

source: NVIDIA GPU Teaching Kit

IF3230

Sistem Paralel dan Terdistribusi

CUDA

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)

Judhi Santoso (judhi@staff.stei.itb.ac.id)

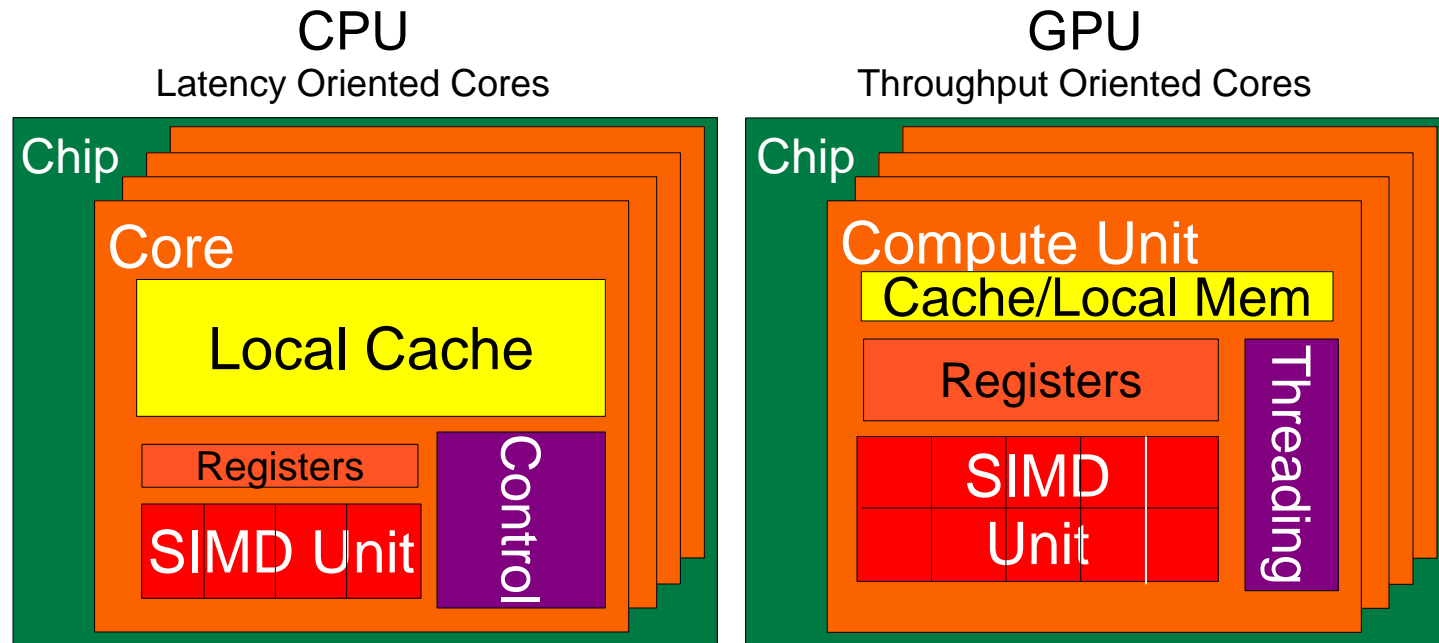
Anggrahita Bayu Sasmita (angga@staff.stei.itb.ac.id)

