Introduction to Parallel Computing (CMSC416 / CMSC818X)

GPGPUs and CUDA

Abhinav Bhatele, Department of Computer Science

UNIVERSITY OF MARYLAND

Many slides borrowed from Daniel Nichols' slides

GPGPUs

- Originally developed to handle computation related to graphics processing
- Also found to be useful for scientific computing
- Hence the name: General Purpose Graphics Processing Unit

Accelerators

- IBM's Cell processors
  - Used in Sony's Playstation 3 (2006)
- GPUs: NVIDIA, AMD, Intel
  - First programmable GPU: GeForce 256
  - Around 1999-2001, early GPGPU results
- FPGAs

https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf

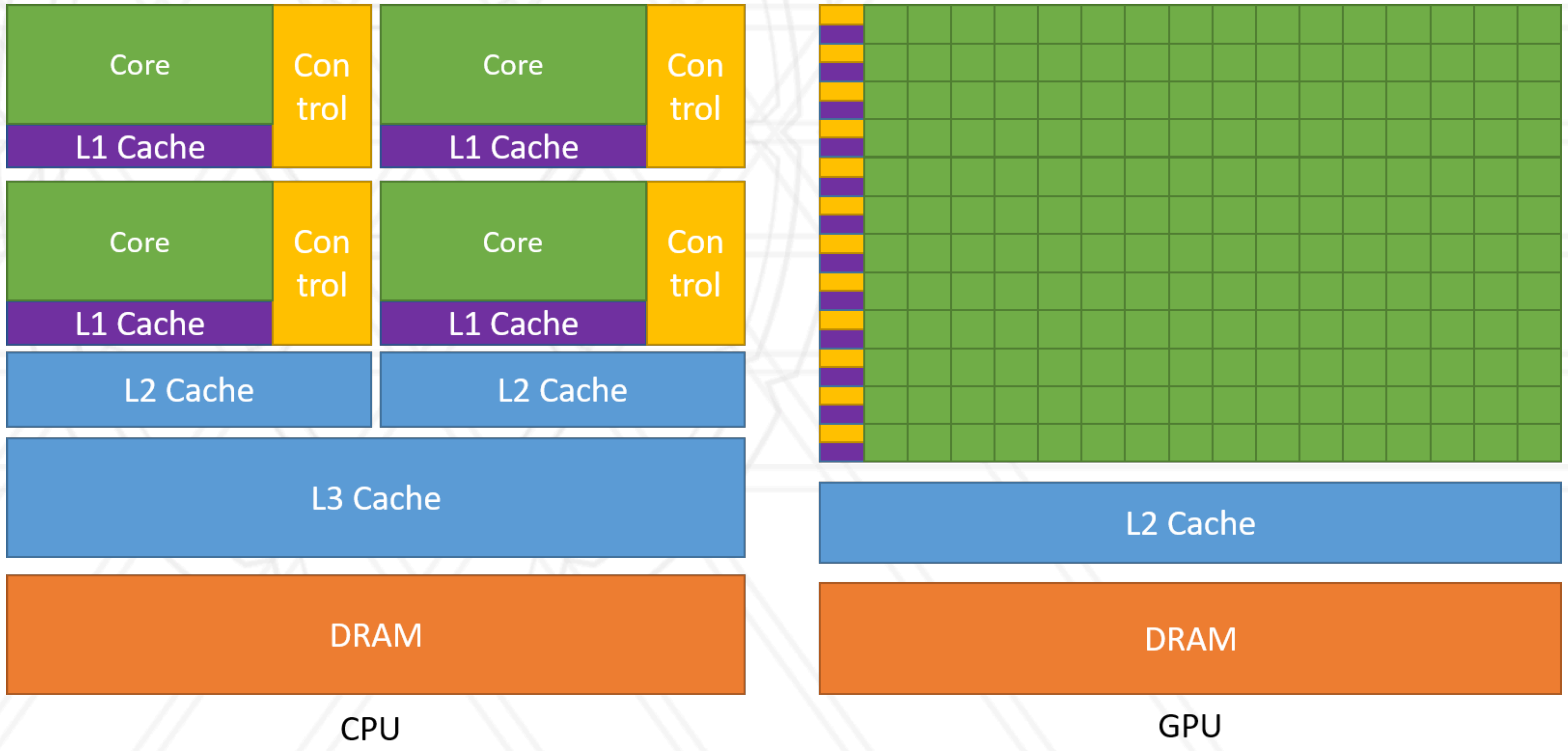
Used for mainstream HPC

- 2013: NAMD used for molecular dynamics simulations on a supercomputer with 3000 NVIDIA Tesla GPUs

