# Week # 16 - CUDA

# CPU

## Single Core CPUs

- Flexible
- Performant
- How to squeeze more functionality?
  - Power Hungary
  - Memory CPU speed disparity
  - Instruction Level Parallelism
    - Pipelining
    - Super Scalar

"Power Wall + Memory Wall + ILP Wall = Brick Wall"

## Multi-Cores CPUs

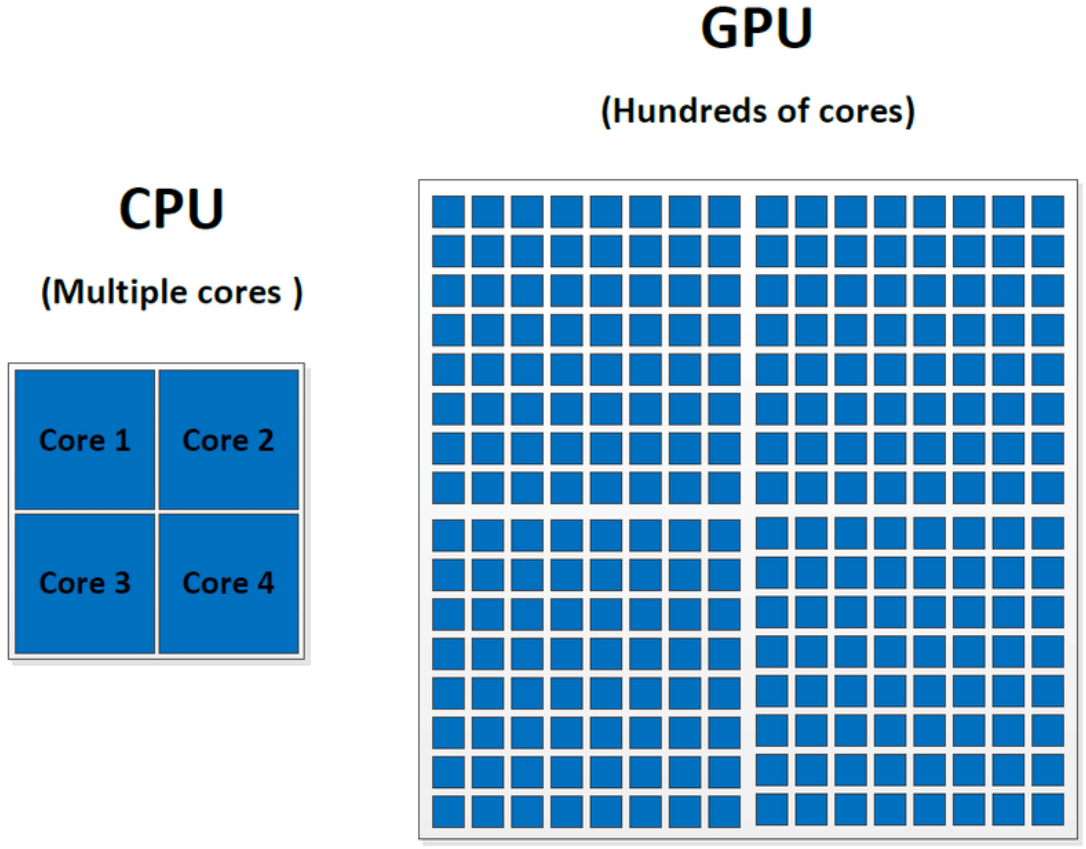
- Instead of adding complexity to a single core we scale cores in the same dimensions of the silicon
- Current Multi-Cores (Discuss)

- Do we need more?
  - Vector Processing Units (VPUs),
  - Graphic Processing Units (GPUs),
  - Associative Processing Units (APUs),
  - Tensor Processing Units (TPUs)

  - Field Programmable Gate Arrays (FPGAs),
  - Quantum Processing Units (QPUs)

# Graphic Processing Units (GPUs)

- **More HW for Computation**
  - **Many Cores**
- **More power efficient**
- **Restricted Programming Model**
- GPUs were originally designed to accelerate the rendering of 3D graphics. Over time, they became more flexible and programmable, enhancing their capabilities.
- This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques.
- Other developers also began to tap the power of GPUs to dramatically accelerate additional workloads in high performance computing (HPC), deep learning, and more.



