GPU Programming with CUDA

Parallelization

Prof. Dr. Matthias S. Müller
Dr. Christian Terboven
Dr. Sandra Wienke
Julian Miller

What is This Chapter About?

- How to parallelize an application with CUDA
  - Offloading regions
  - Data management

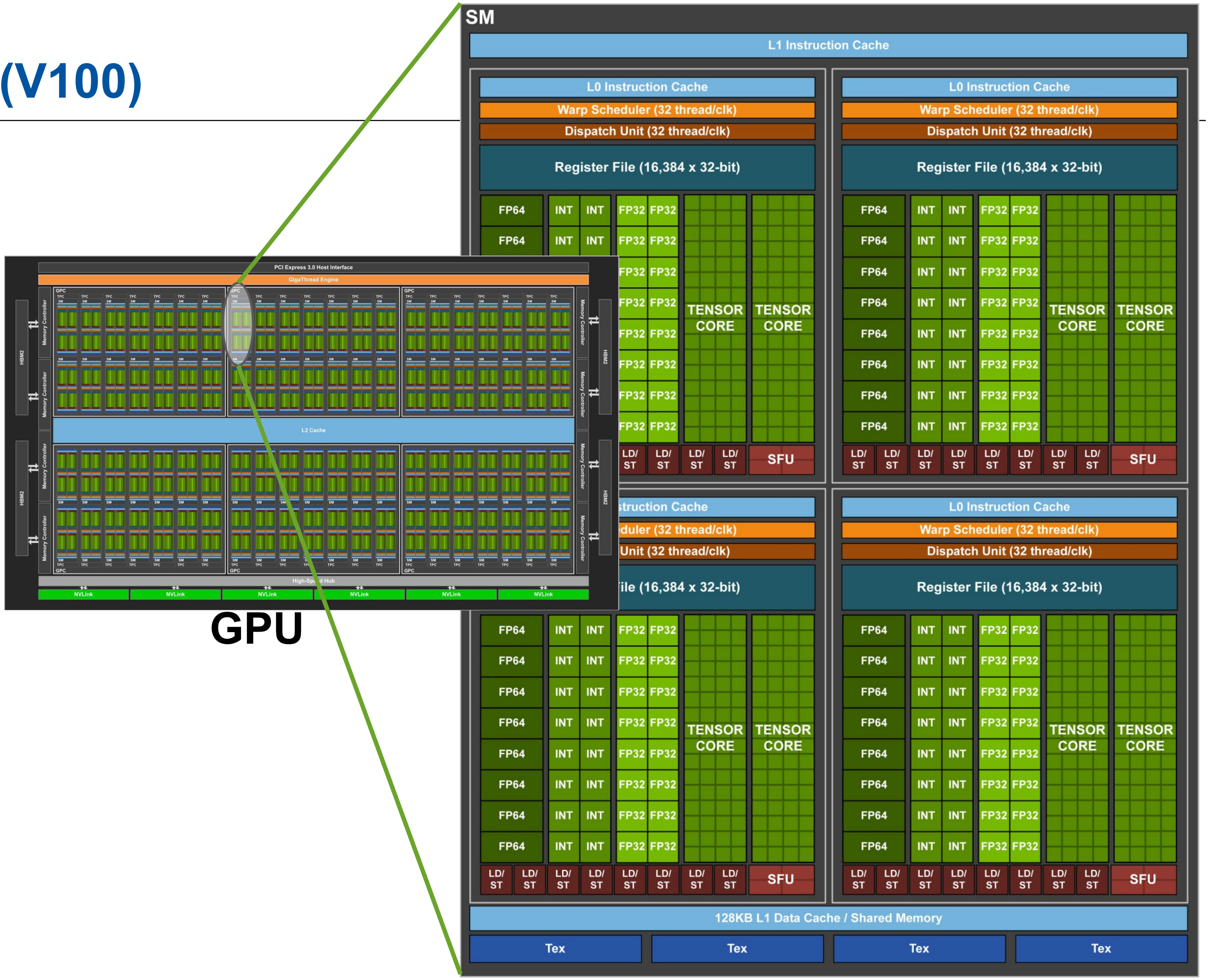
Example DAXPY: CPU

```
void daxpy(int n, double a, double *x, double *y) {
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

Output:
$ $CC daxpy.c daxpy.exe
$ daxpy.exe
Max error: 0.00000
Time elapsed: 0.115152 s

Example GPU Architecture: Volta (V100)

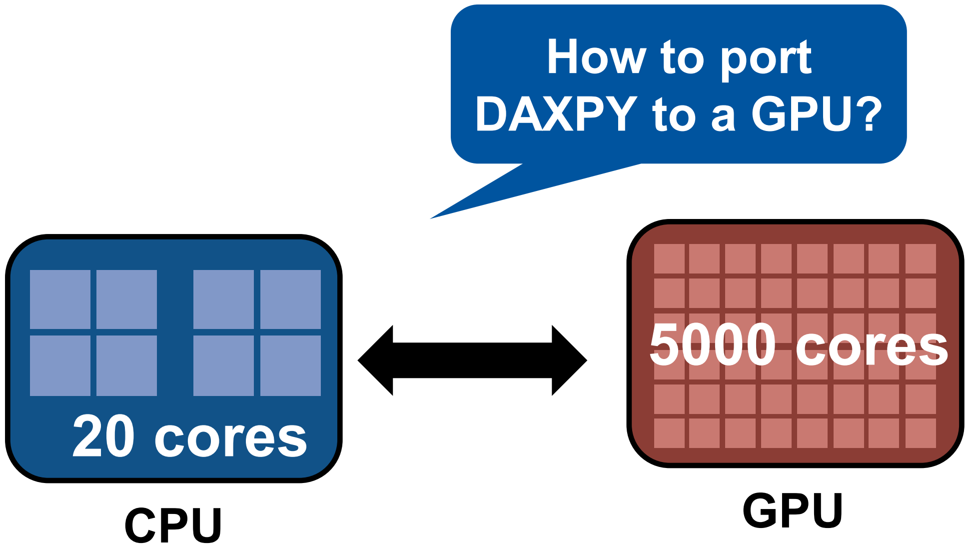
- 21.1 billion transistors
- 80 streaming multiprocessors (SM)
  - Each: 64 (SP) cores, 32 (DP) cores, 8 Tensor cores
- Peak performance
  - SP: 15.7 Tflops
  - DP: 7.8 Tflops
  - Tensor: 125 Tflops
- 32 GB / 16 GB HBM2 memory
  - 900 GB/s bandwidth
- 300W thermal design power



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

Example DAXPY: How to Port to GPU?

```
void daxpy(int n, double a, double *x, double *y) {
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```



Kernel Directives

- Kernel code
  - Function qualifiers: `__global__`, `__device__`, `__host__`
  - Built-in variables: `gridDim`: contains dimensions of grid (type `dim3`)
    `blockDim` : contains dimensions of block (type `dim3`)
    `blockIdx` : contains block index within grid (type `uint3`)
    `threadIdx`: contains thread index within block (type `uint3`)
  - Compute unique IDs, e.g. global 1D Idx:
    `gIdx = blockIdx.x * blockDim.x + threadIdx.x`

- Kernel usage
  - Kernel arguments can be passed directly to the kernel
  - Kernel invocation with *execution configuration* (chevron syntax):
    `func<<<dimGrid, dimBlock>>> (parameter)`

Example: DAXPY

```
__global__ void daxpyGPU(int n, double a, double *x, double *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  const int n = 100000000; const double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }

  daxpyGPU<<<1, 1>>>(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y); return 0;
}
```

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 2.00000
Time elapsed: 0.06s

Example DAXPY: Debugging

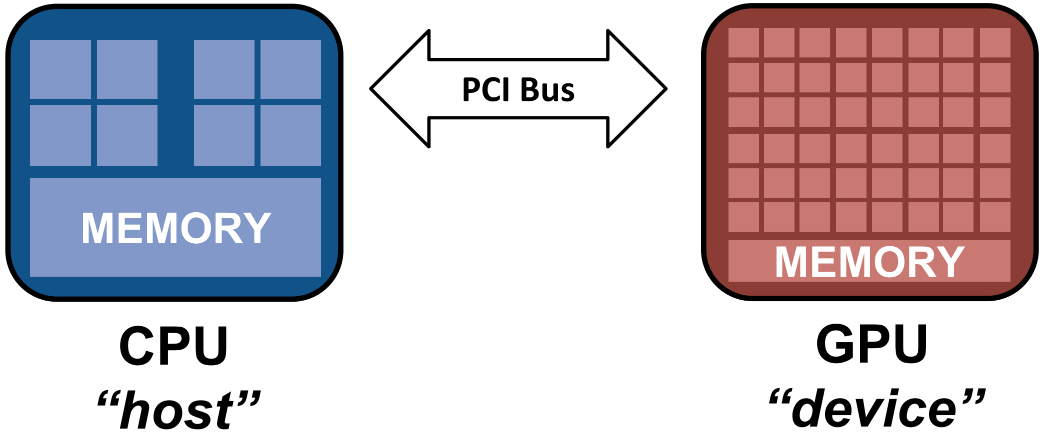
- No compiler error but cryptic runtime error
- NVIDIA Profiler

```
$ nvprof daxpy.exe
==40419== NVPROF is profiling process 40419, command: daxpy.exe
==40419== Profiling application: daxpy.exe
==40419== Profiling result:
No kernels were profiled.

==40419== API calls:
No API activities were profiled.
```

- Cuda-memcheck

```
$ cuda-memcheck daxpy.exe
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 8
=========     at 0x000000b8 in daxpyGPU(int, double, double*, double*)
=========     by thread (0,0,0) in block (0,0,0)
```

Recap: GPU Memory Spaces

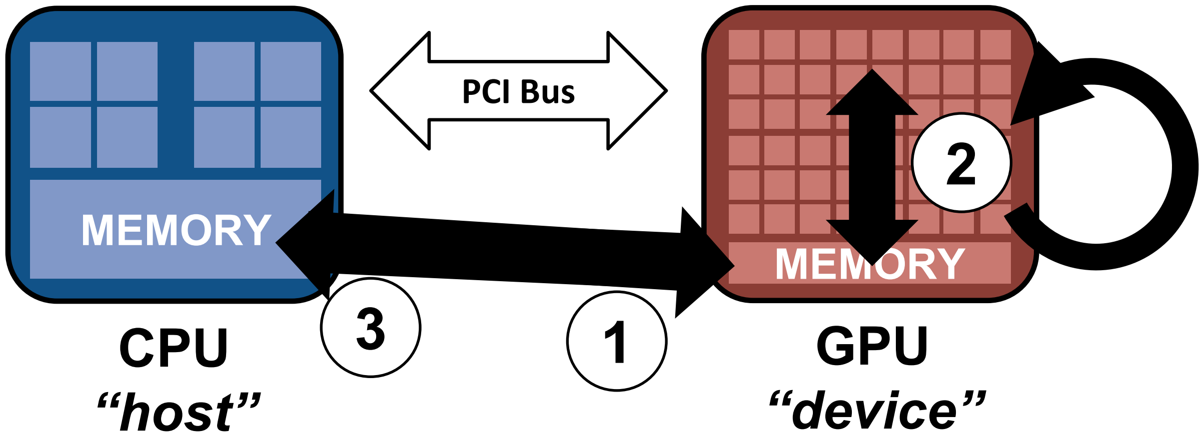


We refer to "discrete GPUs" here.

- Weak memory model
  - Host + device memory = separate entities
  - No coherence between host + device
    - Data transfers needed

- Pointers are addresses
  - Dereferencing pointers to memory in the other space likely fails with a segmentation violation

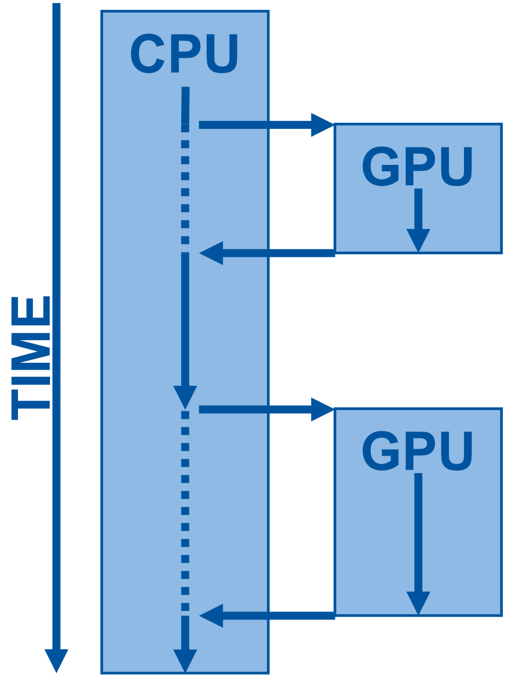
Recap: Offloading



We refer to "discrete GPUs" here.

- Weak memory model
  - Host + device memory = separate entities
  - No coherence between host + device
    - Data transfers needed
- Host-directed execution model
  - Copy input data from CPU mem. to device mem.
  - Execute the device program
  - Copy results from device mem. to CPU mem.

processing flow *(simplified)*



CUDA – Data Management

- Variable type qualifiers

  `__device__`, `__shared__`, `__constant__`

- Memory management

  `cudaMalloc(pointerToGPUMem, size)`

  `cudaFree(pointerToGPUMem)`

- Memory transfer (synchronous)

  `cudaMemcpy(dest, src, size, direction)`

`direction:`
`cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost`
`cudaMemcpyDeviceToDevice`
`cudaMemcpyDefault` *(with UVA)*

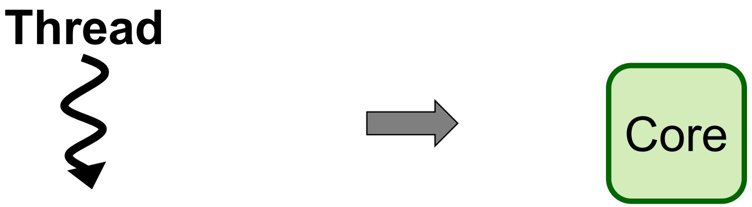
Example DAXPY: Data Management

```
int main(int argc, const char* argv[]) {
  const int n = 100000000; const double a = 2.0;
  double *h_x = (double *) malloc(n * sizeof(double));
  double *h_y = (double *) malloc(n * sizeof(double));
  // Initialize x, y on host

  // Device pointer and memory allocation on device
  double *d_x, *d_y;
  cudaMalloc(&d_x, n * sizeof(double));
  cudaMalloc(&d_y, n * sizeof(double));
  // copy memory to device
  cudaMemcpy(d_x, h_x, n * sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, h_y, n * sizeof(double), cudaMemcpyHostToDevice);

  daxpyGPU<<<1, 1>>>(n, a, d_x, d_y);  // Invoke daxpy kernel

  // copy memory to host
  cudaMemcpy(h_y, d_y, n * sizeof(double), cudaMemcpyDeviceToHost);
  // Check if all values are 4.0

  cudaFree(d_x); cudaFree(d_y); free(h_x); free(h_y); return 0;
}
```

For comparison: ~0.12s on a single CPU core

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 0.00000
Total runtime: 8.50s
Kernel runtime: 8.01s

Mapping to Hardware



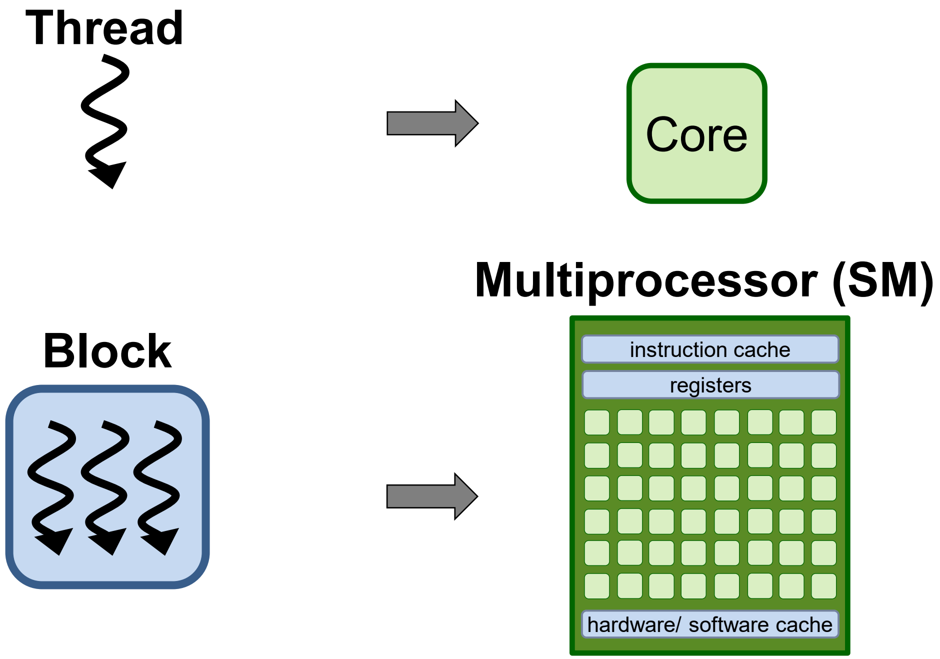
- Each thread is executed by a core

Example DAXPY: Thread Parallelism

```
__global__ void daxpyGPU(int n, double a, double *x, double *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = a * x[i] + y[i];
}


int main(int argc, const char* argv[]) {
  ...
  // Invoke daxpy kernel
  daxpyGPU<<<1, 256>>>(n, a, d_x, d_y);
  ...
}
```

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 0.00000
Total runtime: 0.65s
Kernel runtime: 0.16s

Mapping to Hardware



- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources

Example DAXPY: Thread Parallelism

```
__global__ void daxpyGPU(int n, double a, double *x, double *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = a * x[i] + y[i];
}


int main(int argc, const char* argv[]) {
  ...
  // Invoke daxpy kernel
  daxpyGPU<<<(n+255)/256, 256>>>(n, a, d_x, d_y);
  ...
}
```

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 0.00000
Total runtime: 0.49s
Kernel runtime: 2.91ms

Example DAXPY: Thread Parallelism

- Grid-strided loop

```
__global__ void daxpyGPU(int n, double a, double *x, double *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = a * x[i] + y[i];
}


int main(int argc, const char* argv[]) {
  ...
  // Invoke daxpy kernel
  daxpyGPU<<<(n+255)/256, 256>>>(n, a, d_x, d_y);
  ...
}
```

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 0.00000
Total runtime: 0.49s
Kernel runtime: 2.91ms

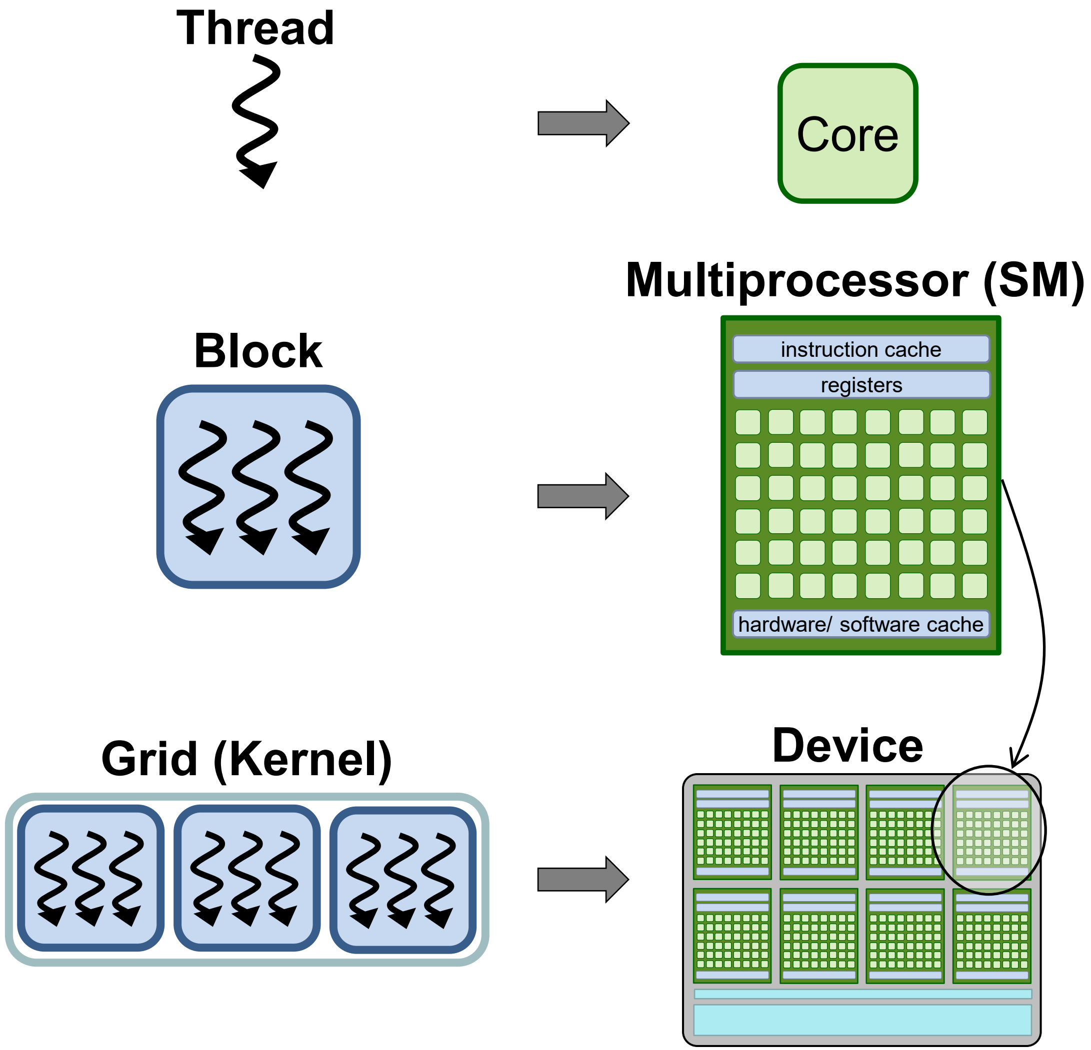
Example DAXPY: Thread Parallelism

- Alternative implementation

```
__global__ void daxpyGPU(int n, double a, double *x, double *y) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n)
        y[tid] = a * x[tid] + y[tid];
}


int main(int argc, const char* argv[]) {
  ...
  // Invoke daxpy kernel
  daxpyGPU<<<(n+255)/256, 256>>>(n, a, d_x, d_y);
  ...
}
```

Output:
$ nvcc daxpy.cu
$ a.out
Max error: 0.00000
Total runtime: 0.49s
Kernel runtime: 2.91ms

Mapping to Hardware



- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources
- Each kernel is executed on a device