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
THE HIV-1 CAPSID
Atomic structure of the AIDS pathogen's protein coat
PAGE 643
COSMOLOGY
THE FIRST LIGHT
In pursuit of the most distant galaxies
PAGE 554
CITATION
CROSSING THE BORDERS
International collaborations make the most impact
PAGE 557
ANTICANCER DRUGS
A SITTING TARGET
An indirect hit on 'undruggable' KRAS protein
PAGES 577 & 638
NATURE.COM/NATURE
30 May 2013

GPGPU Hardware

- Higher instruction throughput
- Hide memory access latencies with computation



Comparing GPUs to CPUs

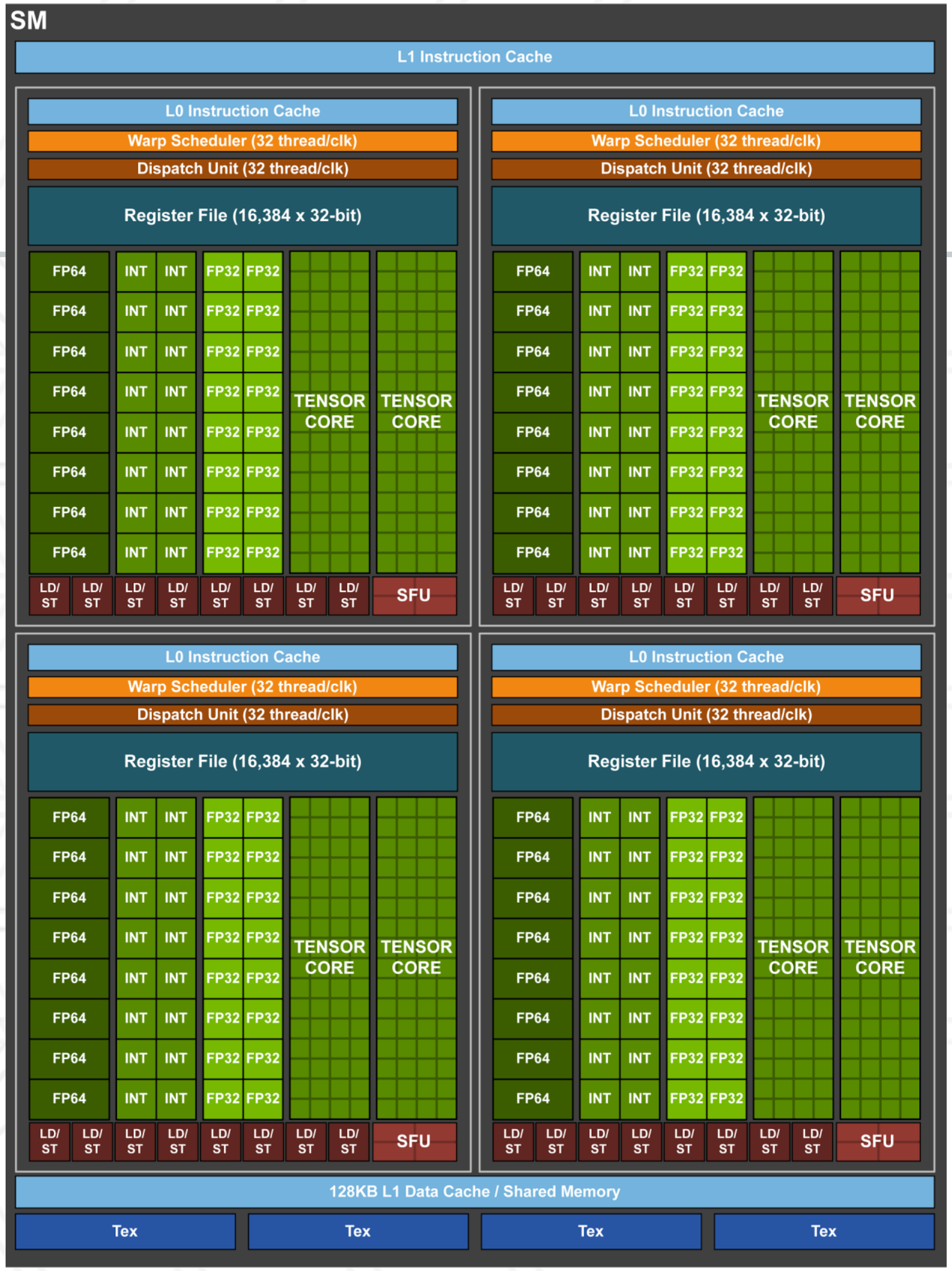
- Intel i9 11900K
  - 8 cores
  - 3.3 GHz
- AMD Epic 7763
  - 64 cores
  - 2.45 GHz

