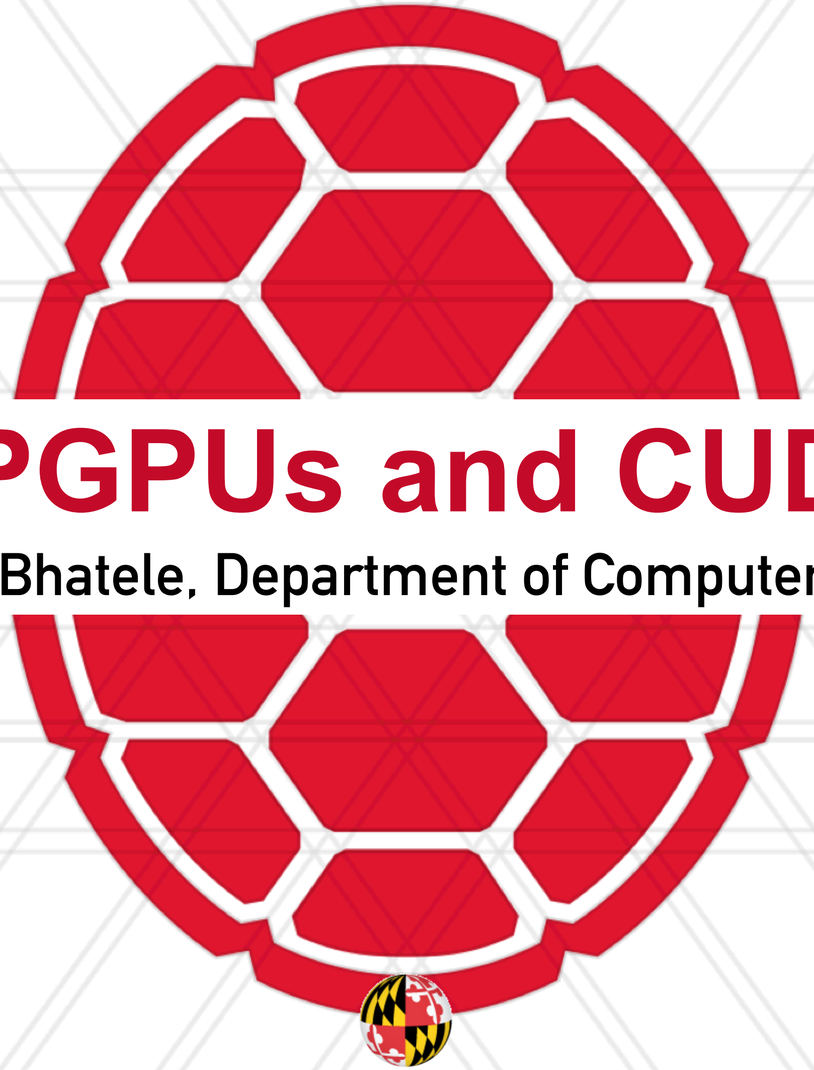
Parallel Computing (CMSC416 / CMSC616)



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: NVIDIA GeForce 256 (1999)
  - Around 1999-2001, early GPGPU results
- FPGAs