Februari 2022

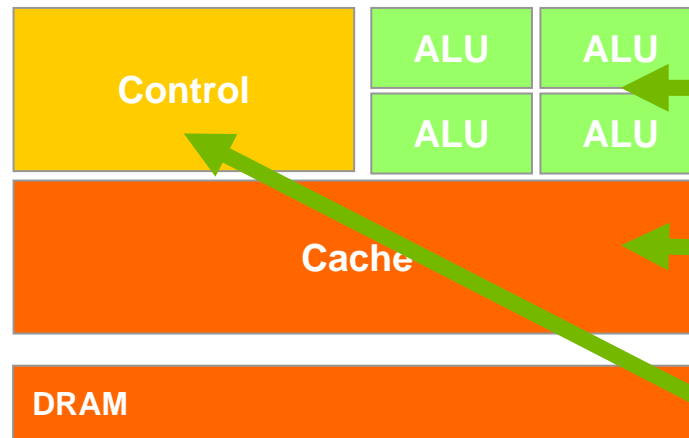
Objectives

- 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

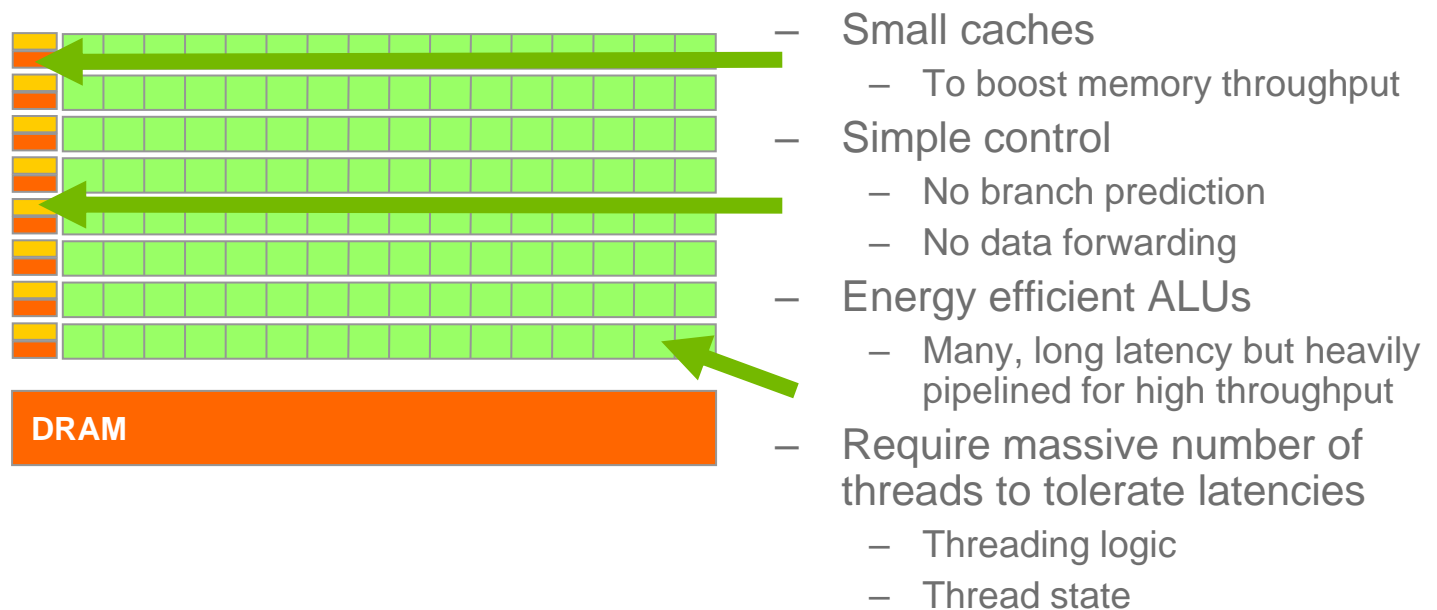


CPUs: Latency Oriented Design



- Powerful ALU
 - Reduced operation latency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



Winning Applications Use Both CPU and GPU

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

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

Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU Accelerated Libraries

Linear Algebra

FFT, BLAS,
SPARSE, Matrix



CUDA|tools



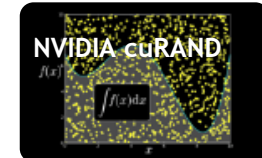
CUSP

Numerical & Math

RAND, Statistics



ArrayFire



Data Struct. & AI

Sort, Scan, Zero Sum



Visual Processing

Image & Video



NVIDIA
Video
Encode



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>());
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

