CUDA

Overview

1. GPU Architecture
2. Data Parallelism
3. CUDA Program Structure
4. Vector Addition
5. Device Global Memory and Data Transfer
6. Kernel Functions and Threading

GPU Architecture

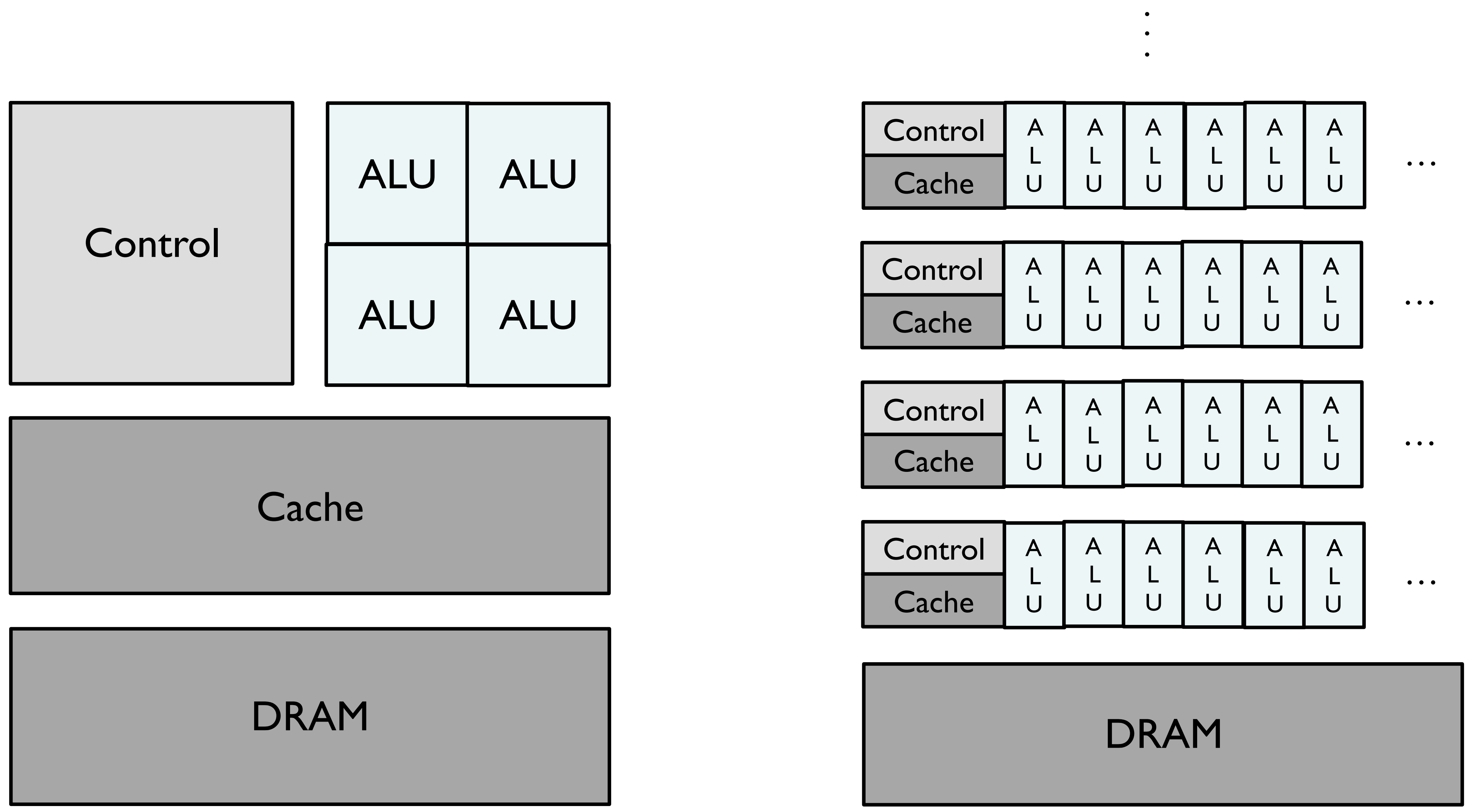
GPU Architecture

Some general terminology:

A *device* refers to a GPU in the computer.

