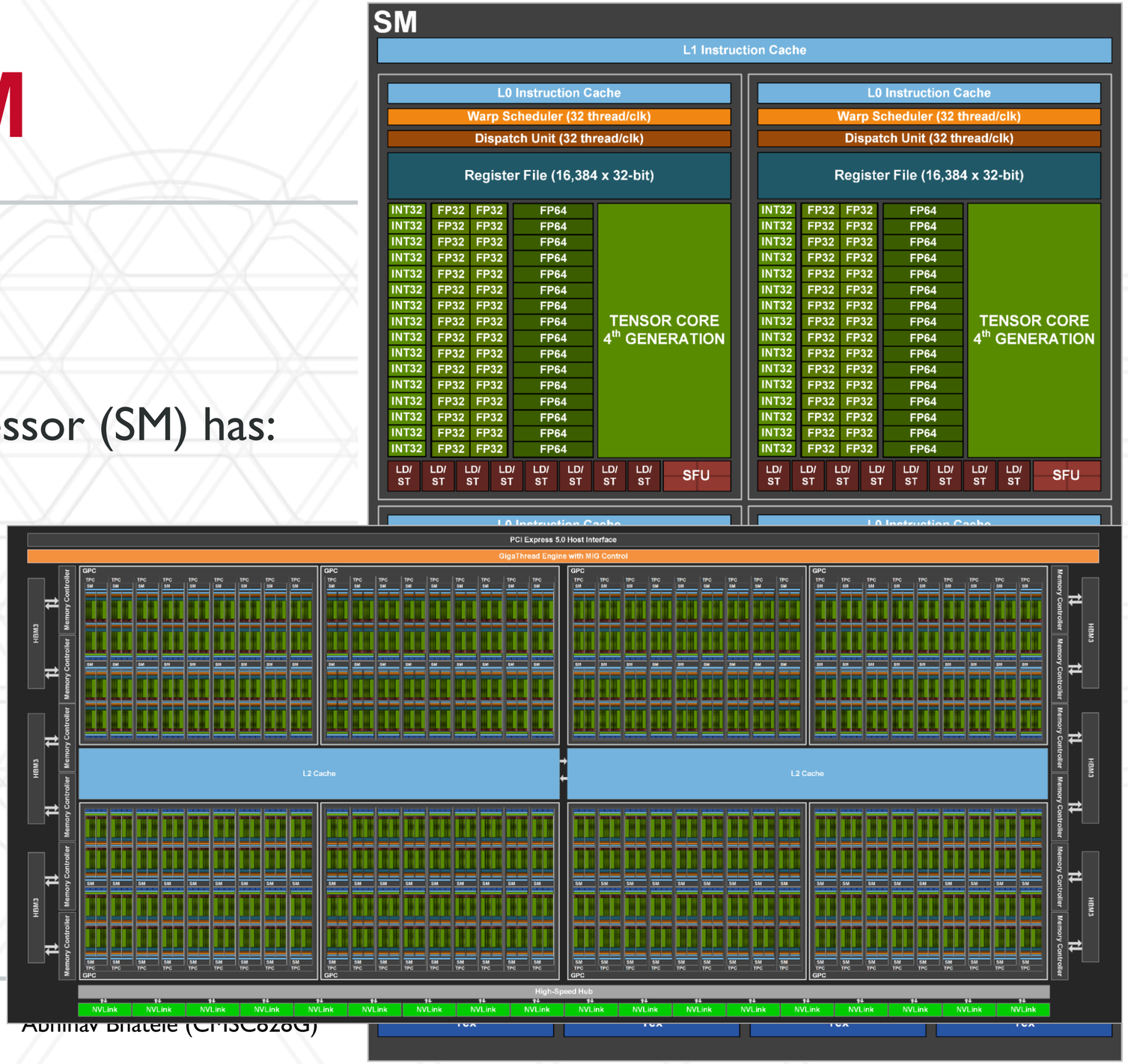
- CUDA Core
  - Single serial execution unit
- Each H100 Streaming Multiprocessor (SM) has:
  - Shared L1 cache
  - 128 FP32 cores, 64 FP64 cores
  - 64 INT32 cores
  - 84 Tensor cores
- CUDA capable device or GPU
  - Collection of SMs

Hopper H100 SM

- CUDA Core
  - Single serial execution unit
- Each H100 Streaming Multiprocessor (SM) has:
  - Shared L1 cache
  - 128 FP32 cores, 64 FP64 cores
  - 64 INT32 cores
  - 84 Tensor cores
- CUDA capable device or GPU
  - Collection of SMs