- NVIDIA GeForce RTX 3090
  - 10,496 cores
  - 1.4 GHz
- NVIDIA A100
  - 17,712 cores
  - 0.76 GHz

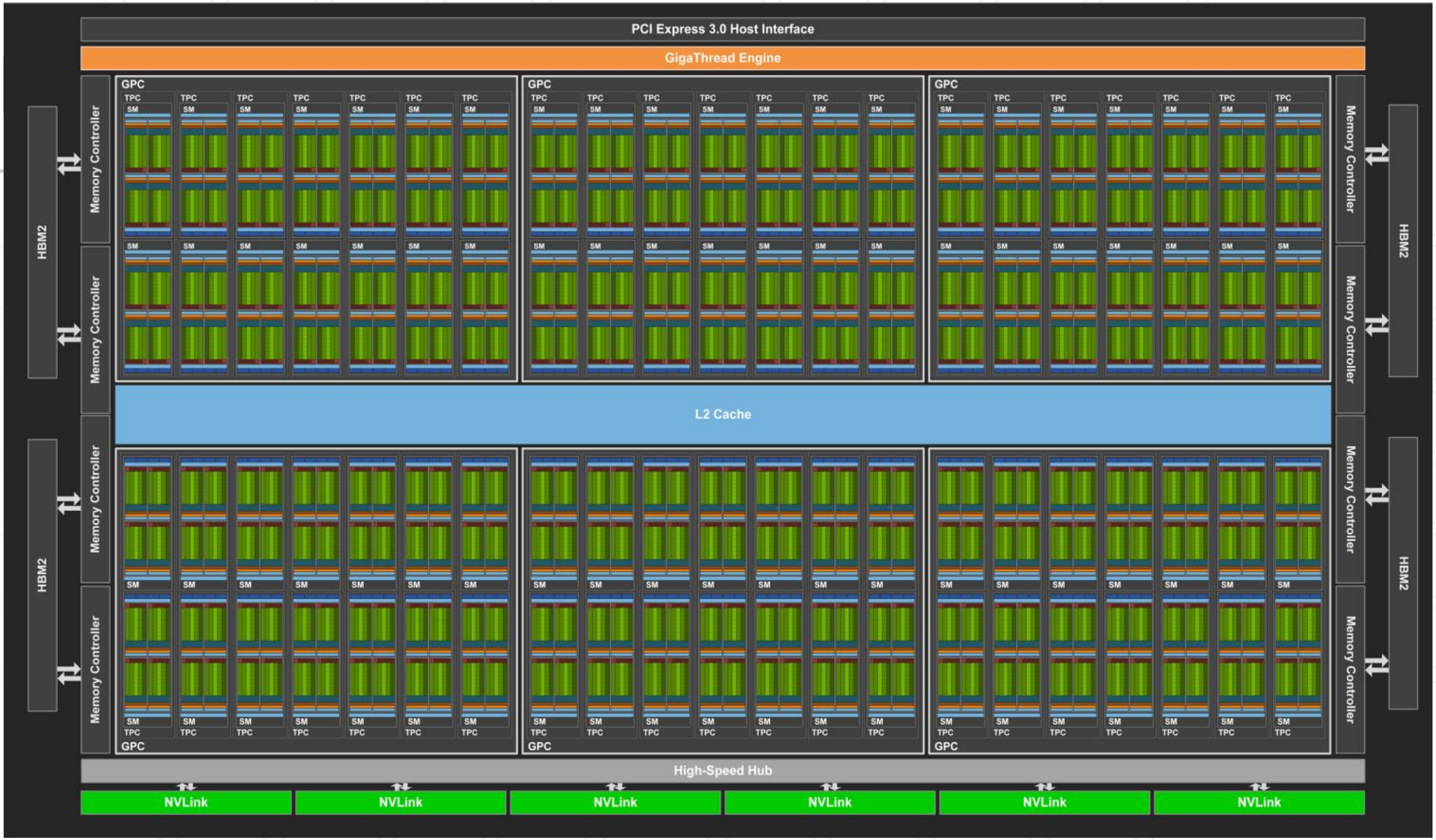
Volta GV100 SM

- CUDA Core
  - Single serial execution unit
- Each Volta Streaming Multiprocessor (SM) has:
  - 64 FP32 cores
  - 64 INT32 cores
  - 32 FP64 cores
  - 8 Tensor cores
- CUDA capable device or GPU
  - Collection of SMs

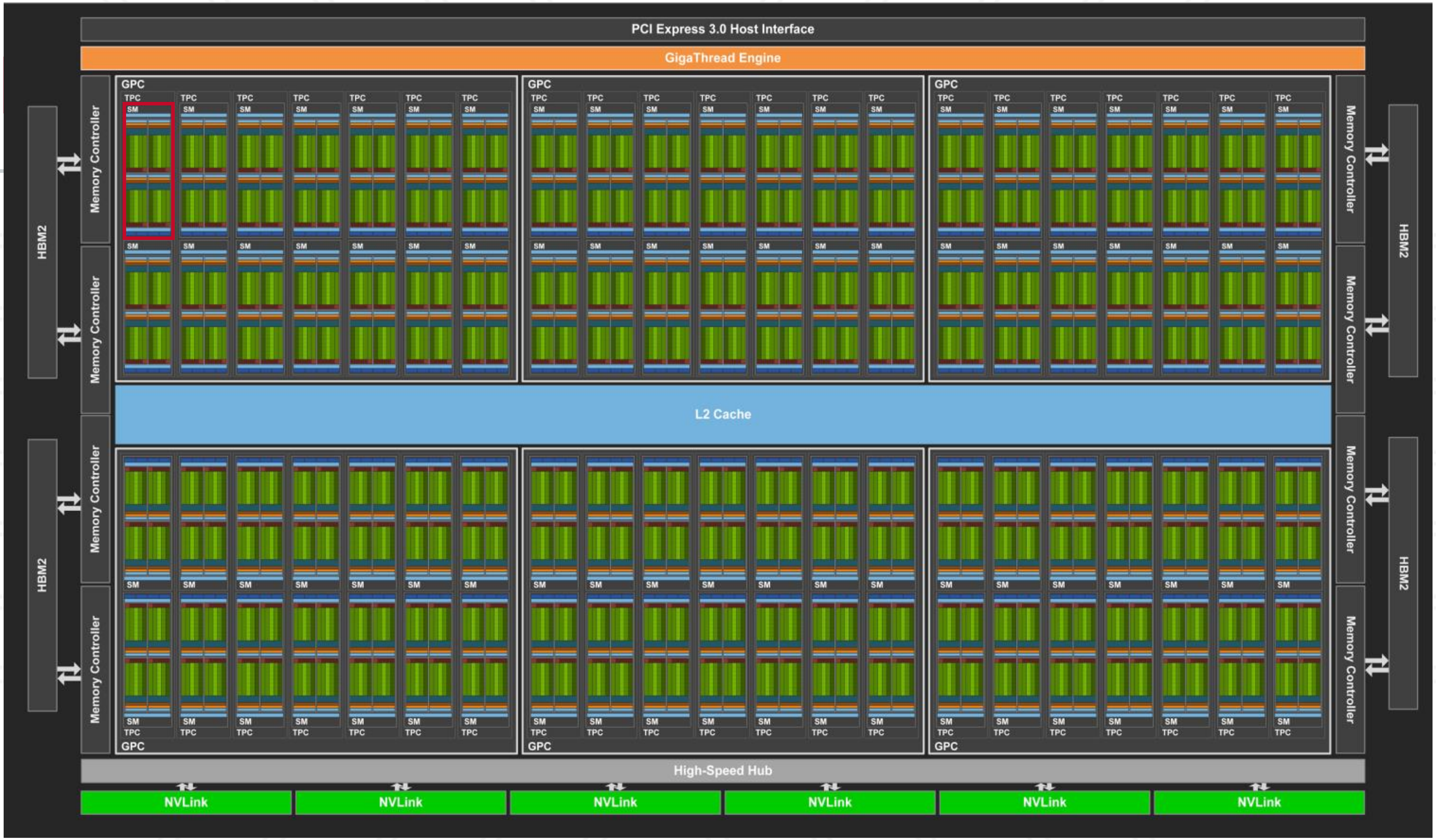
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Vo