A *host* refers to a CPU in the computer.

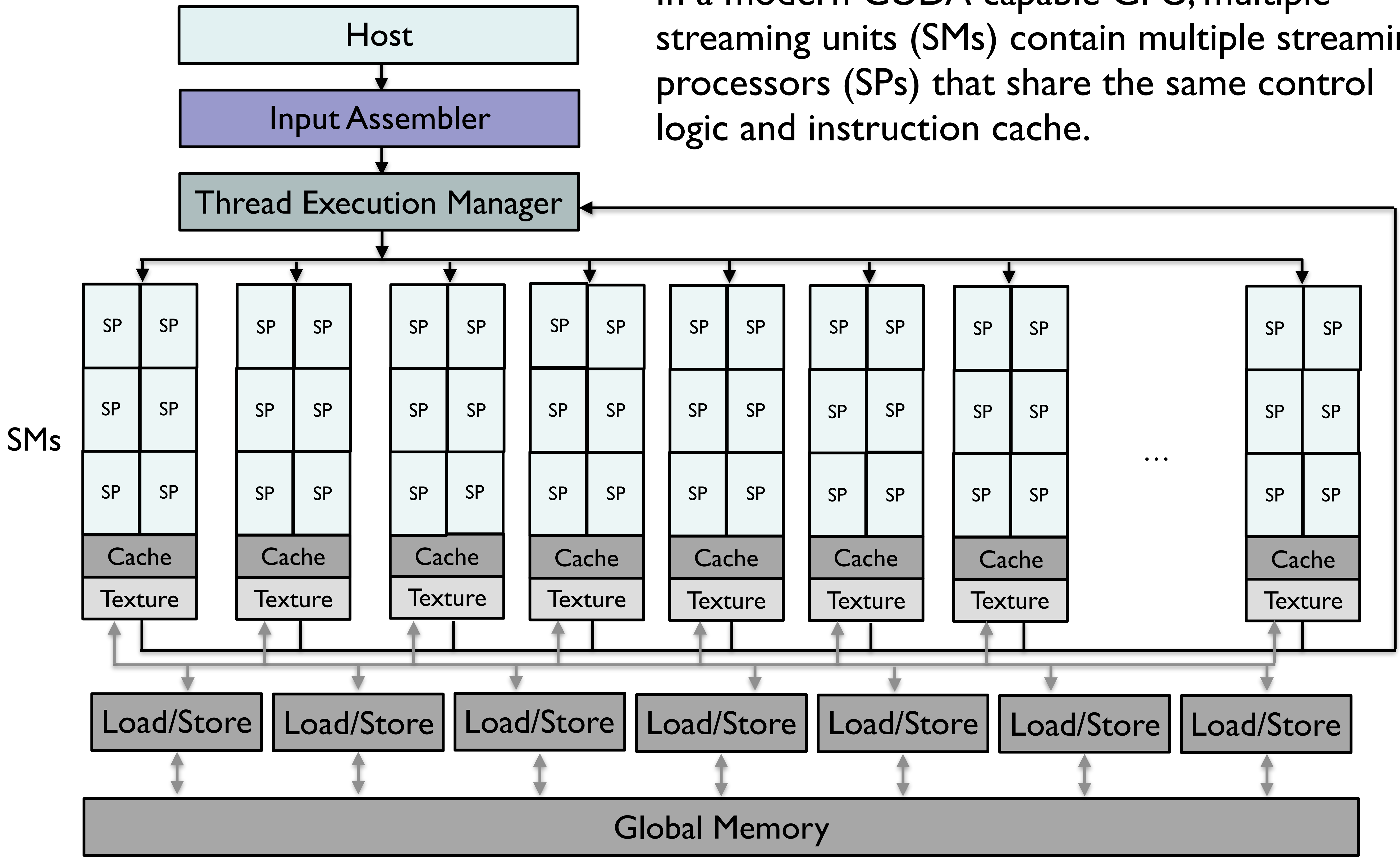
GPUs vs CPUs



CPUs are designed to minimize the execution time of single threads. GPUs are designed to maximize the throughput of many identical threads working on different memory.

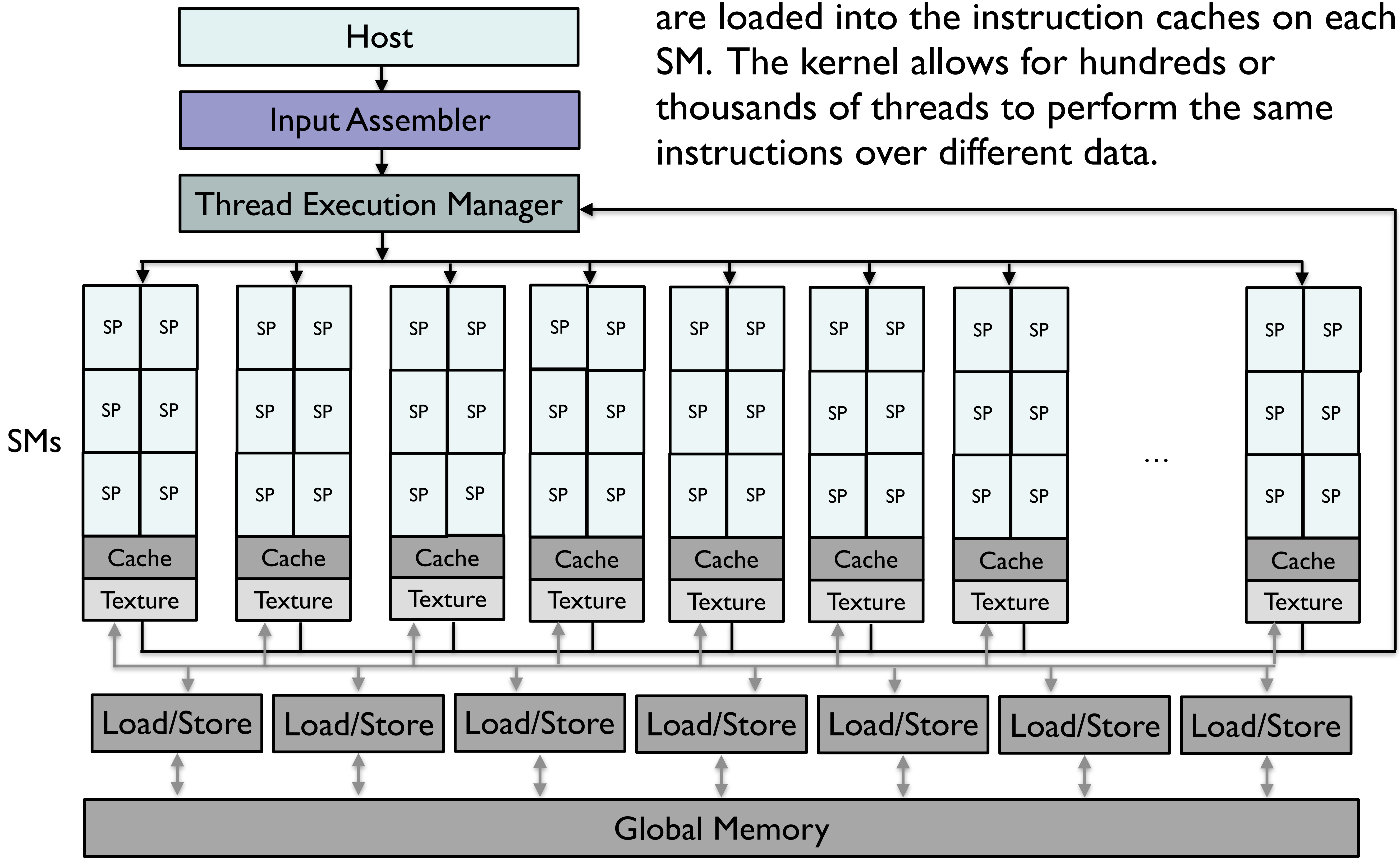
CUDA Capable GPU Architecture

In a modern CUDA capable GPU, multiple streaming units (SMs) contain multiple streaming processors (SPs) that share the same control logic and instruction cache.



CUDA Capable GPU Architecture

CUDA programs define *kernels* which are what are loaded into the instruction caches on each SM. The kernel allows for hundreds or thousands of threads to perform the same instructions over different data.



Data Parallelism

Task Parallelism vs Data Parallelism

Vector processing units and GPUs take care of data parallelism, where the same operation or set of operations need to be performed over a large set of data.

Most parallel programs utilize data parallelism to achieve performance improvements and scalability.

Task Parallelism vs Data Parallelism

Task parallelism, however, is when a large set of independent (or mostly independent) tasks need to be performed. The tasks are generally much more complex than the operations performed by data parallel programs.

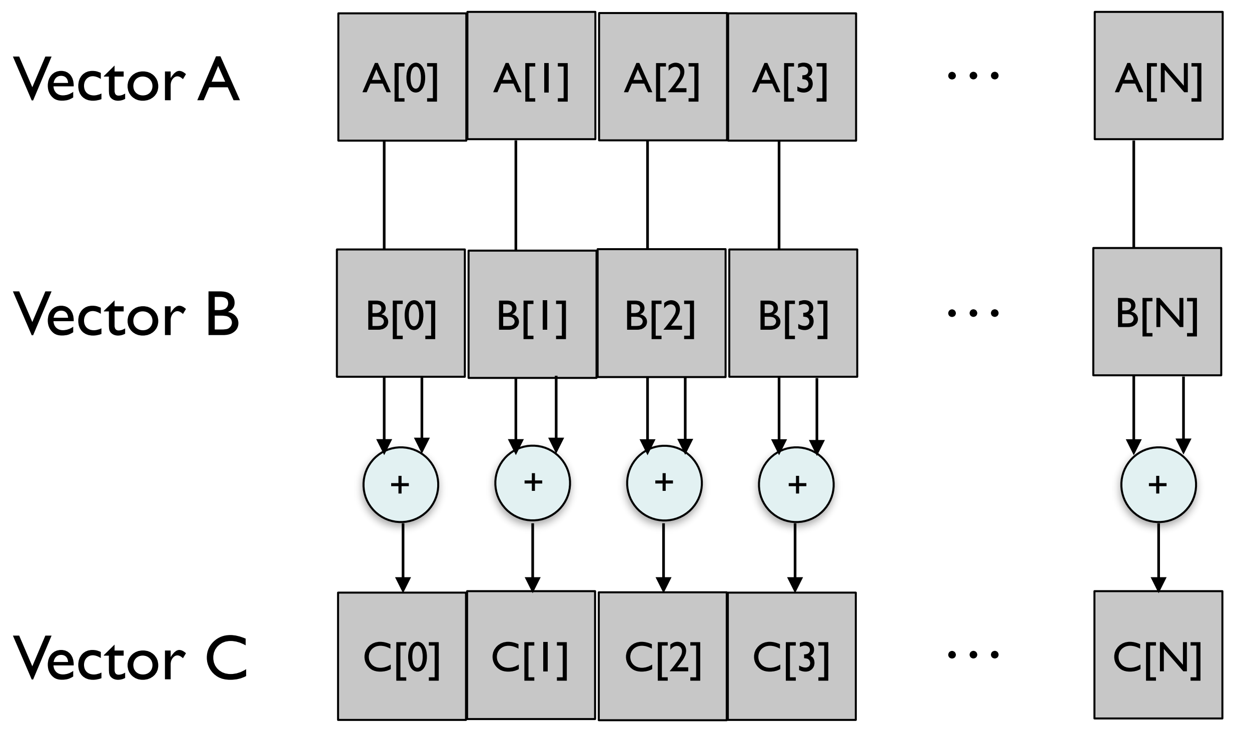
Task Parallelism vs Data Parallelism

It is worth mentioning that both can be combined — the tasks can contain data parallel elements.

Further, in many ways systems are getting closer to do both. GPU hardware and programming languages like CUDA are allowing more operations to be performed in parallel for the data parallel tasks.

Data Processing Example

For example, vector addition is a data parallel:

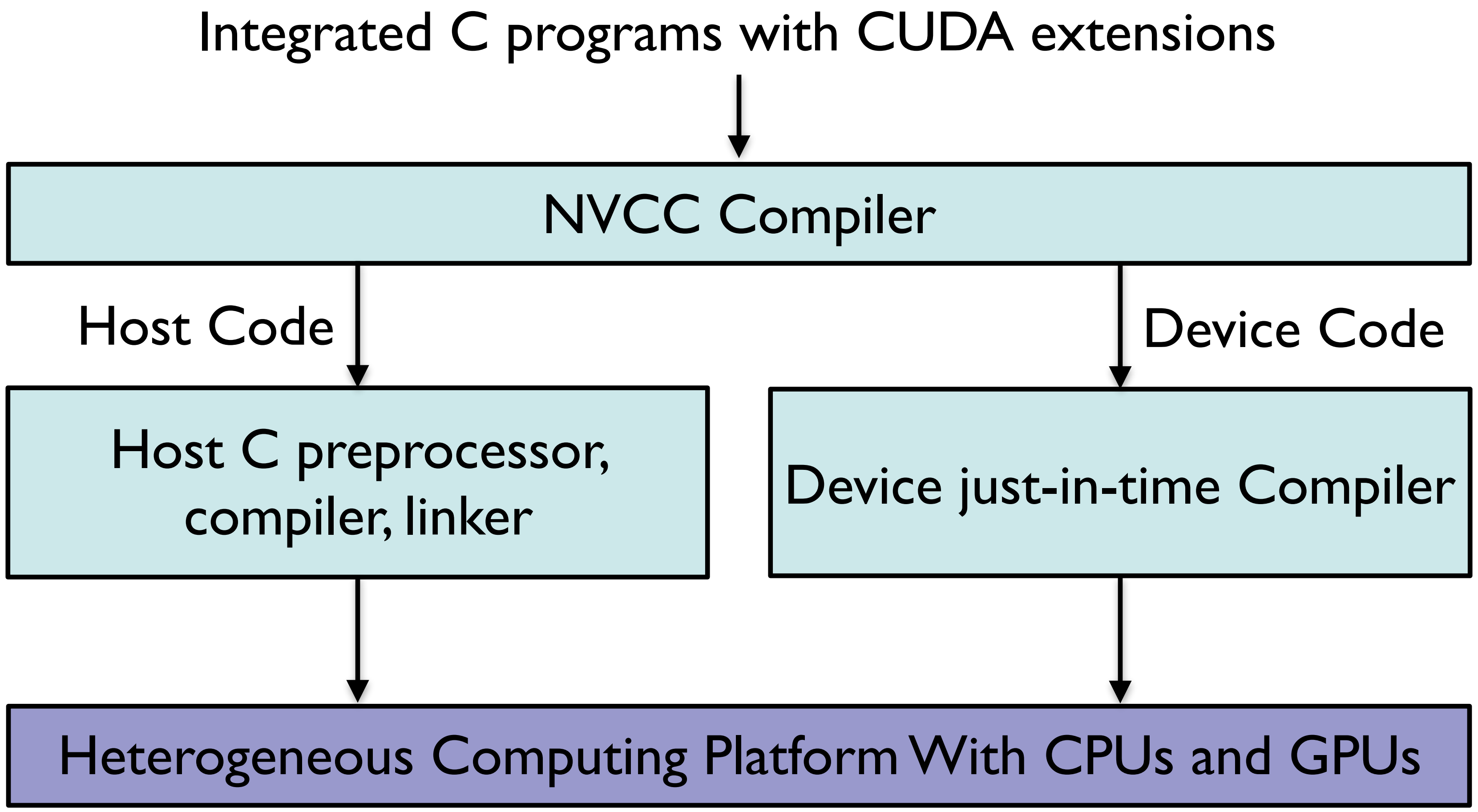


CUDA Program Structure

CUDA Program Structure

CUDA programs are C/C++ programs containing code that uses CUDA extensions.

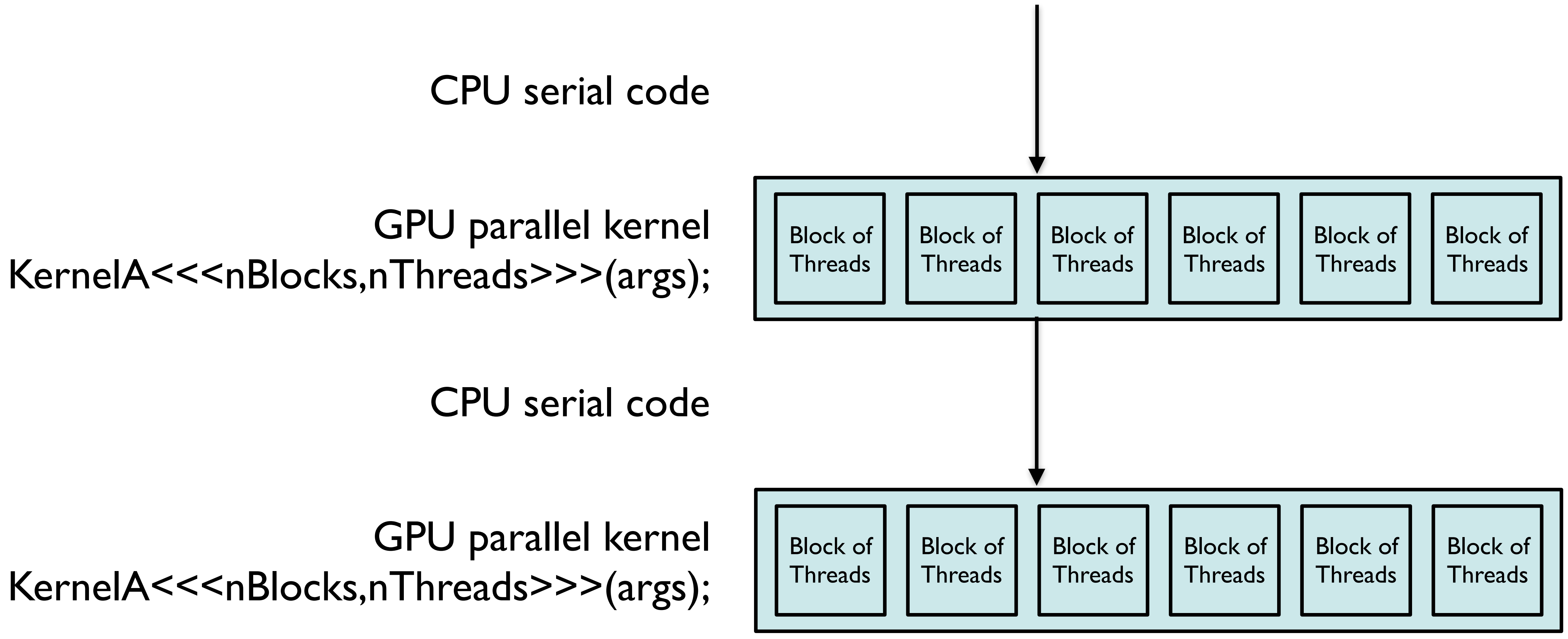
The CUDA compiler (nvcc, nvcxx) is essentially a wrapper around another C/C++ compiler (gcc, g++, llvm). The CUDA compiler compiles the parts for the GPU and the regular compiler compiles for the CPU:


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code
Host C preprocessor, compiler, linker
Device just-in-time Compiler
Heterogeneous Computing Platform With CPUs and GPUs

CUDA Program Structure

When CUDA programs run, they alternate between CPU serial code and GPU parallel kernels (which execute many threads in parallel).

The program will block waiting for the CUDA kernel to complete executing all its parallel threads.



Vector Addition

Vector Addition in C

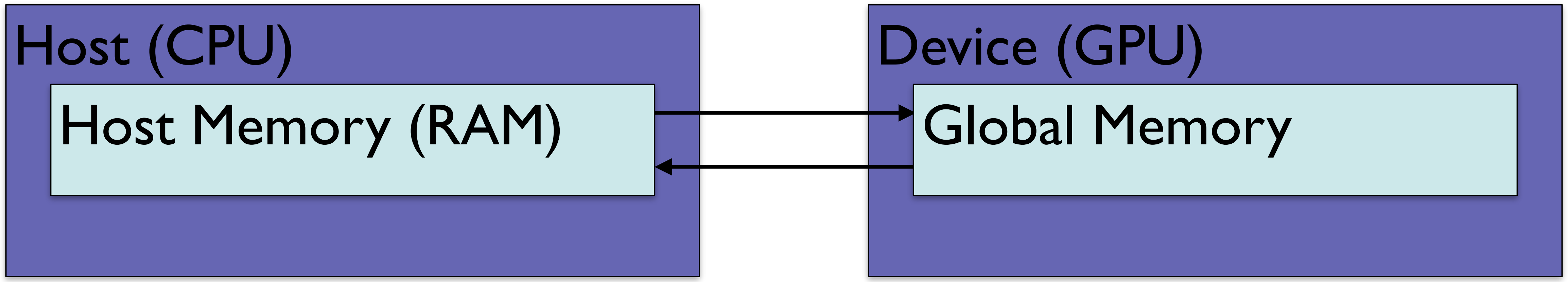
```
//Compute vector sum h_C = h_A + h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
  for (int i = 0; i < n; i++) {
    h_C[i] = h_A[i] + h_B[i];
  }
}

int main() {
  float* h_A = new float[100];
  float* h_B = new float[100];
  float* h_C = new float[100];
  …
  //assign elements into h_A, h_b
  …
  vecAdd(h_A, h_B, h_C, 100);
}
```

The above is a simple C program which performs vector addition. The arrays h_A and h_b are added with the results being stored into h_C.

Skeleton Vector Add in CUDA

It’s important to note that GPUs and CPUs do *not* share the same memory, so in a CUDA program you have to move the memory back and forth.



Skeleton Vector Add in CUDA

```
//We want to parallelize this!
//Compute vector sum h_C = h_A + h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n) {
  for (int i = 0; i < n; i++) {
    h_C[i] = h_A[i] + h_B[i];
  }
}

int main() {
  float* h_A = new float[100];
  float* h_B = new float[100];
  float* h_C = new float[100];
  …
  //assign elements into h_A, h_b

  //allocate enough memory for h_A, h_B, h_C on the GPU
  //copy the memory of h_A and h_B onto the GPU
  vecAddCUDA(h_A, h_B, h_C); //perform this on the GPU!
  //copy the memory of h_C on the GPU back into h_C on the CPU
}
```

Device Global Memory and Data Transfer

Memory Operations in CUDA

CUDA provides three methods for allocating on and moving memory to GPUs.

```
cudaMalloc();   //allocates memory on the GPU (like malloc)
cudaFree();     //frees memory on the GPU (like free)

cudaMemcpy();   //copies memory from the host to the
                //device (like memcpy)
```

All three methods can return a `cudaError_t` type, which can be used to test for error conditions, eg:

```
cudaError_t err = cudaMalloc((void**) &d_A, size);
```

Memory Operations in CUDA

All three methods can return a `cudaError_t` type, which can be used to test for error conditions, eg:

```
cudaError_t err = cudaMalloc((void**) &d_A, size);

if (err != cudaSuccess) {
  printf("%s in file %s at line %d\n",
         cudaGetErrorString(err),
         __FILE__, __LINE__);
  exit(EXIT_FAILURE);
}
```

Skeleton Vector Add in CUDA

```
int main() {
  int size = 100;
  float* h_A = new float[size];
  float* h_B = new float[size];
  float* h_C = new float[size];
  //assign elements into h_A, h_b
  …

  //allocate enough memory for h_A, h_B, h_C as d_A, d_B, d_C
  //on the GPU
  float *d_A, *d_B, *d_C;
  cudaMalloc((void**) &d_A, sizeof(float) * size);
  cudaMalloc((void**) &d_B, sizeof(float) * size);
  cudaMalloc((void**) &d_C, sizeof(float) * size);
  //copy the memory of h_A and h_B onto the GPU
  cudaMemcpy(d_A, h_A, sizeof(float) * size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, sizeof(float) * size, cudaMemcpyHostToDevice);

  vecAddCUDA(h_A, h_B, h_C); //perform this on the GPU!

  //copy the memory of h_C on the GPU back into h_C on the CPU
  cudaMemcpy(d_C, h_C, sizeof(float) * size, cudaMemcpyDeviceToHost);

  //free the memory on the device
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

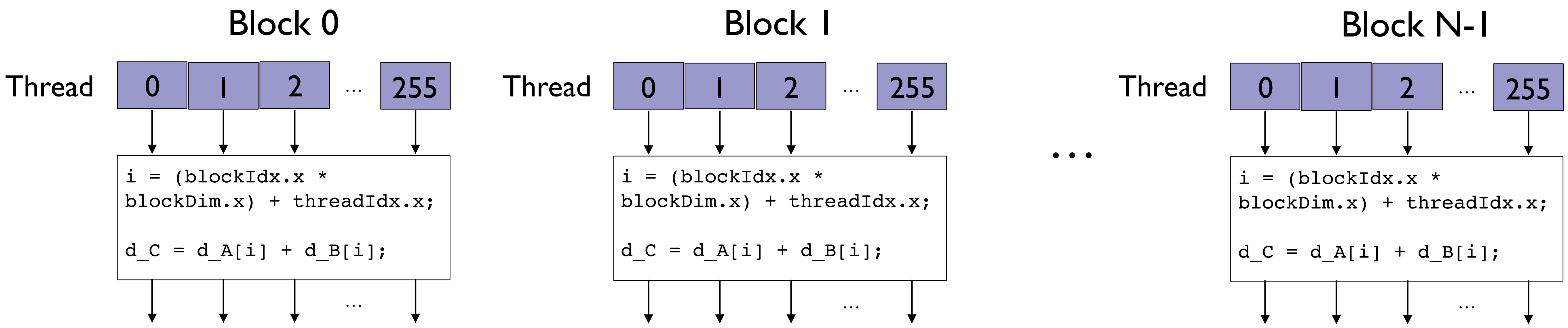
Kernel Functions and Threading

CUDA Grids

CUDA assigns groups of threads to *blocks*, and groups of blocks to *grids*. For now, we'll just worry about blocks of threads.

Using the *blockIdx*, *blockDim*, and *threadIdx* keywords, you can determine which thread is being run in the kernel, from which block.

For our GPU vector add, this will work something like this:



```
i = (blockIdx.x *
blockDim.x) + threadIdx.x;

d_C = d_A[i] + d_B[i];
```

```
i = (blockIdx.x *
blockDim.x) + threadIdx.x;

d_C = d_A[i] + d_B[i];
```

```
i = (blockIdx.x *
blockDim.x) + threadIdx.x;

d_C = d_A[i] + d_B[i];
```

Vector Add Kernel

```
//Computes the vector sum on the GPU.
__global__ void vecAddKernel(float* A, float* B, float* C, int n) {
  int i = threadIdx.x + (blockDim.x * blockIdx.x);
  if (i < n) C[i] = A[i] + B[i];
}

int main() {
  …
  //this will create ceil(size/256.0) blocks, each with 256 threads
  //that will each run the vecAddKernel.
  vecAddKernel<<<ceil(size/256.0), 256>>>(d_A, d_B, d_C, size);
  …
}
```

Vector Add Kernel

```
//Computes the vector sum on the GPU.
__global__ void vecAddKernel(float* A, float* B, float* C, int n) {
  int i = threadIdx.x + (blockDim.x * blockIdx.x);
  if (i < n) C[i] = A[i] + B[i];
}

int main() {
  …
  //this will create ceil(size/256.0) blocks, each with 256 threads
  //that will each run the vecAddKernel.
  vecAddKernel<<<ceil(size/256.0), 256>>>(d_A, d_B, d_C, size);

  //this will check to see if there was an error in the kernel
  cudaError_t err = cudaGetLastError();
  if (err != cudaSuccess) printf("Error: %s\n”, cudaGetErrorString(err));

}
```

CUDA keywords

	Executed on the:	Only callable from the:
__device__ float deviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

By default, all functions are __host__ functions, so you typically do not see this frequently in CUDA code.

__device__ functions are for use within kernel functions, they can only be called and used on the GPU.

__global__ functions are accessible from the CPU, however they are executed on the GPU.

Conclusions