# Latency vs Throughput

Latency

- Minimize time of one task
- Metric: seconds
- Focus of CPU

Throughput

- Maximize stuff per time
- Metric: jobs/hour, pixels/sec
- Focus of GPUs

# Latency vs Throughput

- Latency-Amount of time to complete a task(time , seconds)
- Throughput-Task completed per unit time(Jobs/Hour)


Your goals are not aligned with post office goals

Your goal: Optimize for Latency (want to spend a little time)

Post office: Optimize for throughput (number of customers they serve per a day)

**CPU:** Optimize for **latency**(minimize the time elapsed of one particular task)

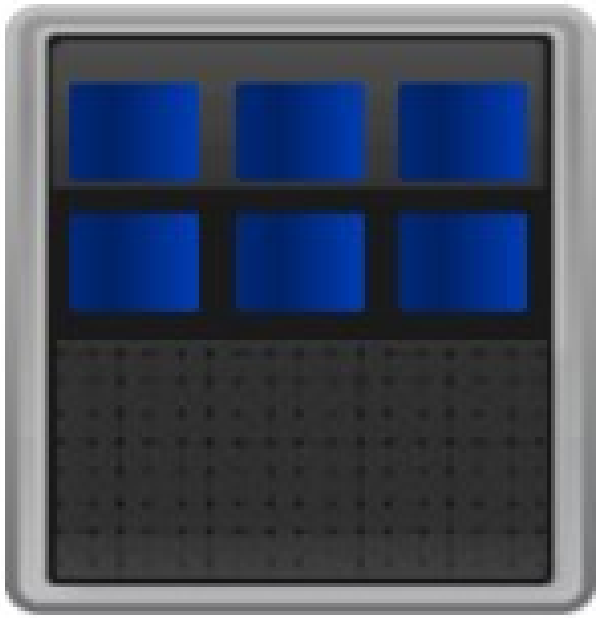
**GPU:** Chose to Optimize for **throughput**

# NVIDIA Tesla A100 with 54 Billion Transistors


Announced and released on May 14, 2020 was the Ampere-based A100 accelerator. With 7nm technologies, the A100 has 54 billion transistors and features 19.5 teraflops of FP32 performance, 6912 CUDA cores, 40GB of graphics memory, and 1.6TB/s of graphics memory bandwidth. The A100 80GB model announced in Nov 2020, has 2.0TB/s graphics memory bandwidth

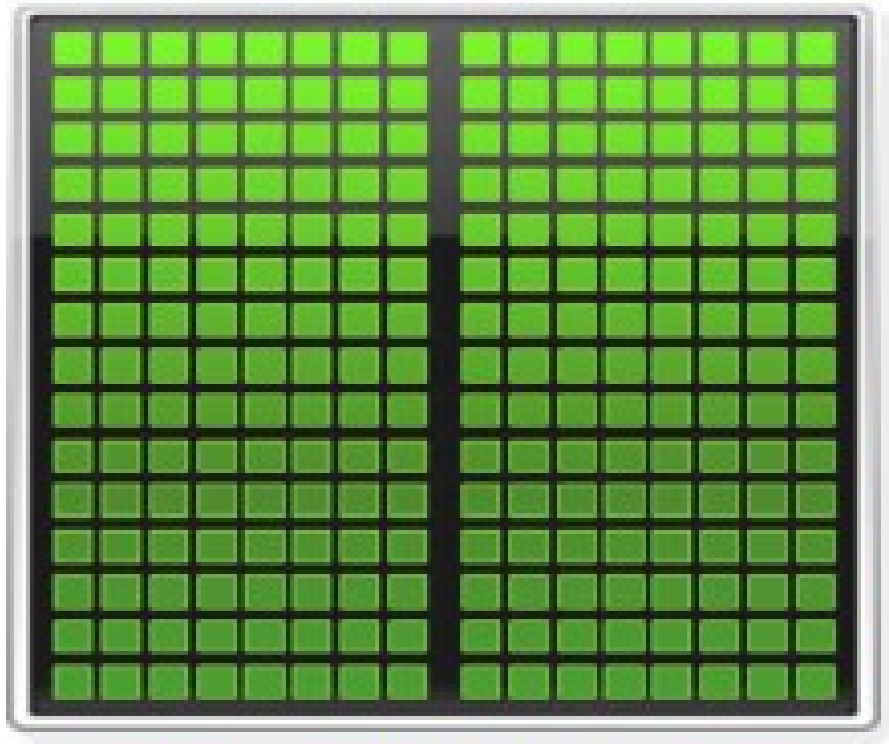
# CPU

Optimized for Serial Tasks



# GPU Accelerator

Optimized for Many Parallel Tasks



CHAPTER

# Introduction to Data Parallelism and CUDA C

# 3

**CHAPTER OUTLINE**



Our main objective is to teach the key concepts involved in writing massively parallel programs in a heterogeneous computing system. This requires many code examples expressed in a reasonably simple language that supports massive parallelism and heterogeneous computing. We have chosen CUDA C for our code examples and exercises. CUDA C is an extension to the popular C programming language[1] with new keywords and application programming interfaces for programmers to take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPU's.

- In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel.

- When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

## TASK PARALLELISM VERSUS DATA PARALLELISM

Data parallelism is not the only type of parallelism widely used in parallel programming. Task parallelism has also been used extensively in parallel programming. Task parallelism is typically exposed through task decomposition of applications. For example, a simple application may need to do a vector addition and a matrix–vector multiplication. Each of these would be a task. Task parallelism exists if the two tasks can be done independently.

In large applications, there are usually a larger number of independent tasks and therefore a larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks includes vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce CUDA streams.

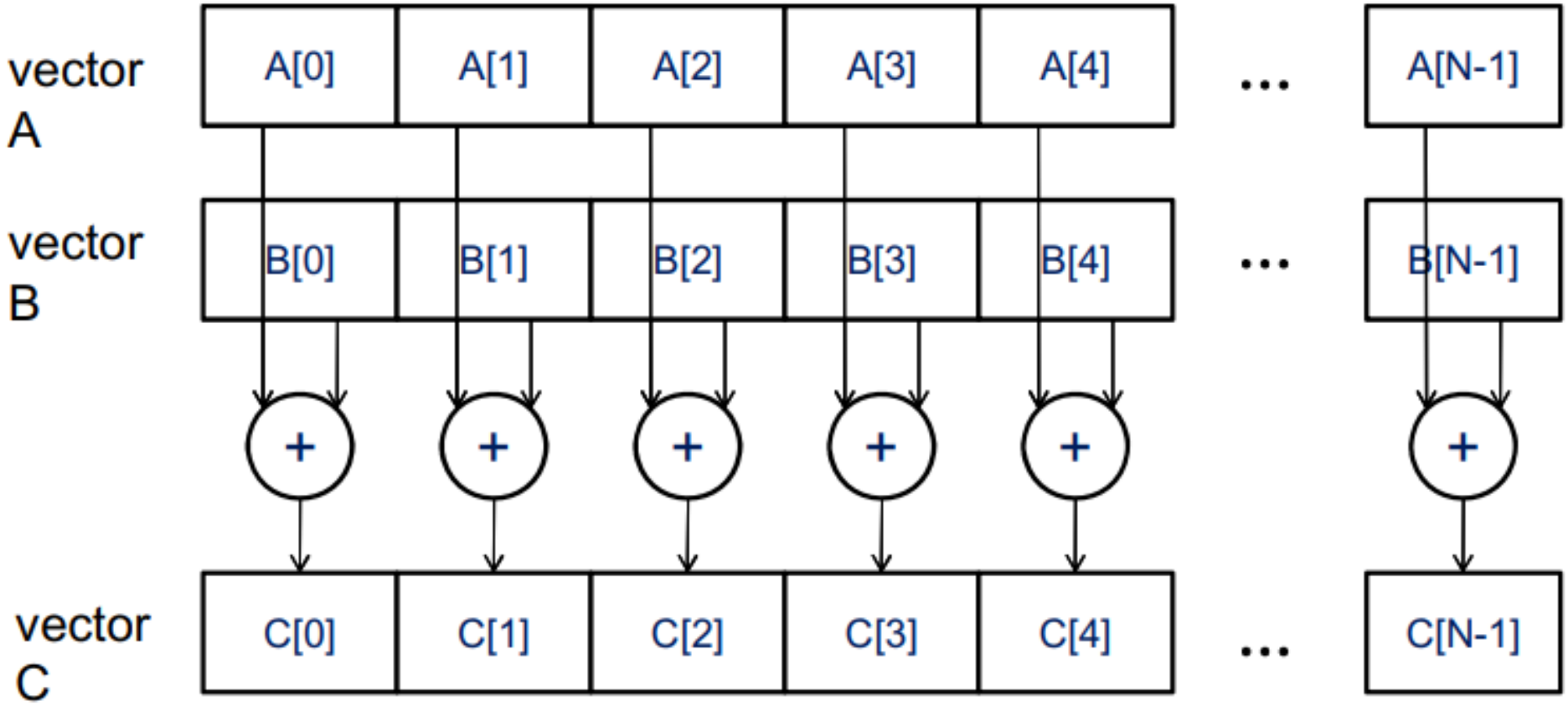
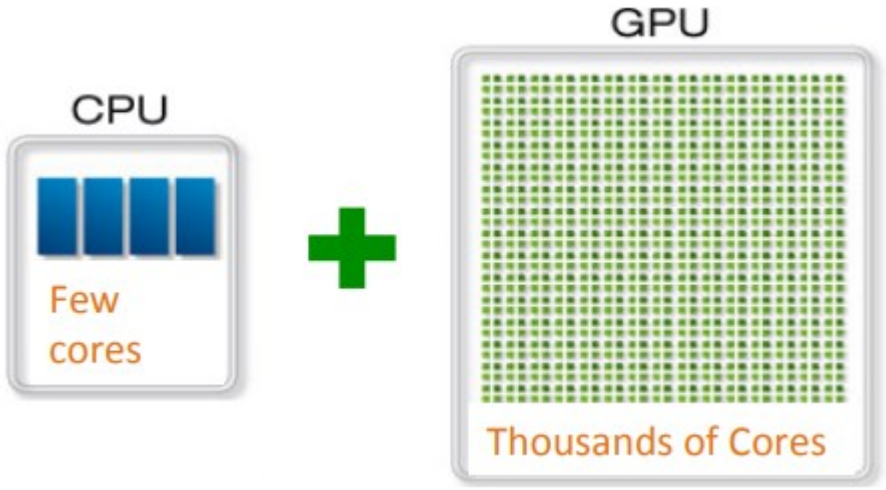