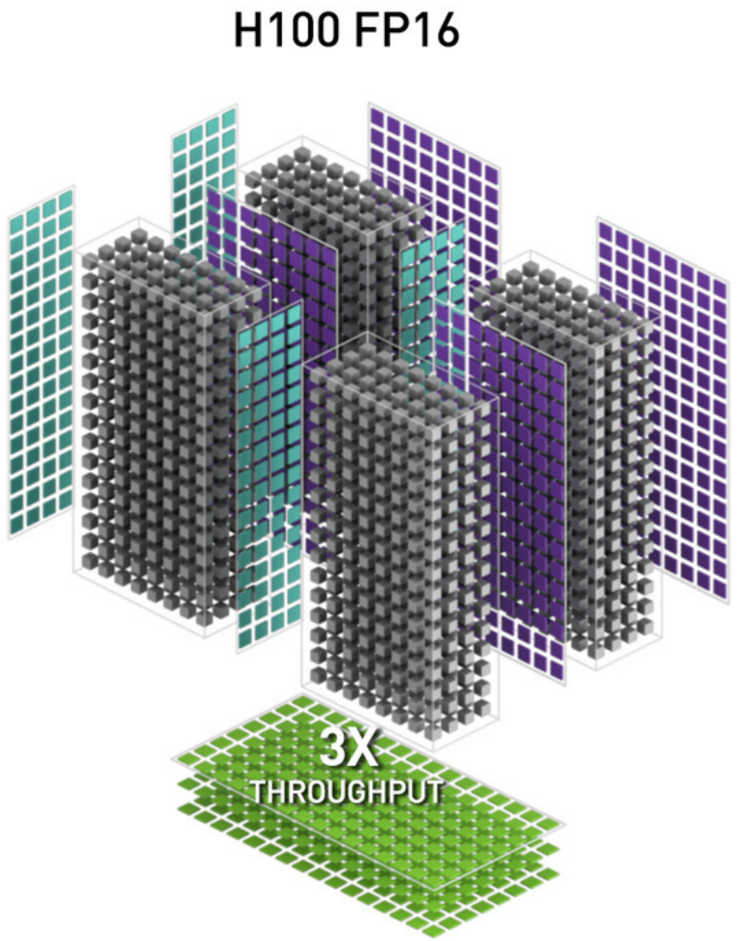
Hopper H100 SM

- CUDA Core
  - Single serial execution unit
- Each H100 Streaming Multiprocessor (SM) has:
  - Shared L1 cache
  - 128 FP32 cores, 64 FP64 cores
  - 64 INT32 cores
  - 84 Tensor cores
- CUDA capable device or GPU
  - Collection of SMs



H100 tensor cores

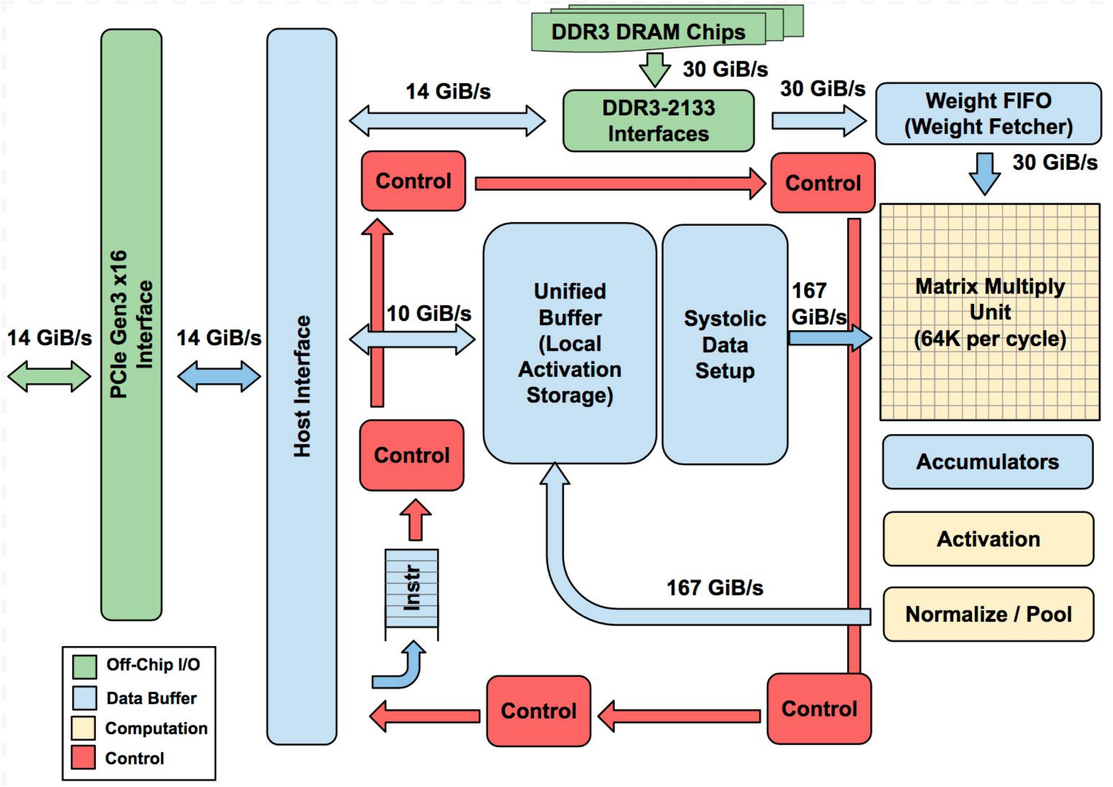
- Tensor cores are specialized cores for matrix multiply accumulate operations
- Operate in parallel across all SMs
- Multiply two 4 x 4 FP16 matrices and add to a 4 x 4 FP16 or FP32 matrix
- Mixed precision