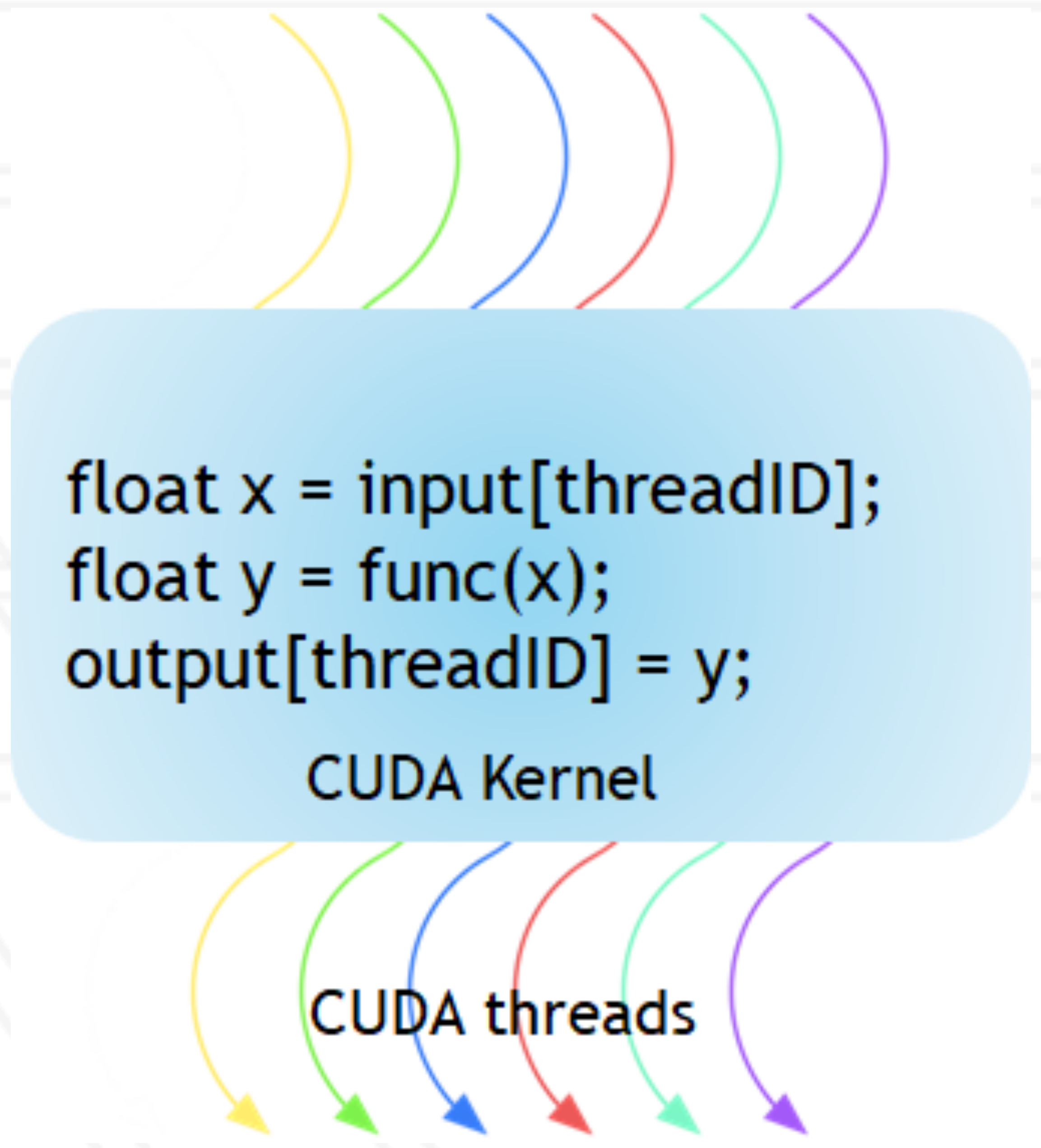

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Vo


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

CUDA: A programming model for NVIDIA GPUs

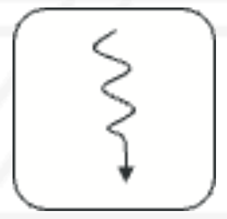
- Allows developers to use C++ as a high-level programming language
- Built around threads, blocks and grids
- Terminology:
  - Host: CPU
  - Device: GPU
  - CUDA kernel: a function that gets executed on the GPU



CUDA software abstraction

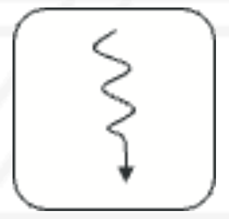
CUDA software abstraction

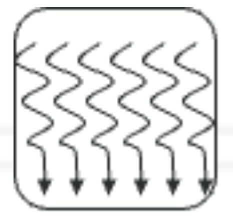
- Thread
  - Serial unit of execution



CUDA software abstraction

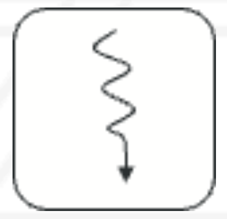
- Thread
  - Serial unit of execution
- Block
  - Collection of threads
  - Number of threads in block <= 1024