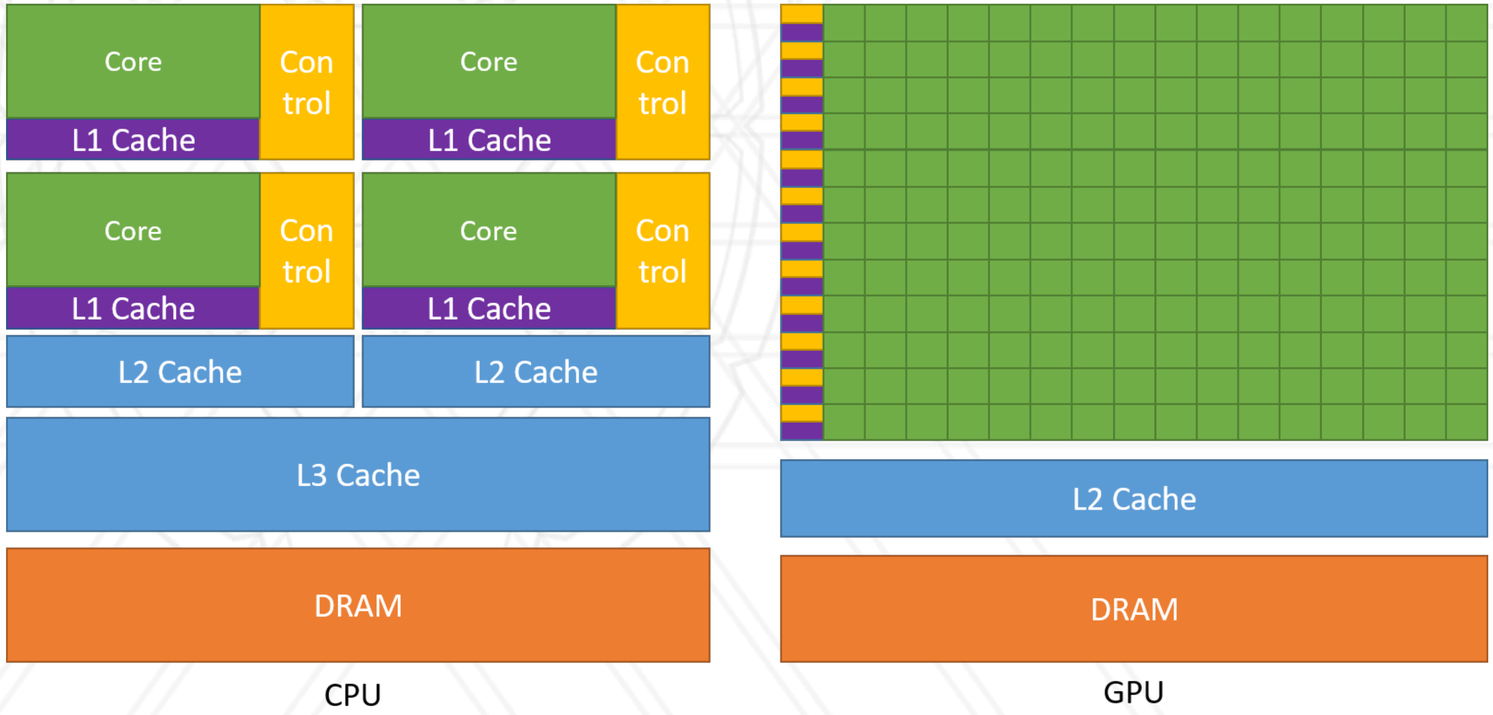
https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf

Used for mainstream HPC

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

GPGPU Hardware

- Higher instruction throughput
- Hide memory access latencies with computation



Comparing GPUs to CPUs

- Intel i9 11900K
  - 8 cores
  - 3.3 GHz
- AMD Epyc 7763
  - 64 cores
  - 2.45 GHz
- NVIDIA GeForce RTX 3090
  - 10,496 cores
  - 1.4 GHz
- NVIDIA A100
  - 17,712 cores
  - 0.76 GHz