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];  
}
```

Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics ▶

MATLAB Mathematica, LabVIEW

Fortran ▶

CUDA Fortran

C ▶

CUDA C

C++ ▶

CUDA C++

Python ▶

PyCUDA, Copperhead, Numba

F# ▶

Alea.cuBase

CUDA - C

Applications

Libraries

Compiler
Directives

Programming
Languages

Easy to use
Most Performance

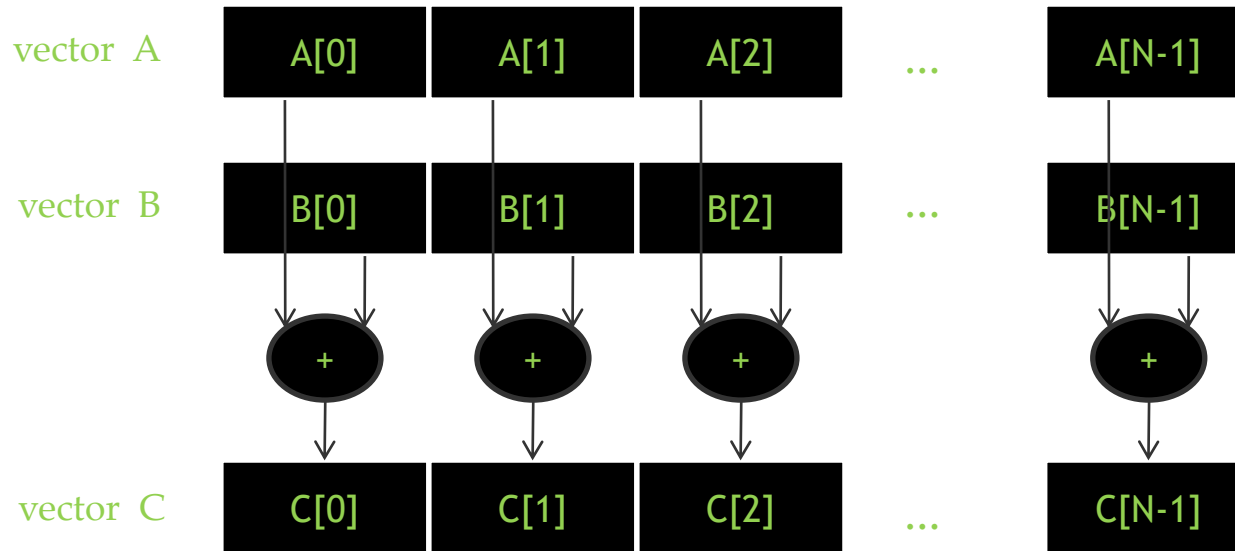
Easy to use
Portable code

Most Performance
Most Flexibility

Objective

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

Data Parallelism - Vector Addition Example

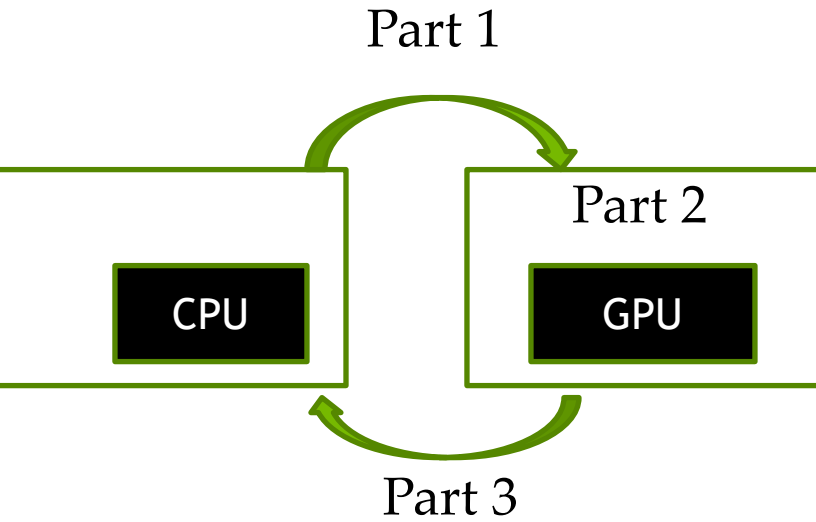


Vector Addition – Traditional C Code

```
// 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
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code