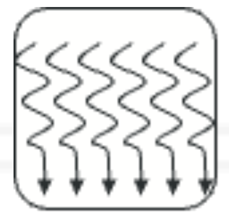


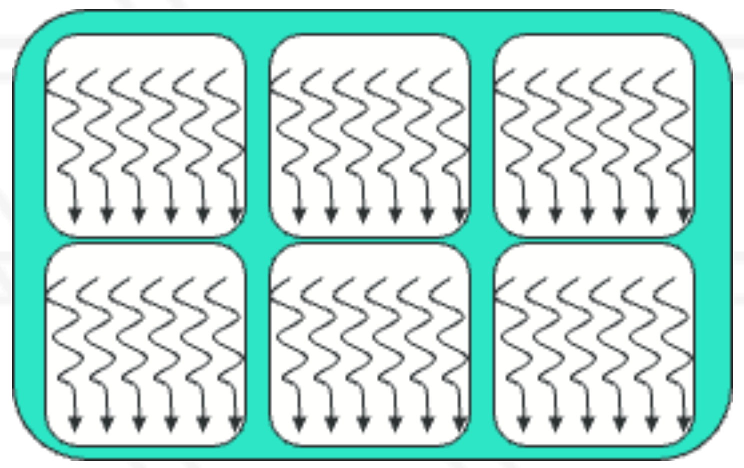


CUDA software abstraction

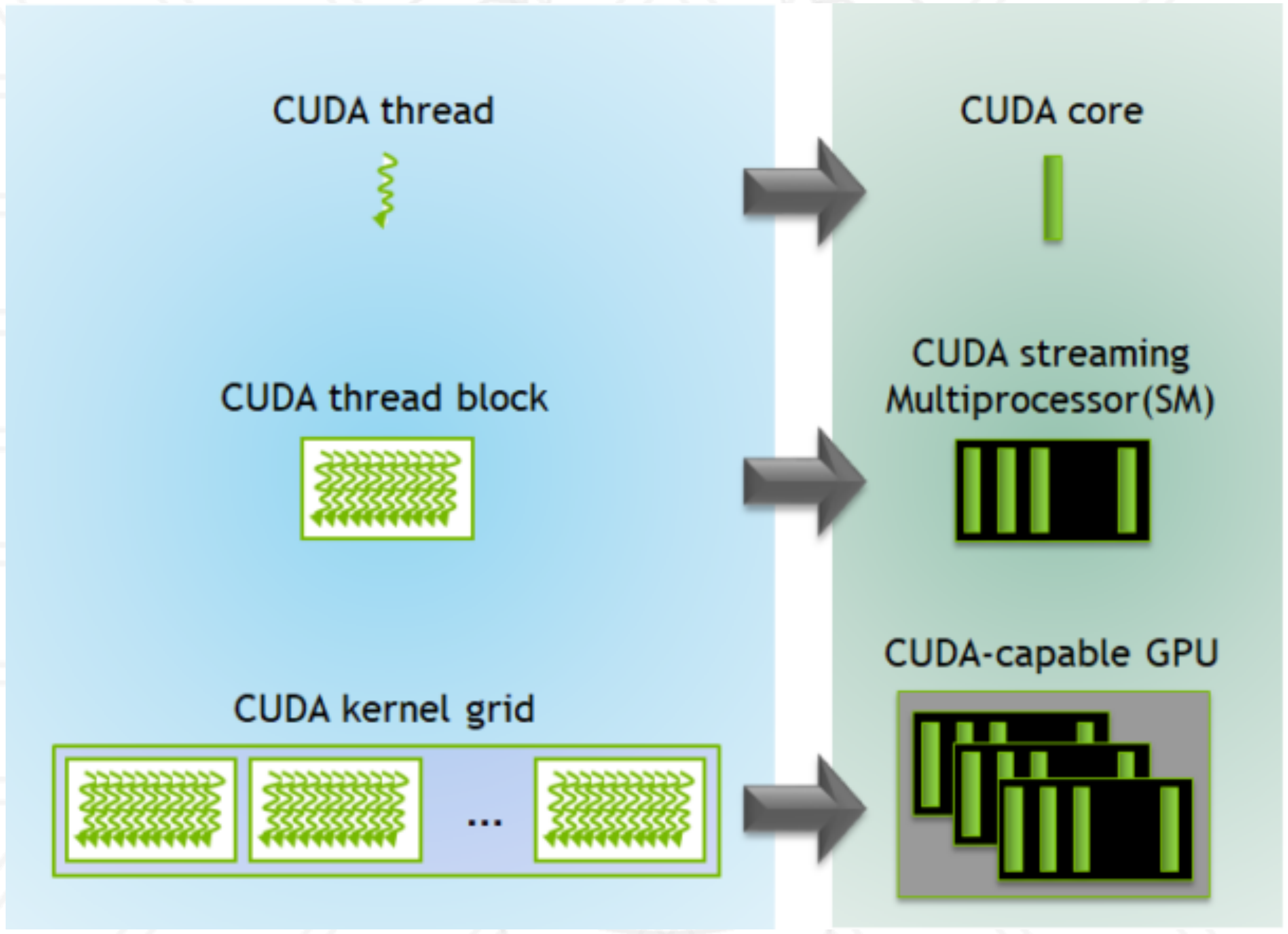
- Thread
  - Serial unit of execution
- Block
  - Collection of threads
  - Number of threads in block <= 1024
- Grid
  - Collection of blocks







Software to hardware mapping



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

Three steps to writing a CUDA kernel

- Copy input data from host to device memory
- Load the GPU program (kernel) and execute
- Copy the results back to host memory

Copying data to the GPU

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

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

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

// ... some computation on GPU …

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

cudaFree(d_Matrix);
```

Copying data to the GPU

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

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

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

// ... some computation on GPU …

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

cudaFree(d_Matrix);
```

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
cudaMemcpyHostToHost
cudaMemcpyDefault

CUDA syntax

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

int main() {
   ...
   saxpy<<<1, N>>>(x, y, alpha);
   ...
}
```

CUDA syntax

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

int main() {
   ...
   saxpy<<<1, N>>>(x, y, alpha);
   ...
}
```

CUDA syntax

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

int main() {
   ...
   saxpy<<<1, N>>>(x, y, alpha);
   ...
}
```

<<<#blocks, threads_per_block>>>

CUDA syntax

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

int main() {
   ...
   saxpy<<<1, N>>>(x, y, alpha);
   ...
}
```