Hopper H100 SM

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

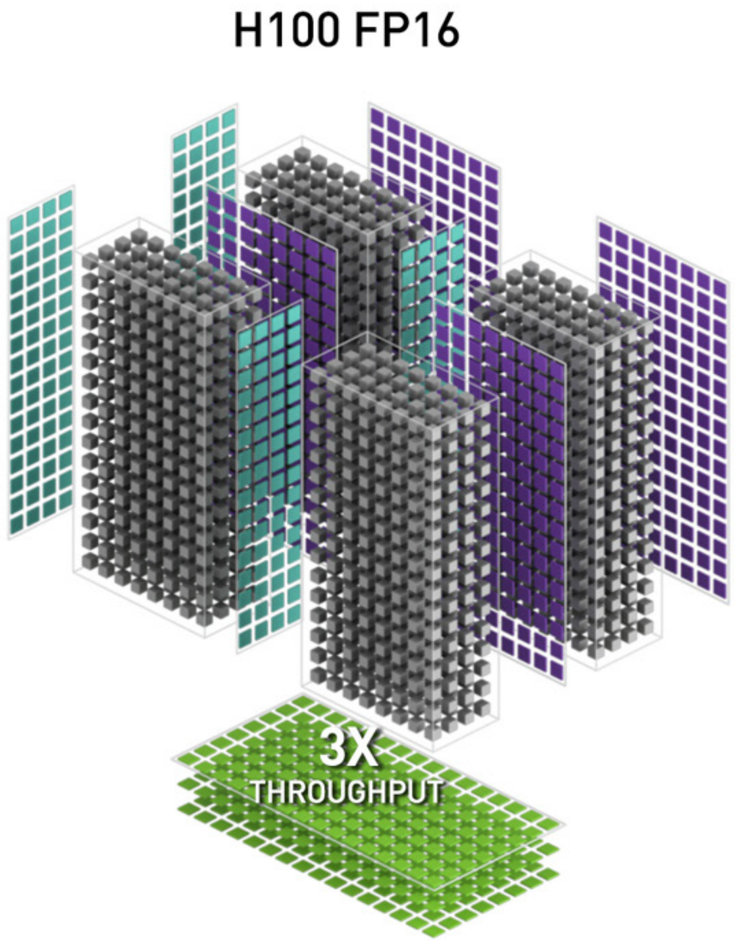
Hopper H100 SM

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

NVIDIA H100 chip

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



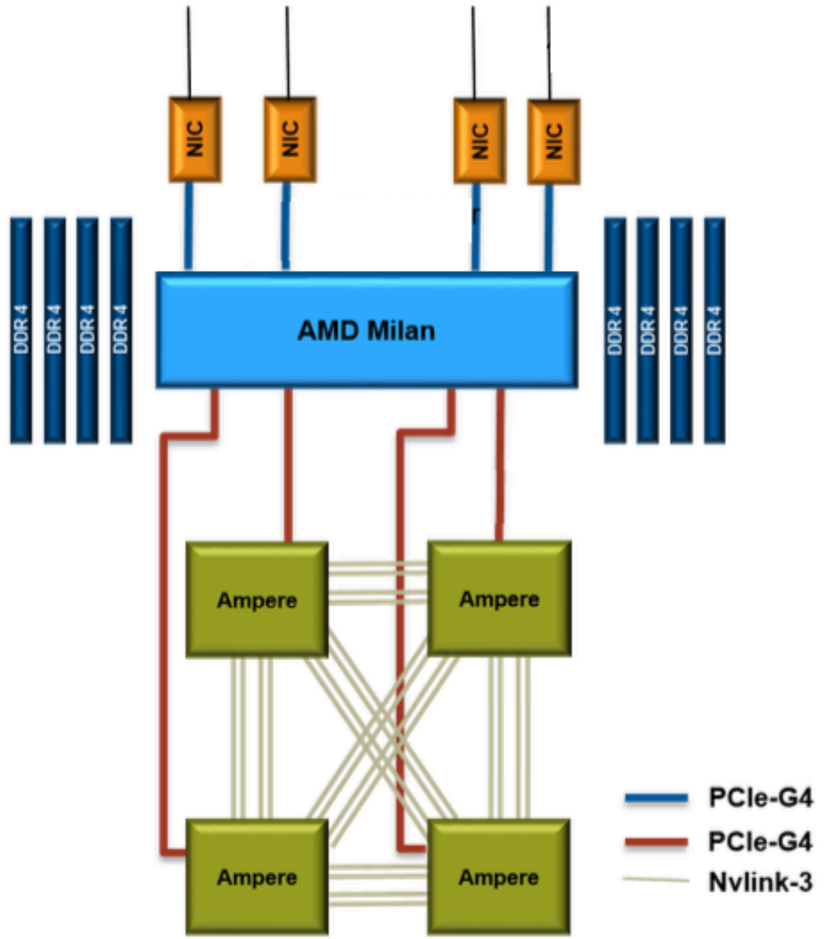
https://resources.nvidia.com/en-us-tensor-core

Announcements

- Assignment 3 is posted
  - Due on October 28 11:59 pm
- Midterm on October 30 during class