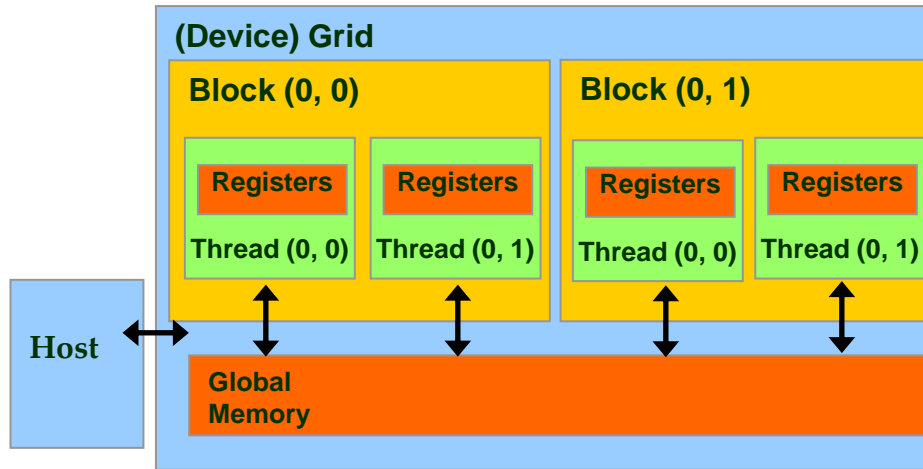


```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

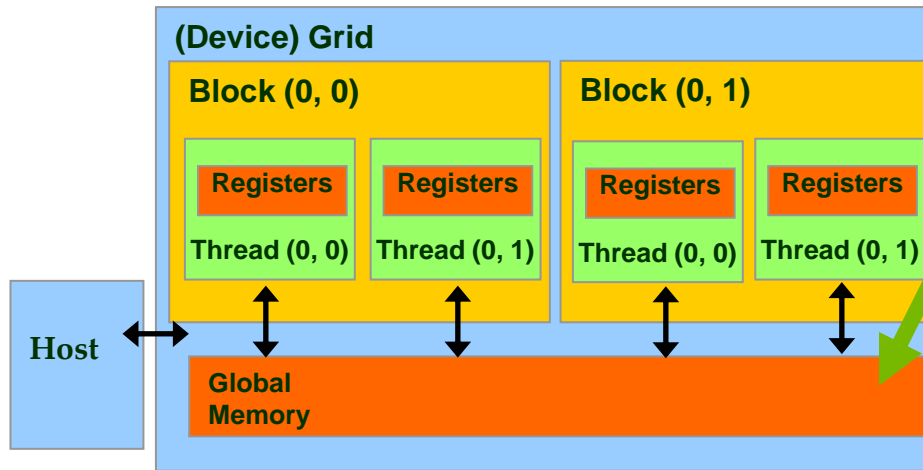
Partial Overview of CUDA Memories



- 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**

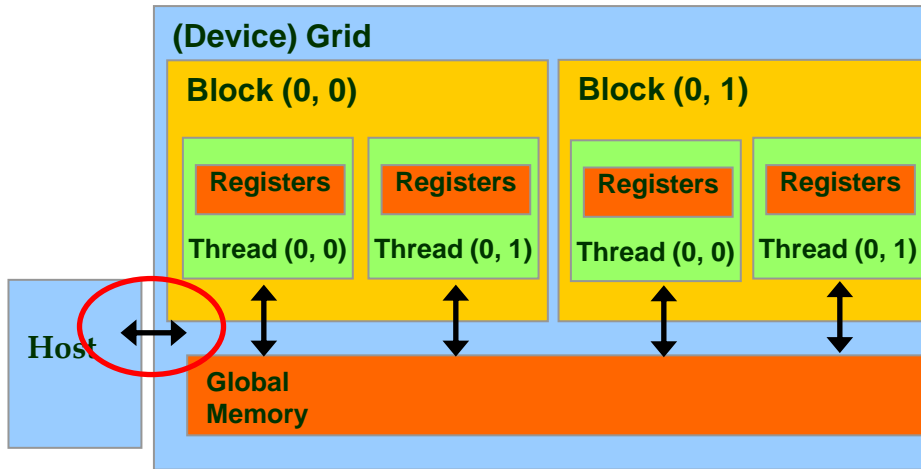
We will cover more memory types and more sophisticated memory models later.

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

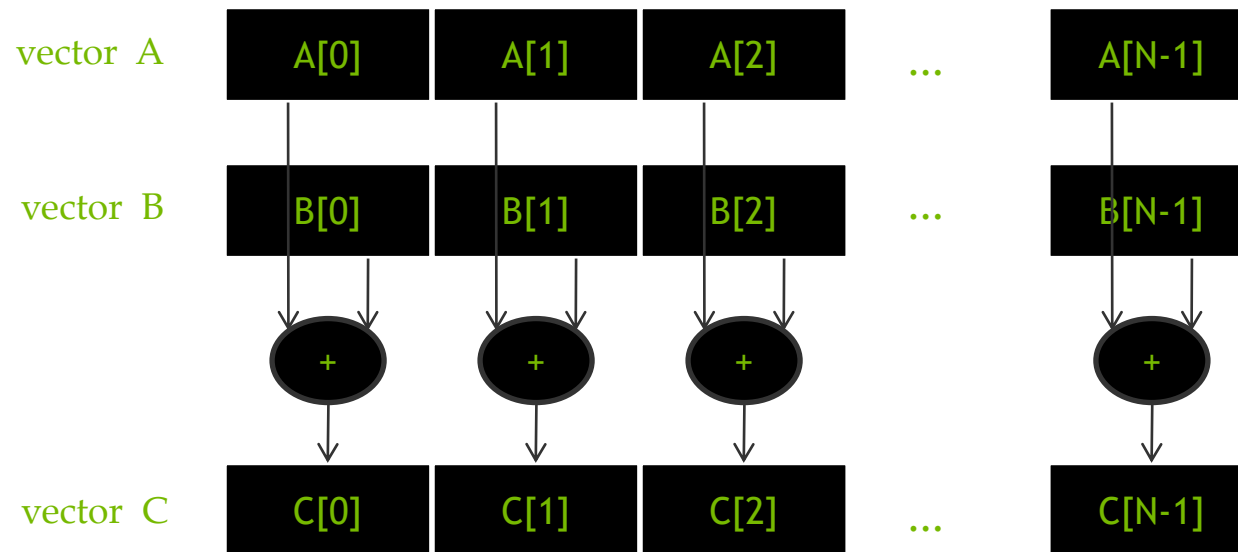
```
cudaError_t err = cudaMalloc((void **) &d_A, size);

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

Objective

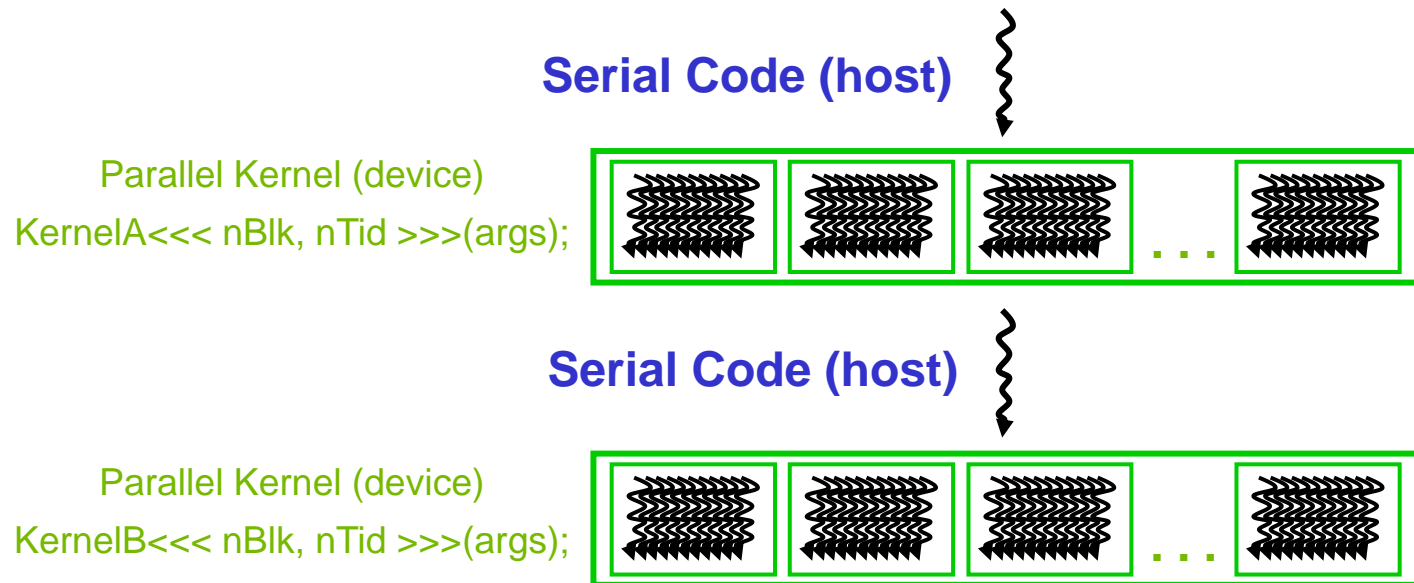
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
 - Hierarchical thread organization
 - Launching parallel execution
 - Thread index to data index mapping

Data Parallelism - Vector Addition Example

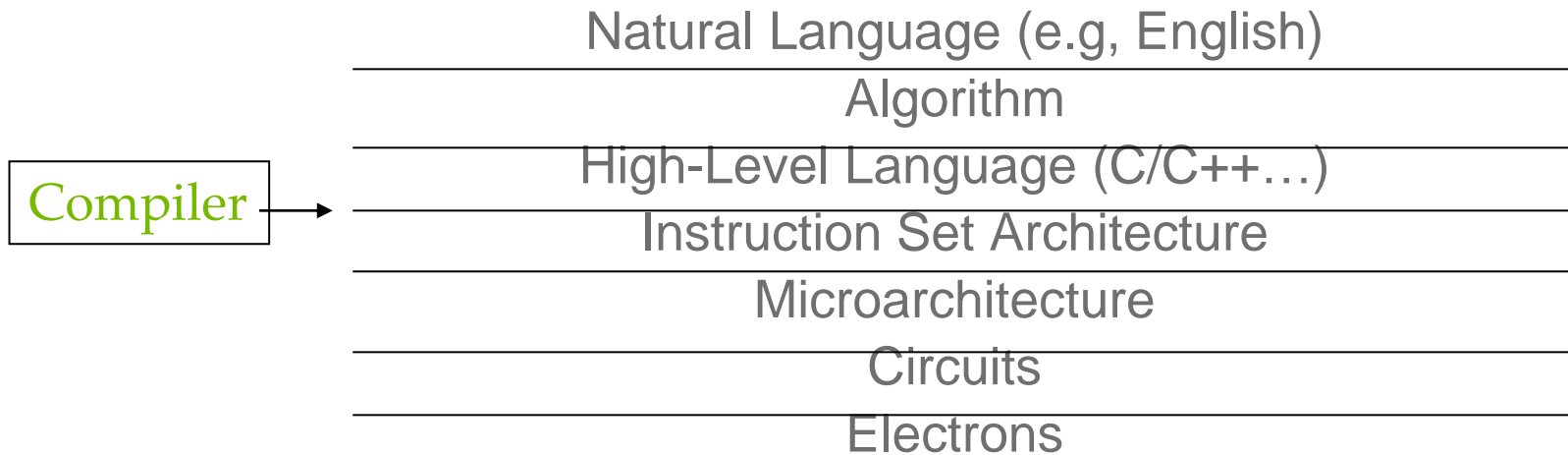


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
 - Serial parts in **host** C code
 - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



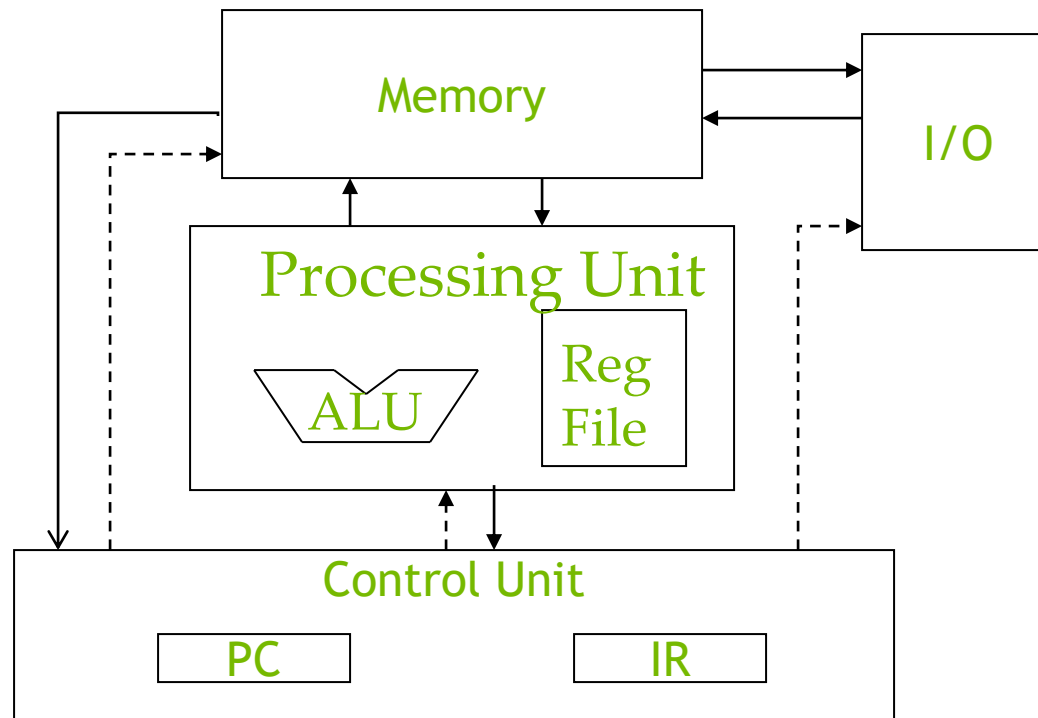
©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