https://resources.nvidia.com/en-us-tensor-core

Google's Tensor Processing Unit

- TPU is an ASIC (Application-specific Integrated Circuit)
- Co-processor just like GPUs
- Each TPU can have one or multiple MMUs
- TPU Pod is a collection of TPUs

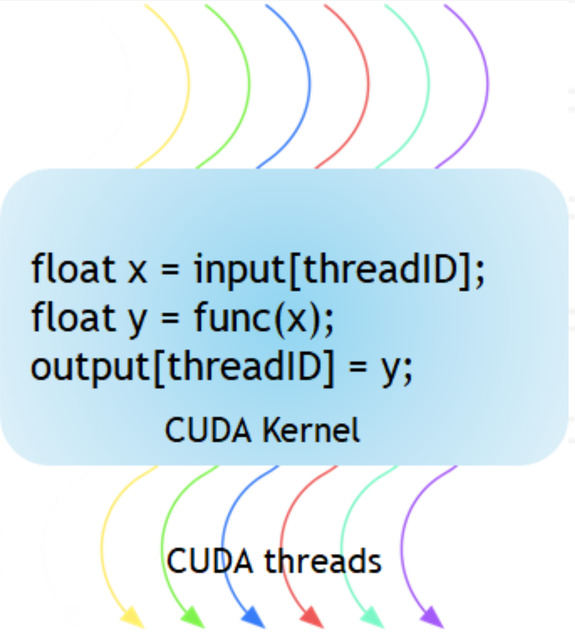


https://arxiv.org/pdf/1704.04760

http://web.cecs.pdx.edu/~mperkows/temp/May22/0020.Matrix-multiplication-systolic.pdf

CUDA: A programming model for NVIDIA GPUs

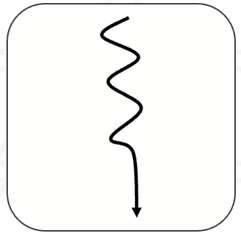
- Allows developers to use C++ as a high-level programming language
  - CUDA is a language extension
- Built around threads, blocks and grids
- Terminology:
  - Host: CPU
  - Device: GPU
  - CUDA kernel: a function that gets executed on the GPU



CUDA software abstraction

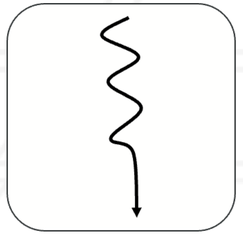
CUDA software abstraction

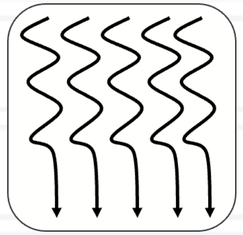
- Thread
  - Serial unit of execution



CUDA software abstraction

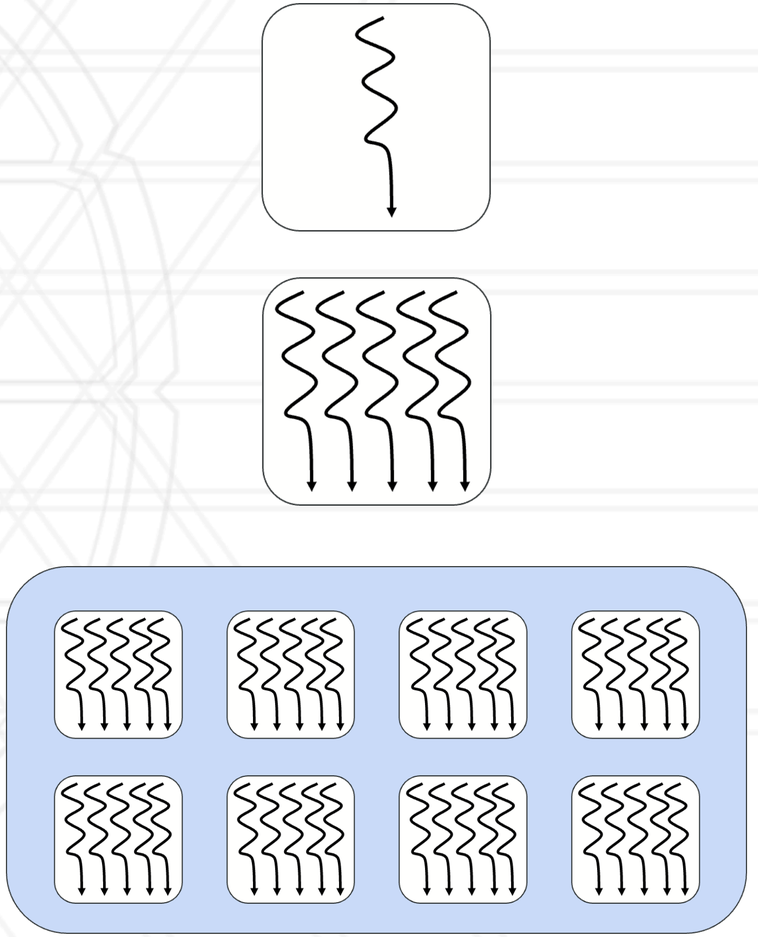
- Thread
  - Serial unit of execution
- Block
  - Collection of concurrent threads
  - Number of threads in block <= 1024





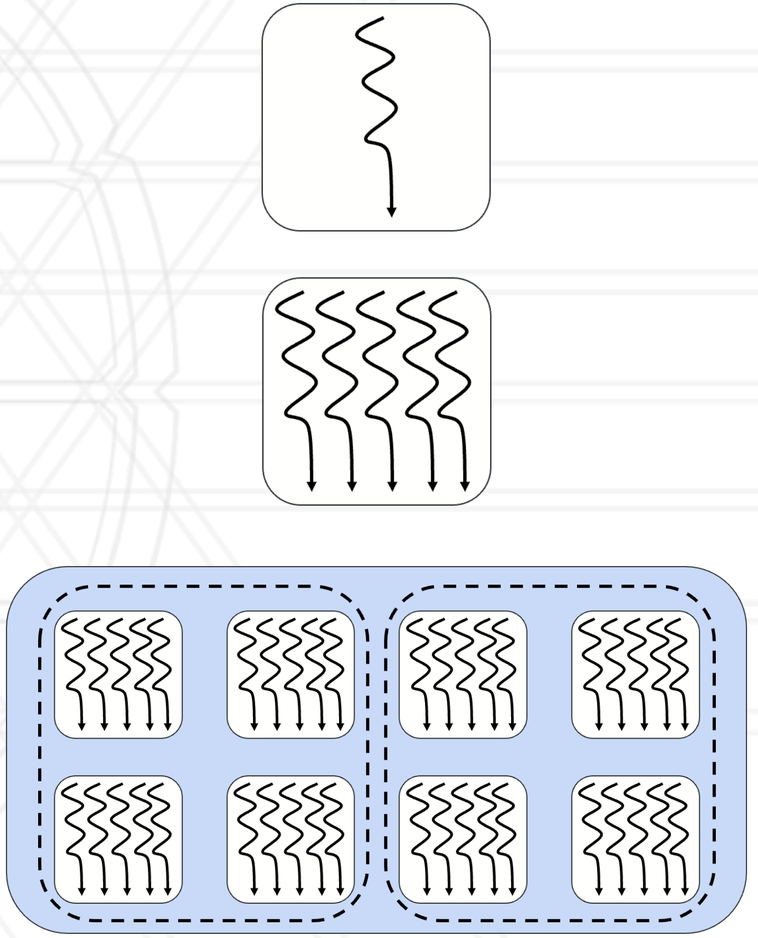
CUDA software abstraction

- Thread
  - Serial unit of execution
- Block
  - Collection of concurrent threads
  - Number of threads in block <= 1024
- Grid
  - Collection of blocks

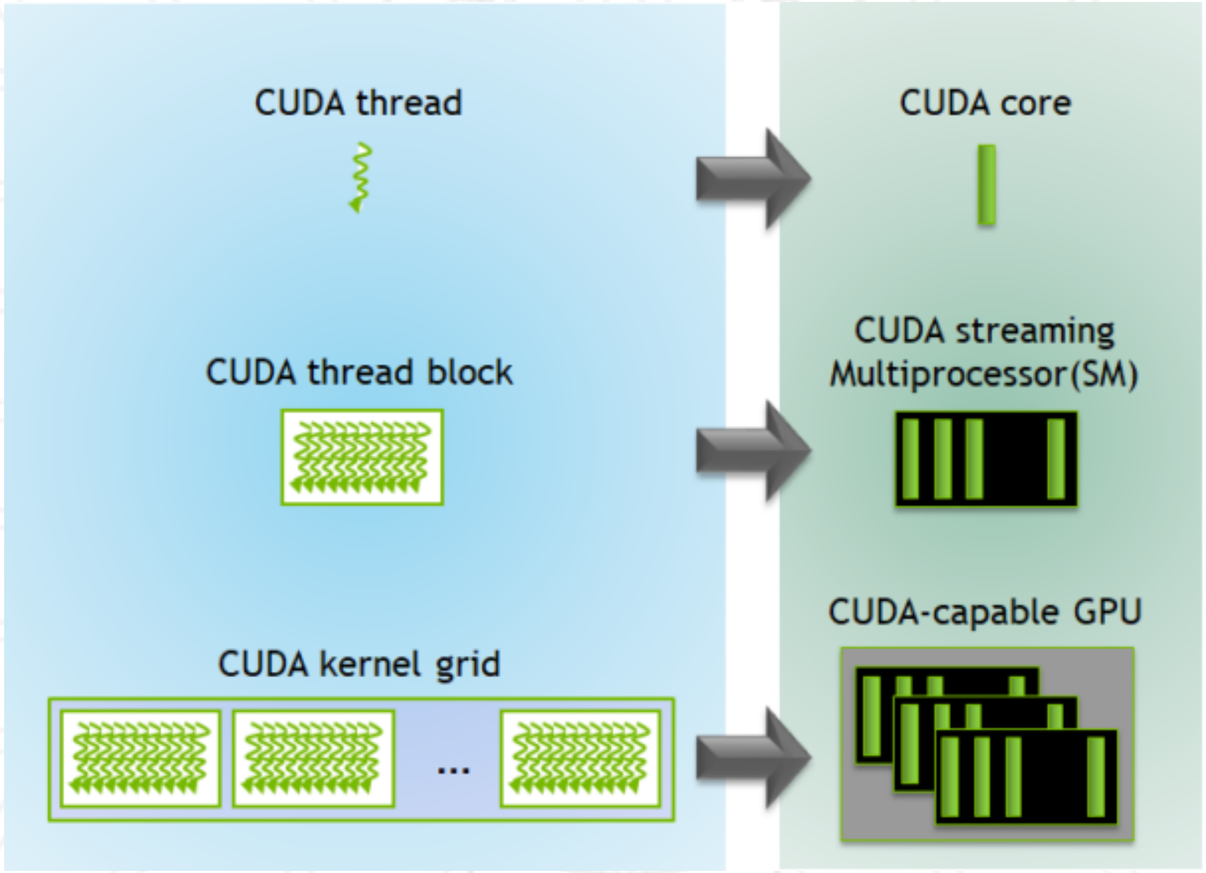


CUDA software abstraction

- Thread
  - Serial unit of execution
- Block
  - Collection of concurrent threads
  - Number of threads in block <= 1024
- Grid
  - Collection of blocks
- Block cluster
  - Groups of blocks within a grid



Software to hardware mapping



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Copying data to the GPU

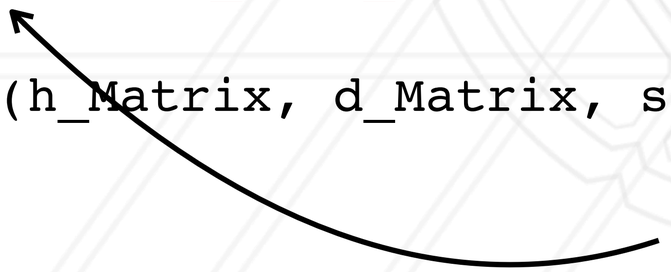
```
double *h_Matrix, *d_Matrix;
h_Matrix = new double[N];

cudaMalloc(&d_Matrix, sizeof(double)*N);

// ... initialize h_Matrix ...

cudaMemcpy(d_Matrix, h_Matrix, sizeof(double)*N, cudaMemcpyHostToDevice);

// saxpy<<<1, N>>>(d_x, d_y, alpha);

cudaMemcpy(h_Matrix, d_Matrix, sizeof(double)*N, cudaMemcpyDeviceToHost);

cudaFree(d_Matrix);
```

saxpy kernel in CUDA

```
__global__ void saxpy(float *x, float *y, float alpha) {
  int i = threadIdx.x;
  y[i] = alpha*x[i] + y[i];
}


int main() {
  ...
  cudaMemcpy(d_Matrix, h_Matrix, sizeof(double)*N, cudaMemcpyHostToDevice);

  saxpy<<<1, N>>>(d_x, d_y, alpha);

  cudaMemcpy(h_Matrix, d_Matrix, sizeof(double)*N, cudaMemcpyDeviceToHost);

   ...
}
```

saxpy kernel in CUDA

```
__global__ void saxpy(float *x, float *y, float alpha) {
  int i = threadIdx.x;
  y[i] = alpha*x[i] + y[i];
}


int main() {
  ...
  cudaMemcpy(d_Matrix, h_Matrix, sizeof(double)*N, cudaMemcpyHostToDevice);

  saxpy<<<1, N>>>(d_x, d_y, alpha);

  cudaMemcpy(h_Matrix, d_Matrix, sizeof(double)*N, cudaMemcpyDeviceToHost);

   ...
}
```



<<<#blocks, threads_per_block>>>

Grid size, Block size

saxpy kernel in CUDA

```
__global__ void saxpy(float *x, float *y, float alpha) {
  int i = threadIdx.x;
  y[i] = alpha*x[i] + y[i];
}


int main() {
  ...
  cudaMemcpy(d_Matrix, h_Matrix, sizeof(double)*N, cudaMemcpyHostToDevice);

  saxpy<<<1, N>>>(d_x, d_y, alpha);

  cudaMemcpy(h_Matrix, d_Matrix, sizeof(double)*N, cudaMemcpyDeviceToHost);

   ...
}
```

What happens when:
array size (N) > 1024?

<<<#blocks, threads_per_block>>>

Grid size, Block size

Memory hierarchy on GPU

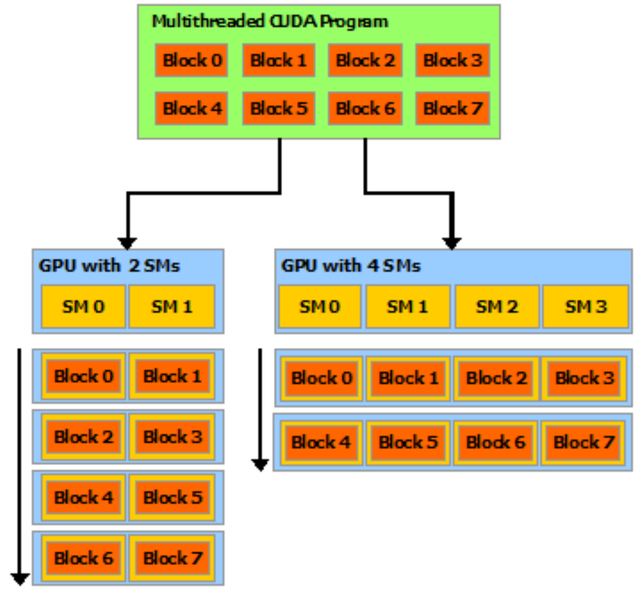
- Global memory: allocated using cudaMalloc
  - Off-chip DRAM (HBM/GDDR)
  - Shared by all threads
- Shared memory: Used via `__shared__` variables
  - On-chip SRAM per SM (shared with L1 cache)
  - Shared by threads in a block
- Local memory: not on chip (name is confusing)
  - Off-chip DRAM
  - Private memory per thread

Dealing with large arrays

```
__global__ void saxpy(float *x, float *y, float alpha, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        y[i] = alpha*x[i] + y[i];
}


int main() {
    ...
    int threadsPerBlock = 512;
    int numBlocks = N/threadsPerBlock
                  + (N % threadsPerBlock != 0);

    saxpy<<<numBlocks, threadsPerBlock>>>(x, y, alpha, N);
    ...
}
```