What happens when:
N > 1024
or N > #device threads

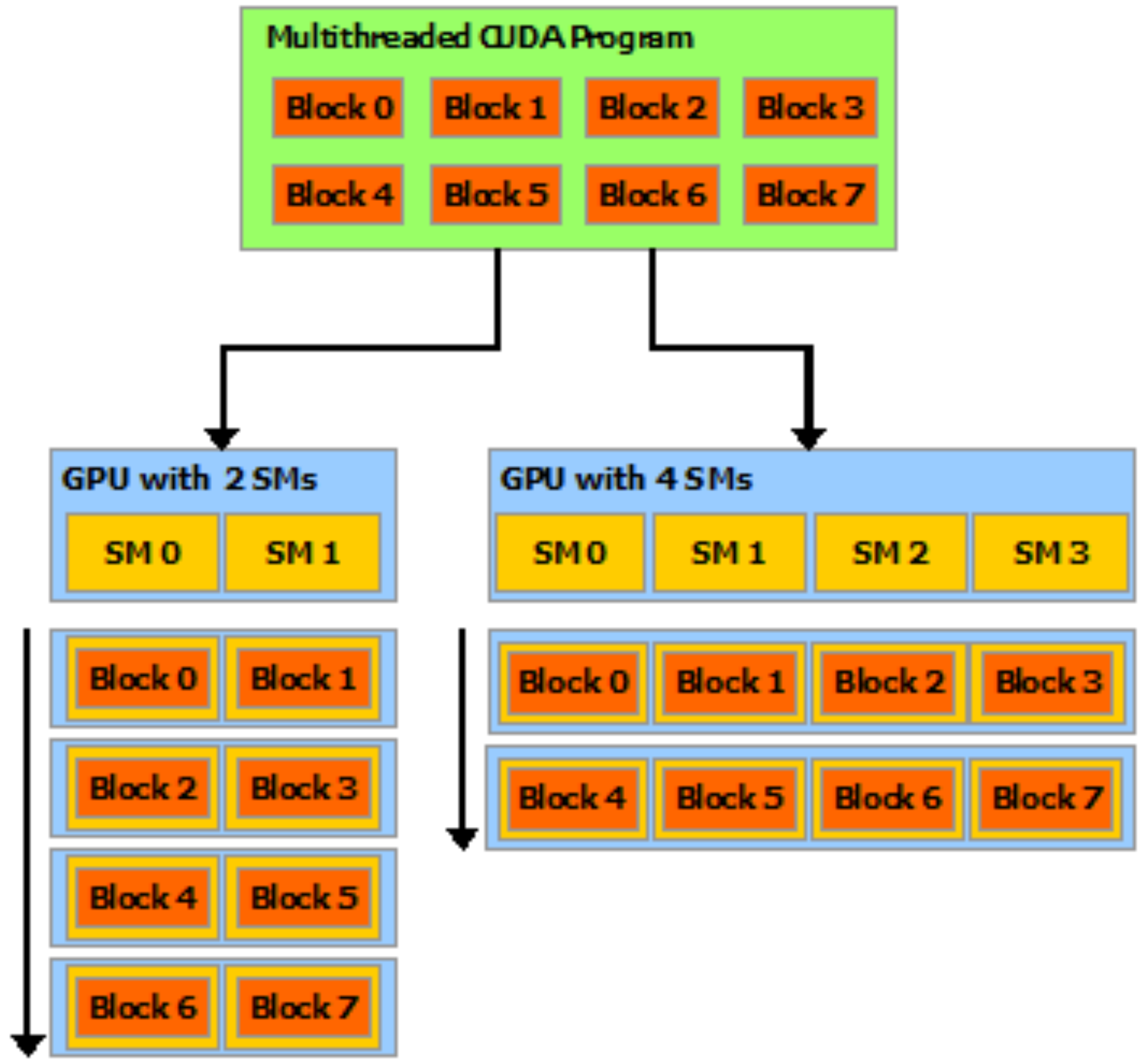
```
<<<#blocks, threads_per_block>>>
```

Multiple blocks

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



Questions?

UNIVERSITY OF MARYLAND

Abhinav Bhatele

5218 Brendan Iribe Center (IRB) / College Park, MD 20742

phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu