

LECTURE 1 - INTRODUCTION TO CUDA C

CUDA C vs. Thrust vs. CUDA Libraries



Introduction to Heterogeneous Parallel Computing

CUDA C vs. CUDA Libs vs. OpenACC

Memory Allocation and Data Movement API Functions

Data Parallelism and Threads

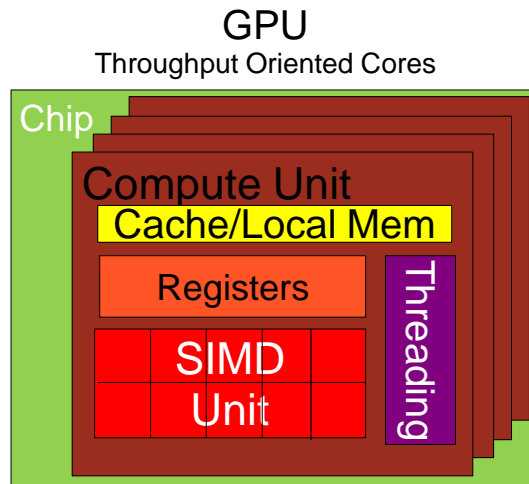
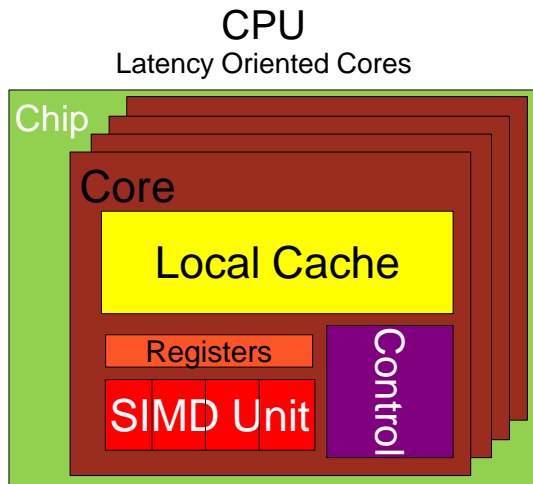


OBJECTIVES

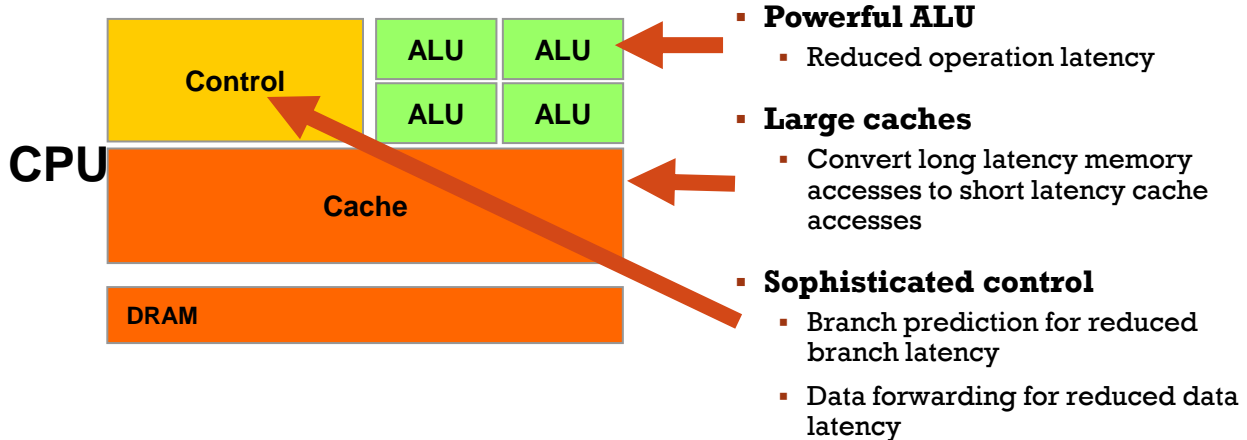
- **To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)**
- **To understand why winning applications increasingly use both types of devices**



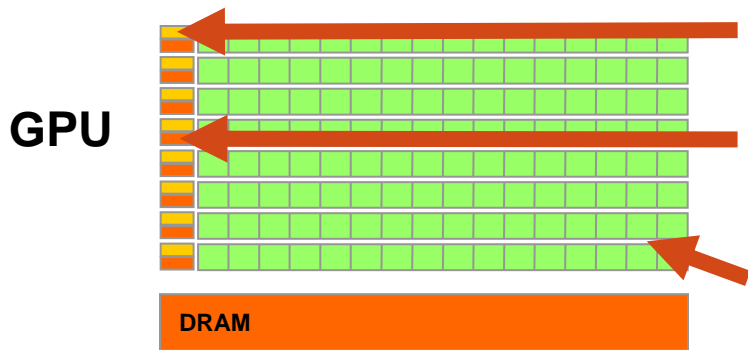
CPU AND GPU ARE DESIGNED VERY DIFFERENTLY



CPUS: LATENCY ORIENTED DESIGN



GPUS: THROUGHPUT ORIENTED DESIGN



- **Small caches**
 - To boost memory throughput
- **Simple control**
 - No branch prediction
 - No data forwarding
- **Energy efficient ALUs**
 - Many, long latency but heavily pipelined for high throughput
- **Require massive number of threads to tolerate latencies**
 - Threading logic
 - Thread state



WINNING APPLICATIONS USE BOTH CPU AND GPU

- **GPUs for parallel parts where throughput wins**
 - GPUs can be 10X+ faster than CPUs for parallel code
- **CPUs for sequential parts where latency matters**
 - CPUs can be 10X+ faster than GPUs for sequential code



Introduction to Heterogeneous Parallel Computing

CUDA C vs. CUDA Libs vs. OpenACC

Memory Allocation and Data Movement API Functions

Data Parallelism and Threads



OBJECTIVE

- **To learn the main venues and developer resources for GPU computing**
- Where CUDA C fits in the big picture



3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

Easy to use
Most Performance

Compiler
Directives

Easy to use
Portable code

Programming
Languages

Most Performance
Most Flexibility



GPU ACCELERATED LIBRARIES

Linear Algebra
FFT, BLAS,
SPARSE, Matrix

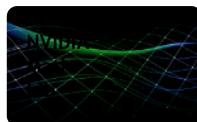


CUDA|tools

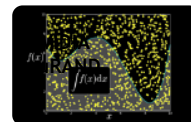


CUSP

Numerical & Math
RAND, Statistics



ArrayFire



Data Struct. & AI
Sort, Scan, Zero Sum



Visual Processing
Image & Video



电子科技大学
University of Electronic Science and Technology of China



VECTOR ADDITION IN THRUST

```
thrust::device_vector<float> deviceInput1(inputLength);  
thrust::device_vector<float> deviceInput2(inputLength);  
thrust::device_vector<float> deviceOutput(inputLength);
```

```
thrust::copy(hostInput1, hostInput1 + inputLength,  
             deviceInput1.begin());
```

```
thrust::copy(hostInput2, hostInput2 + inputLength,  
             deviceInput2.begin());
```

```
thrust::transform(deviceInput1.begin(), deviceInput1.end(),  
                  deviceInput2.begin(),  
                  deviceOutput.begin(),  
                  thrust::plus<float>());
```



OPENACC

- **Compiler directives for C, C++, and FORTRAN**

```
#pragma acc parallel loop  
copyin(input1[0:inputLength],input2[0:inputLength]),  
copyout(output[0:inputLength])  
for(i = 0; i < inputLength; ++i) {  
    output[i] = input1[i] + input2[i];  
}
```



GPU PROGRAMMING LANGUAGES

Numerical analytics ►

MATLAB, Mathematica, LabVIEW

Fortran ►

CUDA Fortran

C ►

CUDA C

C++ ►

CUDA C++

Python ►

PyCUDA, Copperhead, Numba

F# ►

Alea.cuBase



电子科技大学

University of Electronic Science and Technology of China



Introduction to Heterogeneous Parallel Computing

CUDA C vs. CUDA Libs vs. OpenACC

CUDA Programming Model

Memory Allocation and Data Movement API
Functions

Data Parallelism and Threads



PROGRAMMING MODEL

- A **programming model** is an abstraction of the **underlying computer system** that allows for the expression of both **algorithms** and **data structures**.
- **Languages** and **APIs** provide implementations of these abstractions and allow the algorithms and data structures to be put into practice - a programming model exists independently of the choice of both the programming language and the supporting APIs.

Programming mode, Patrick McCormick (LANL) et.al.,
<https://asc.llnl.gov/content/assets/docs/exascale-pmWG.pdf>



SOME DESIGN GOALS

- **Scale to 100s of cores, 1000s of parallel threads**
- **Let programmers focus on parallel algorithms**
 - not mechanics of a parallel programming language.
- **Enable heterogeneous systems (i.e., CPU+GPU)**
 - CPU & GPU are separate devices with separate DRAMs

KEY PARALLEL ABSTRACTIONS

- **Hierarchy of concurrent threads**
- **Lightweight synchronization primitives**
- **Shared memory model for cooperating threads**

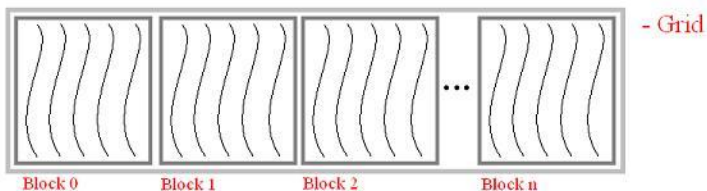
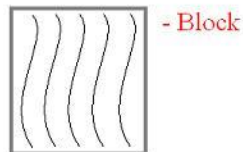
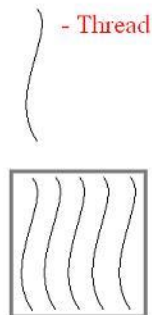


THREAD HIERARCHY

Thread – Distributed by the CUDA runtime
(identified by threadIdx)

Warp – A scheduling unit of up to 32 threads

Block – A user defined group of 1 to 512 threads.
(identified by blockIdx)



Grid – A group of one or more blocks. A grid is created for each CUDA kernel function



CUDA MEMORY HIERARCHY

- The CUDA platform has three primary memory types

Register – per thread memory for automatic variables and register spilling.

Shared Memory – per block low-latency memory to allow for intra-block data sharing and synchronization. Threads can safely share data through this memory and can perform barrier synchronization through `__syncthreads()`

Global Memory – device level memory that may be shared between blocks or grids

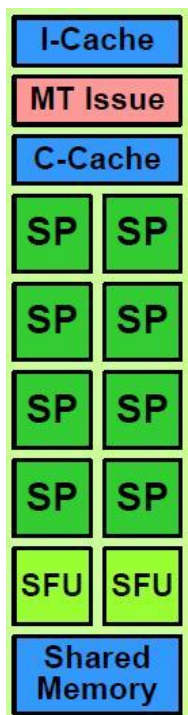


CUDA HARDWARE

- **The primary components of the Tesla architecture are:**
 - Streaming Multiprocessor (The 8800 has 16)
 - Scalar Processor
 - Memory hierarchy
 - Interconnection network
 - Host interface



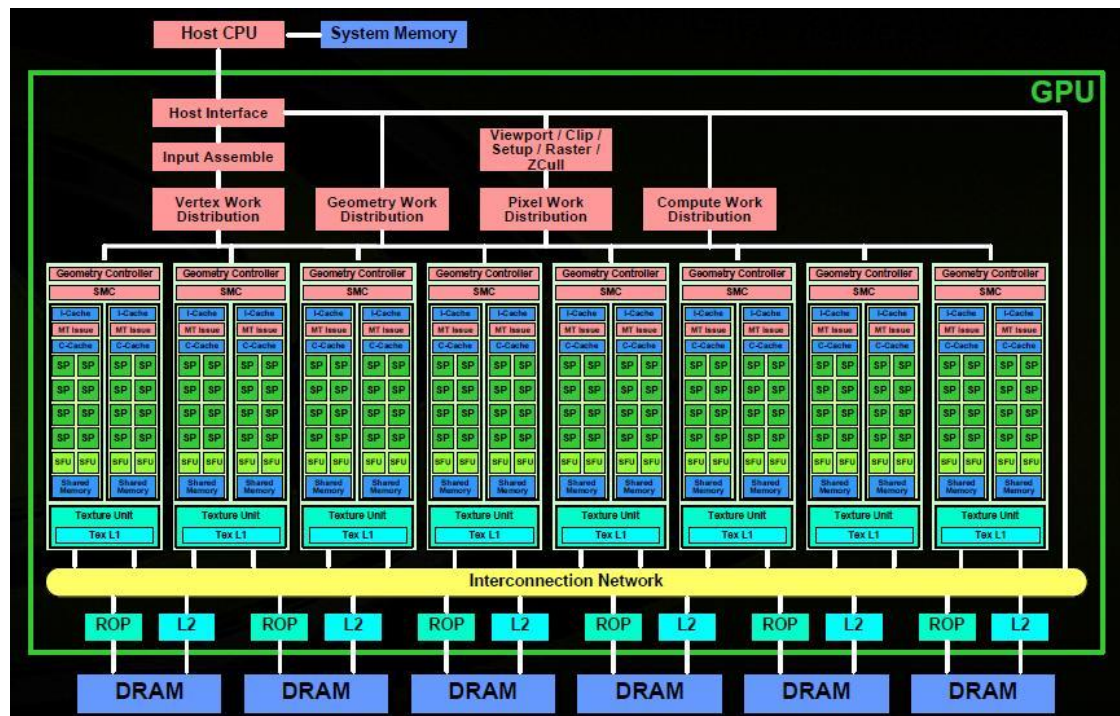
STREAMING MULTIPROCESSOR (SM)



- Each SM has 8 Scalar Processors (SP)
- IEEE 754 32-bit floating point support (incomplete support)
- Each SP is a 1.35 GHz processor (32 GFLOPS peak)
- Supports 32 and 64 bit integers
- 8,192 dynamically partitioned 32-bit registers
- Supports 768 threads in hardware (24 SIMT warps of 32 threads)
- Thread scheduling done in hardware
- 16KB of low-latency shared memory
- 2 Special Function Units (reciprocal square root, trig functions, etc)



THE GPU



Introduction to Heterogeneous Parallel Computing

CUDA C vs. CUDA Libs vs. OpenACC

CUDA Programming Model

Memory Allocation and Data Movement API
Functions

Data Parallelism and Threads

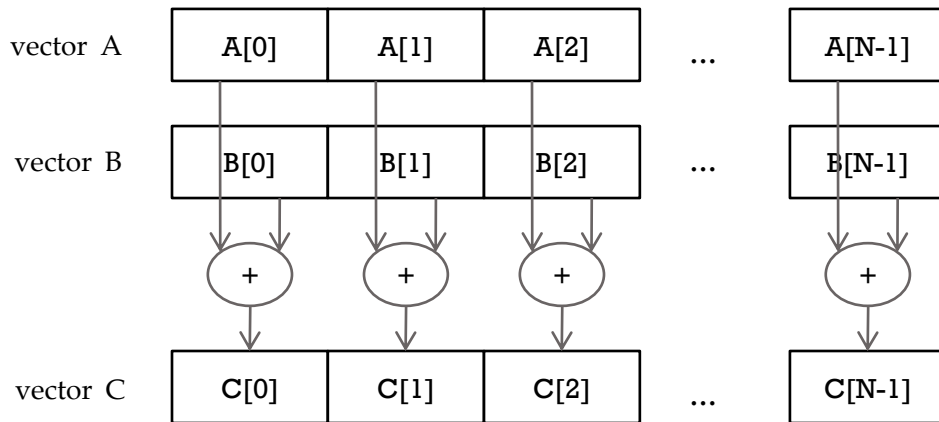


OBJECTIVE

- **To learn the basic API functions in CUDA host code**
 - Device Memory Allocation
 - Host-Device Data Transfer



DATA PARALLELISM - VECTOR ADDITION EXAMPLE



VECTOR ADDITION – TRADITIONAL C CODE

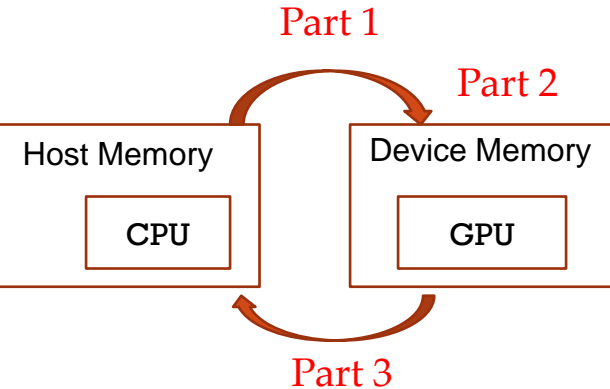
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...

    (h_A, h_B, h_C, N);
}
```



HETEROGENEOUS COMPUTING VECADD CUDA HOST CODE



```
#include <cuda.h>
```

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)  
{
```

```
    int size = n* sizeof(float);
```

```
    float *d_A, *d_B, *d_C;
```

```
    // Part 1
```

```
    // Allocate device memory for A, B, and C
```

```
    // copy A and B to device memory
```

```
    // Part 2
```

```
    // Kernel launch code – the device performs the actual  
    vector addition
```

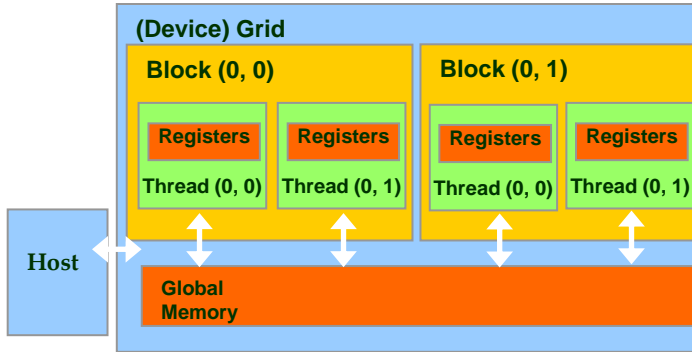
```
    // Part 3
```

```
    // copy C from the device memory
```

```
}
```



PARTIAL OVERVIEW OF CUDA MEMORIES

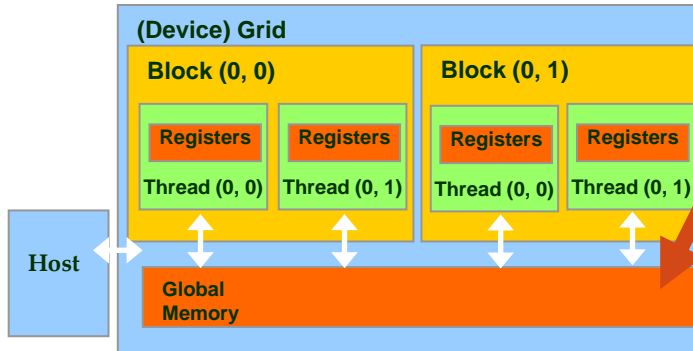


- **Device code can:**
 - R/W per-thread **registers**
 - R/W all-shared **global memory**
- **Host code can**
 - Transfer data to/from per grid **global memory**

We will cover more memory types and more sophisticated memory models later.



CUDA DEVICE MEMORY MANAGEMENT API FUNCTIONS



- **cudaMalloc()**

- Allocates an object in the device global memory

- Two parameters

- **Address of a pointer** to the allocated object
 - **Size of** allocated object in terms of bytes

- **cudaFree()**

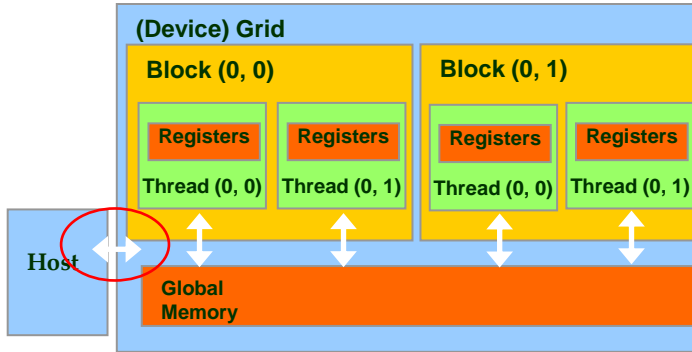
- Frees object from device global memory

- One parameter

- **Pointer** to freed object



HOST-DEVICE DATA TRANSFER API FUNCTIONS



- **cudaMemcpy()**
 - memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer
- Transfer to device is asynchronous



VECTOR ADDITION HOST CODE

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)  
{  
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;  
  
    cudaMalloc((void **) &d_A, size);  
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
    cudaMalloc((void **) &d_B, size);  
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);  
    cudaMalloc((void **) &d_C, size);  
  
    // Kernel invocation code – to be shown later  
  
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);  
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);  
  
}
```



IN PRACTICE, CHECK FOR API ERRORS IN HOST CODE

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

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



Introduction to Heterogeneous Parallel Computing

CUDA C vs. CUDA Libs vs. OpenACC

Memory Allocation and Data Movement
API Functions

Data Parallelism and Threads

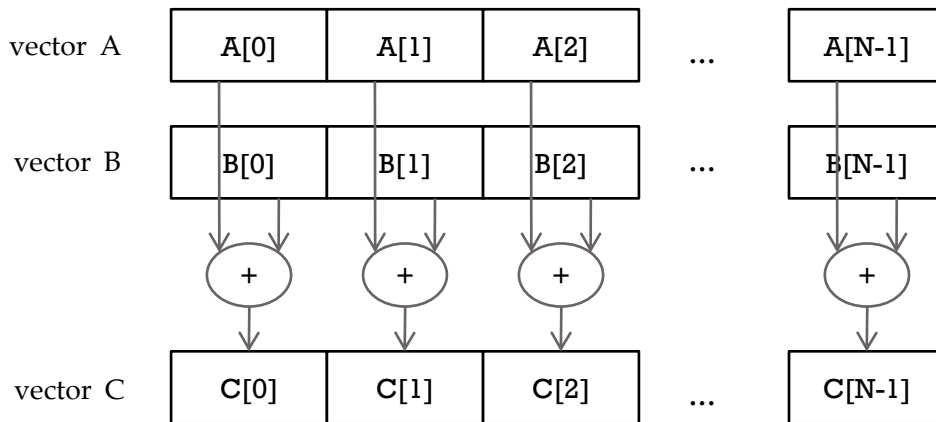


OBJECTIVE

- **To learn about CUDA threads, the main mechanism for exploiting of data parallelism**
 - Hierarchical thread organization
 - Launching parallel execution
 - Thread index to data index mapping



DATA PARALLELISM - VECTOR ADDITION EXAMPLE



CUDA EXECUTION MODEL

- **Heterogeneous** host (CPU) + device (GPU)
application C program

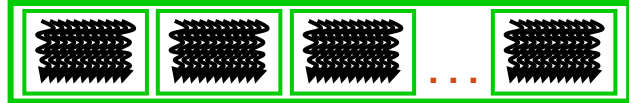
- Serial parts in **host** C code
- Parallel parts in **device** SPMD kernel code

Serial Code (host)



Parallel Kernel (device)

KernelA<<< nBlk, nTid >>>(args);

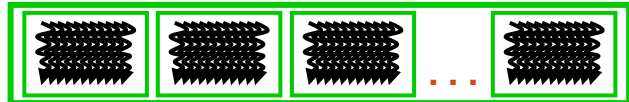


Serial Code (host)



Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);



电子科技大学

University of Electronic Science and Technology of China



A PROGRAM AT THE ISA LEVEL

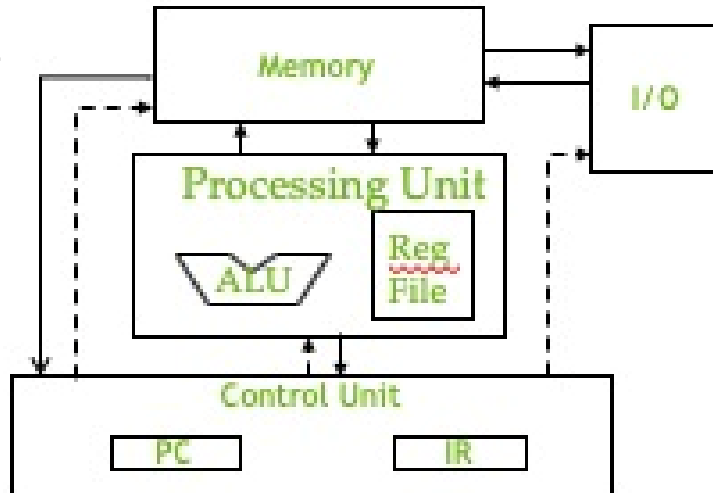
- **A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.**
 - Both CPUs and GPUs are designed based on (different) instruction sets
- **Program instructions operate on data stored in memory and/or registers.**



A THREAD AS A VON-NEUMANN PROCESSOR

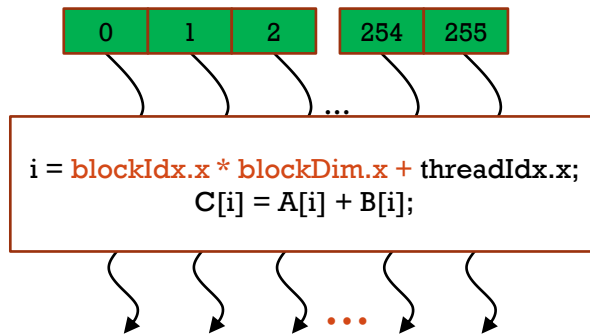
A thread is a “virtualized” or
“abstracted”

Von-Neumann Processor

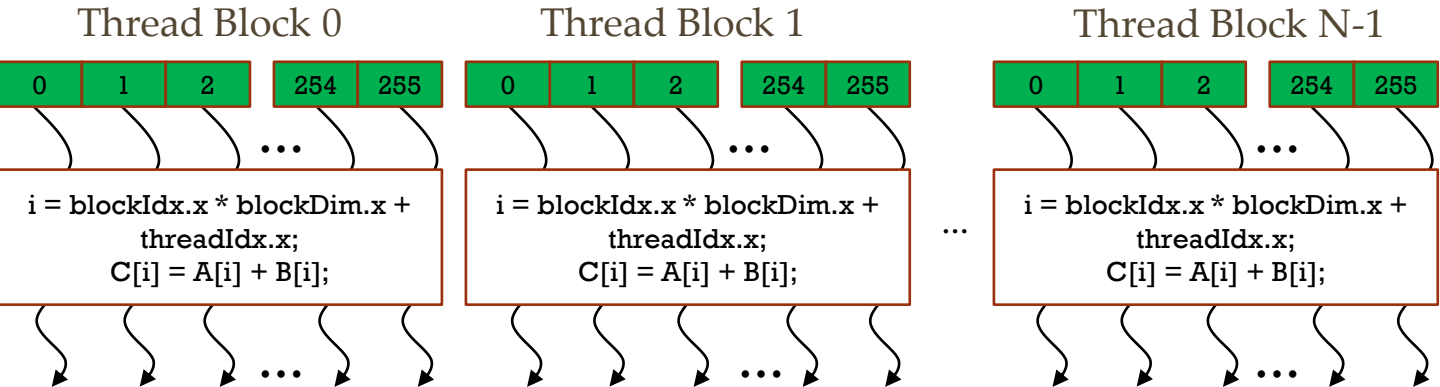


ARRAYS OF PARALLEL THREADS

- A CUDA kernel is executed by a **grid** (array) of threads
 - All threads in a grid run the same kernel code (**SPMD**, Single Program Multiple Data)
 - Each thread has indexes that it uses to compute memory addresses and make control decisions



THREAD BLOCKS: SCALABLE COOPERATION

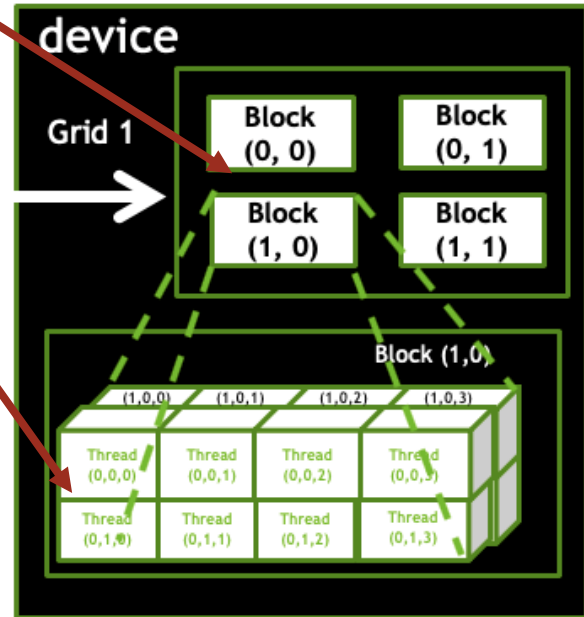


- **Divide thread array into multiple blocks**
 - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
 - **Threads in different blocks do not interact**



BLOCKIDX AND THREADIDX

- Each thread uses indices to decide what data to work on
 - **blockIdx**: 1D, 2D, or 3D (CUDA 4.0)
 - **threadIdx**: 1D, 2D, or 3D
- Simplifies **memory addressing** when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes
 - ...



NVCC COMPILER

- **NVIDIA provides a CUDA-C compiler**
 - **nvcc**
- **NVCC compiles device code then forwards code on to the host compiler (e.g. g++)**
- **Can be used to compile & link host only applications**