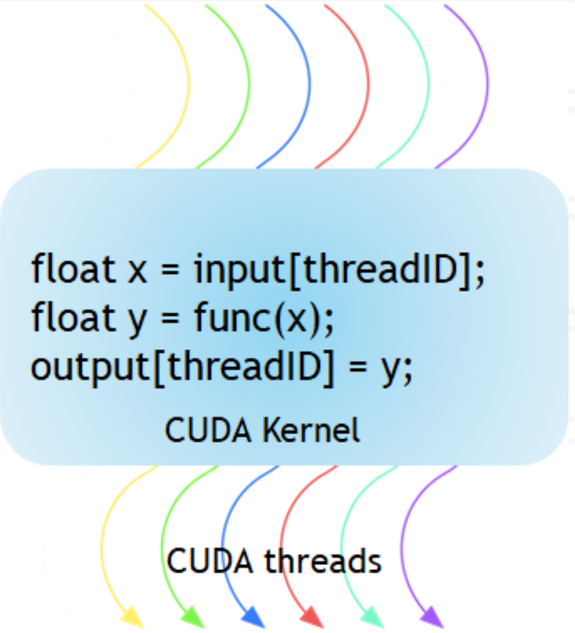
Nodes with GPUs

- NIC: Network interface card that connects the node to the network
- PCIe: high-speed interface often used to connect CPUs and GPUs
- NVLink: NVIDIA's high-speed interface often used between GPUs



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

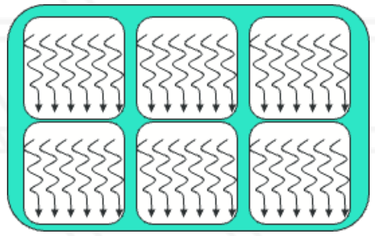
- Thread
  - Serial unit of execution

CUDA software abstraction

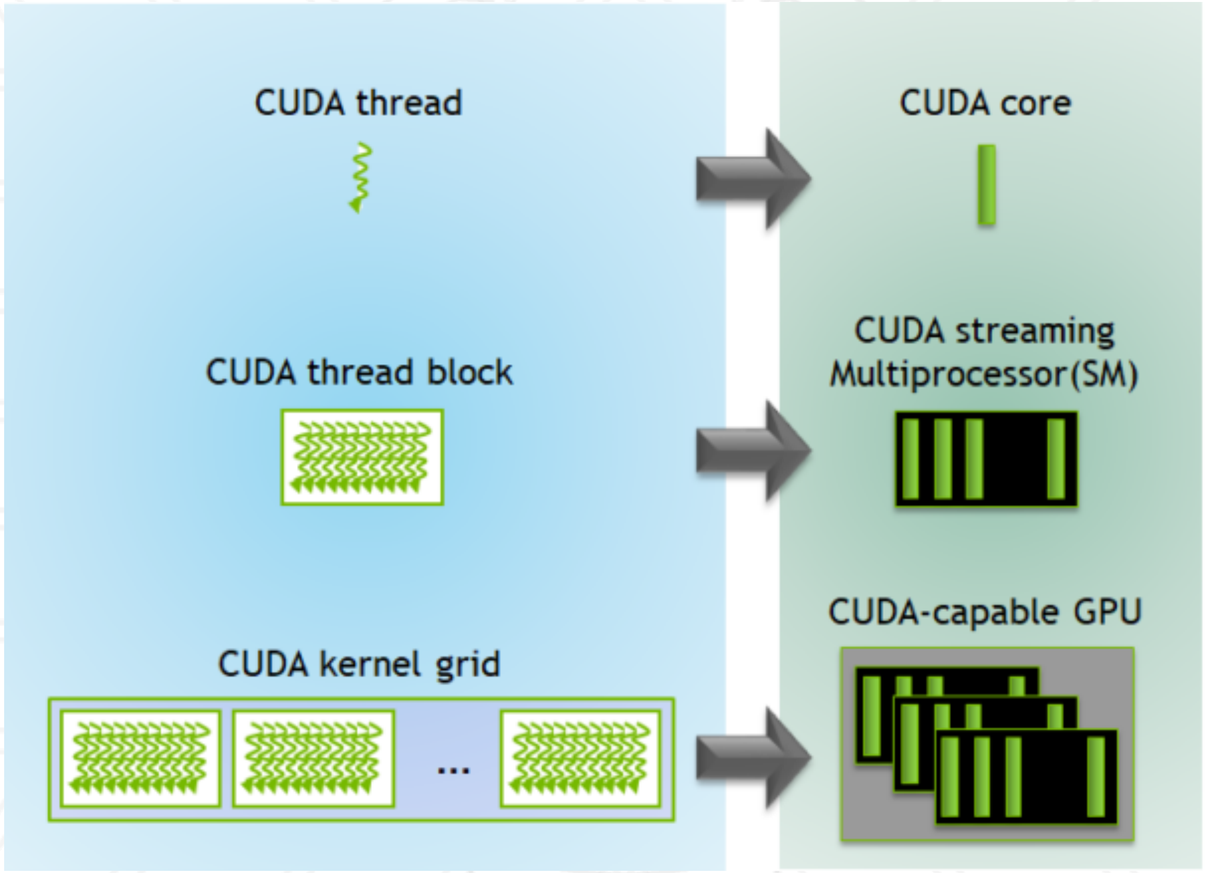
- Thread
  - Serial unit of execution
- Block
  - Collection of threads
  - Number of threads in block <= 1024