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.

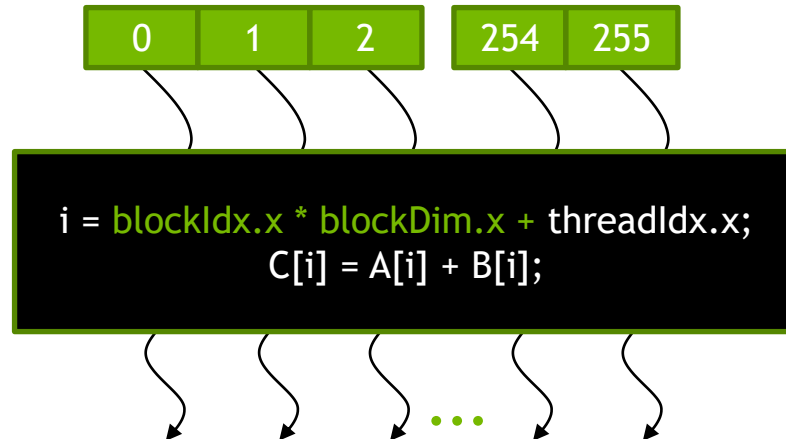
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or
“abstracted”
Von-Neumann Processor

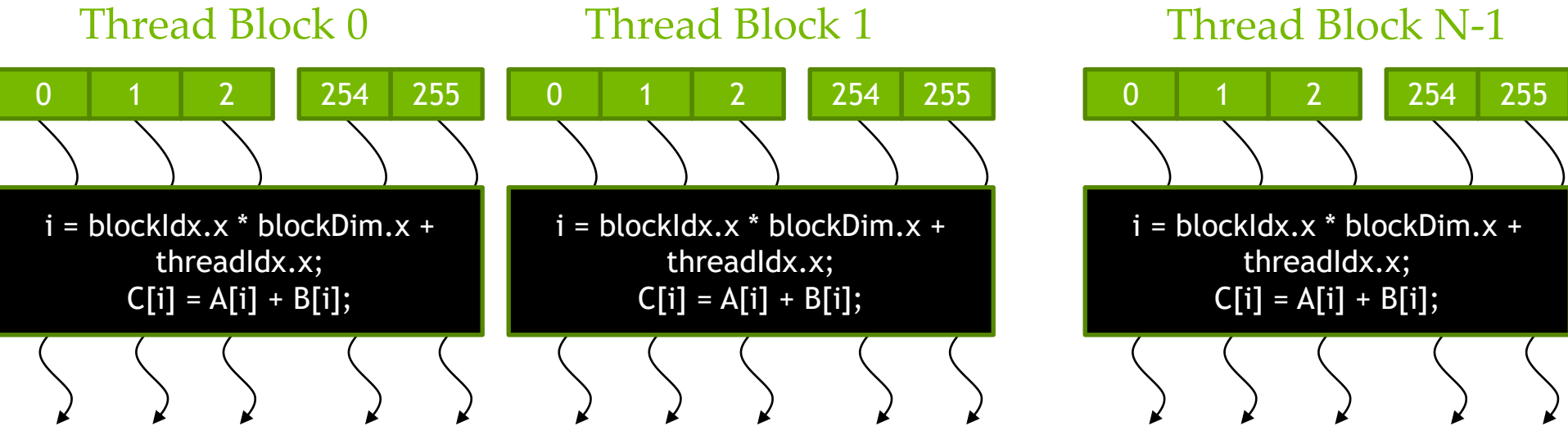


Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
 - All threads in a grid run the same kernel code (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
 - ...