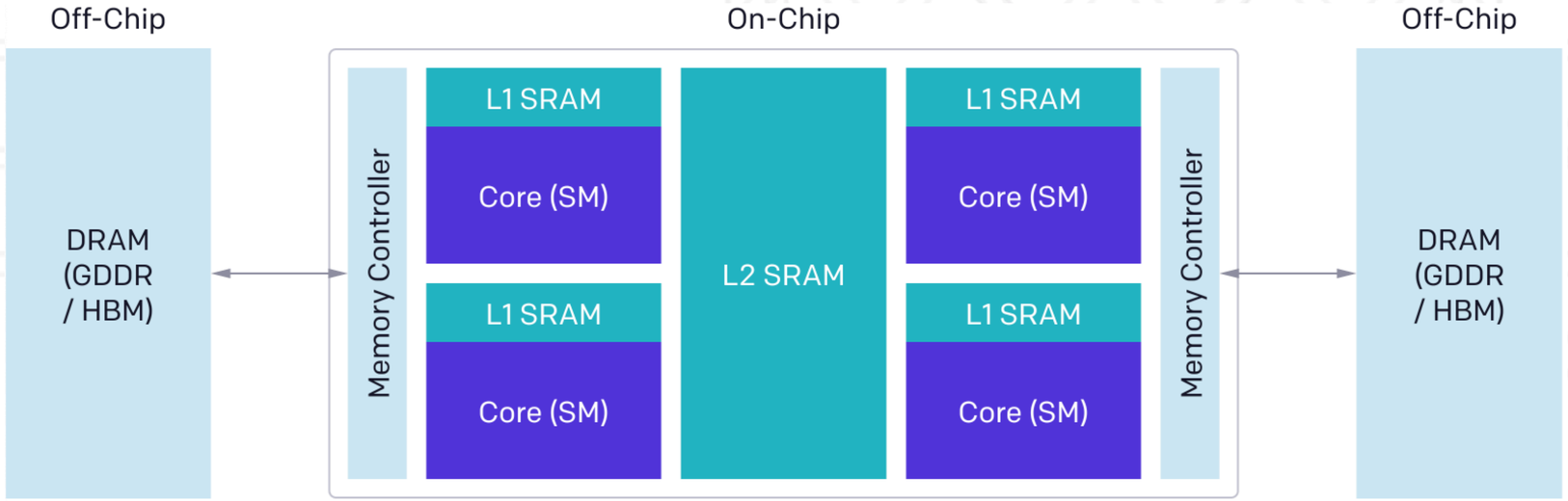
Dealing with large arrays

```
__global__ void saxpy(float *x, float *y, float alpha, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        y[i] = alpha*x[i] + y[i];
}


int main() {
    ...
    int threadsPerBlock = 512;
    int numBlocks = N/threadsPerBlock
                  + (N % threadsPerBlock != 0);

    saxpy<<<numBlocks, threadsPerBlock>>>(x, y, alpha, N);
    ...
}
```



Triton: Python-like library for GPU programming

- Open-source, developed by OpenAI
- Specifically targeted for neural networks
- Python-like syntax but supposed to be high performing



Triton's programming model

- User writes kernels as decorated Python functions
- These kernels are launched concurrently with different `program_id`'s on a grid of *instances*
  - A program instance is a block of work
- Within-instance parallelism is exposed via operations on *blocks* — small arrays whose dimensions are powers of 2

saxpy kernel in Triton

```
@triton.jit
def saxpy(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)

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

    mask = offsets < N

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

    y = alpha*x + y

    tl.store(y_ptr + offsets, y, mask=mask)
```

saxpy kernel in Triton

```
@triton.jit
def saxpy(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)

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

    mask = offsets < N

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

    y = alpha*x + y

    tl.store(y_ptr + offsets, y, mask=mask)
```

program instance

saxpy kernel in Triton

```
@triton.jit
def saxpy(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)

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

    mask = offsets < N

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

    y = alpha*x + y

    tl.store(y_ptr + offsets, y, mask=mask)
```

program instance

vectorized indexing

saxpy kernel in Triton

```
@triton.jit
def saxpy(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)

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

    mask = offsets < N

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

    y = alpha*x + y

    tl.store(y_ptr + offsets, y, mask=mask)
```

program instance

vectorized indexing

explicit loads and stoers

saxpy kernel in Triton

Block size is typically chosen to be a power of 2

```
@triton.jit
def saxpy(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)

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

    mask = offsets < N

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

    y = alpha*x + y

    tl.store(y_ptr + offsets, y, mask=mask)
```

program instance

vectorized indexing

explicit loads and stoers

Launching the kernel

```
import torch

...

x = torch.randn(N, device="cuda", dtype=torch.float32)
y = torch.randn(N, device="cuda", dtype=torch.float32)

BLOCK_SIZE = 256
grid = ( (N + BLOCK_SIZE - 1) // BLOCK_SIZE, )

saxpy[grid](
    x, y, alpha, N, BLOCK_SIZE=BLOCK_SIZE
)
```

Launching the kernel

```
import torch

...

x = torch.randn(N, device="cuda", dtype=torch.float32)
y = torch.randn(N, device="cuda", dtype=torch.float32)

BLOCK_SIZE = 256
grid = ( (N + BLOCK_SIZE - 1) // BLOCK_SIZE, )

saxpy[grid](
    x, y, alpha, N, BLOCK_SIZE=BLOCK_SIZE
)
```

grid is used to launch several program instances

Equivalence with CUDA

- Each Triton program instance is like a CUDA thread block
- Triton *lanes* (vector elements) are like CUDA threads
  - `tl.arange(0, BLOCK_SIZE)` -> Lane IDs

Under the hood

- @triton.jit kernels are first compiled to generate Triton-IR on-the-fly
- This is then simplified, optimized, and parallelized automatically before being converted to LLVM-IR
- Eventually converted to PTX/CUBIN for execution on GPUs
- Now supports AMD GPUs also

Softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}}$$

Softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}}$$

$$\sigma(z)_i = \frac{e^{\left(z_i - \max_k z_k\right)}}{\sum_{j=0}^{K-1} e^{\left(z_i - \max_k z_k\right)}}$$

Numerically stable

Softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=0}^{K-1} e^{z_j}}$$

$$\sigma(z)_i = \frac{e^{\left(z_i - \max_k z_k\right)}}{\sum_{j=0}^{K-1} e^{\left(z_i - \max_k z_k\right)}}$$

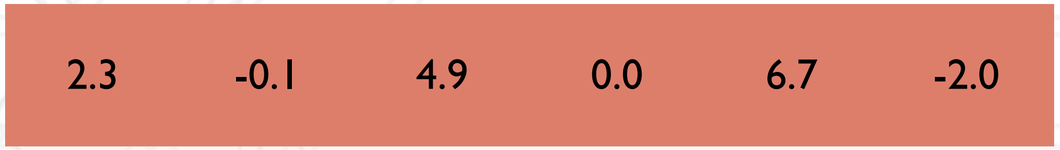
Numerically stable

compute the softmax of each row in the matrix

2.3	-0.1	4.9	0.0	6.7	-2.0
1.1	-0.3	0.0	-5.8	9.2	1.6
-3.0	2.4	6.4	-7.9	-8.1	0.2
-0.6	1.0	3.2	0.9	7.6	-6.0
5.4	4.2	2.0	8.3	7.5	3.0
9.9	7.4	-0.7	-6.3	3.1	8.4

Softmax kernel in Triton

```
@triton.jit
def softmax_kernel(X_ptr, Y_ptr, n_rows, n_cols, BLOCK_SIZE: tl.constexpr):
    row = tl.program_id(0)  # which row

    col_offsets = tl.arange(0, BLOCK_SIZE)
    mask = col_offsets < n_cols

    x_row_ptr = X_ptr + row * n_rows
    x = tl.load(x_row_ptr + cols_offsets, mask=mask, other=-float("inf"))

    z = x - tl.max(x, axis=0)

    numerator = tl.exp(z)
    denominator = tl.sum(numerator, axis=0)

    y = numerator / denominator

    y_row_ptr = Y_ptr + row * n_cols
    tl.store(y_row_ptr + col_offsets, y, mask=mask)
```



Softmax kernel in Triton

```
@triton.jit
def softmax_kernel(X_ptr, Y_ptr, n_rows, n_cols, BLOCK_SIZE: tl.constexpr):
    row = tl.program_id(0)  # which row

    col_offsets = tl.arange(0, BLOCK_SIZE)
    mask = col_offsets < n_cols

    x_row_ptr = X_ptr + row * n_rows
    x = tl.load(x_row_ptr + cols_offsets, mask=mask, other=-float("inf"))

    z = x - tl.max(x, axis=0)

    numerator = tl.exp(z)
    denominator = tl.sum(numerator, axis=0)

    y = numerator / denominator

    y_row_ptr = Y_ptr + row * n_cols
    tl.store(y_row_ptr + col_offsets, y, mask=mask)
```

reductions like max and sum are easy in Triton!

Performance parameters

- BLOCK_SIZE: controls amount of work per program instance
- num_warps: number of warps executed per program instance
  - Typical values: 2, 4, 8
- num_stages: number of pipeline stages used

Auto-tuning in Triton

```
@triton.autotune(
    configs=[
        triton.Config({"BLOCK_SIZE": 128}, num_warps=2),
        triton.Config({"BLOCK_SIZE": 256}, num_warps=4),
        triton.Config({"BLOCK_SIZE": 512}, num_warps=8),
    ],
    key=["N"],
)


@triton.jit
def saxpy_kernel(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
```

- Triton compiles one kernel per config, runs them all, and caches the fastest config
- triton.Config sets compile time constants and launch-time parameters

Auto-tuning in Triton

```
@triton.autotune(
    configs=[
        triton.Config({"BLOCK_SIZE": 128}, num_warps=2),
        triton.Config({"BLOCK_SIZE": 256}, num_warps=4),
        triton.Config({"BLOCK_SIZE": 512}, num_warps=8),
    ],
    key=["N"],
)
```

re-tune if N changes

```
@triton.jit
def saxpy_kernel(x_ptr, y_ptr, alpha, N, BLOCK_SIZE: tl.constexpr):
```

- Triton compiles one kernel per config, runs them all, and caches the fastest config
- triton.Config sets compile time constants and launch-time parameters

UNIVERSITY OF
MARYLAND