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 *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);

// ... some computation on GPU …

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

cudaFree(d_Matrix);
```

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);

// ... some computation on GPU …

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

cudaFree(d_Matrix);
```

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
cudaMemcpyHostToHost
cudaMemcpyDefault

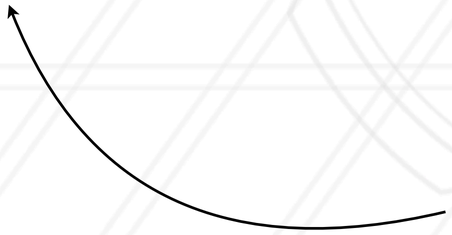
CUDA syntax

```
__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);
    ...
}
```

Grid size, Block size

CUDA syntax

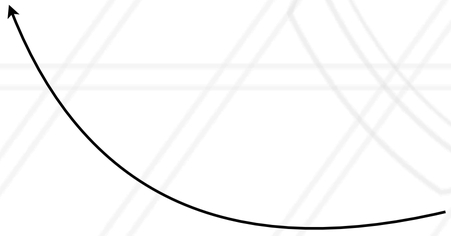
```
__global__ void saxpy(float *x, float *y, float alpha) {
    int i = threadIdx.x;
    y[i] = alpha*x[i] + y[i];
}

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



Grid size, Block size

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);
    ...
}
```



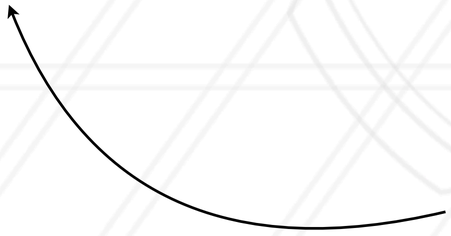
<<<#blocks, threads_per_block>>>

Grid size, Block size

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:
array size (N) > 1024?



<<<#blocks, threads_per_block>>>

Grid size, Block size

Compiling CUDA code

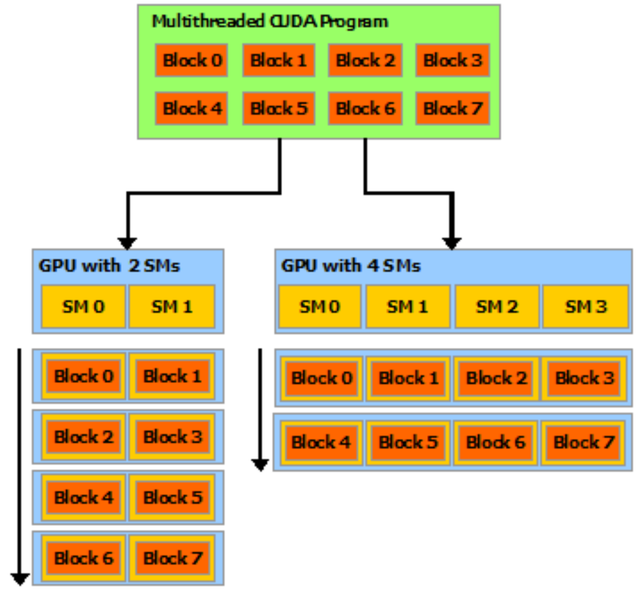
```
nvcc -o saxpy --generate-code arch=compute_80,code=sm_80 saxpy.cu

./saxpy
```

Multiple blocks

```
__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);
    ...
}
```

Multiple blocks

```
__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);
    ...
}
```



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