**FIGURE 3.1**

Data parallelism in vector addition.

# Hybrid CPU GPU programming

- A heterogeneous platform = **CPU** + **GPU**
  - Most solutions work for other/multiple accelerators
- An application workload = an application + its input dataset
- Workload partitioning = workload distribution among the processing units of a heterogeneous system



## 3.2 CUDA PROGRAM STRUCTURE

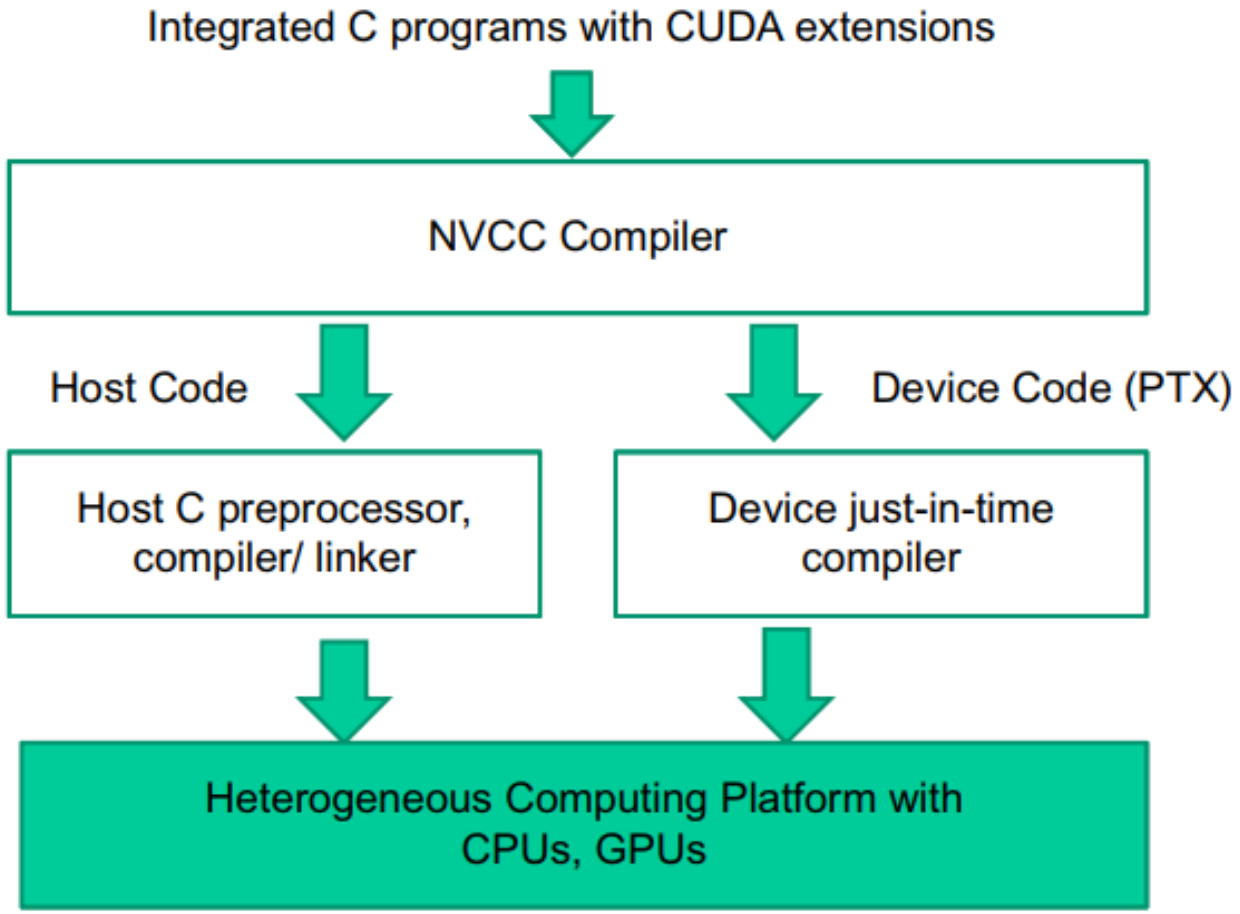
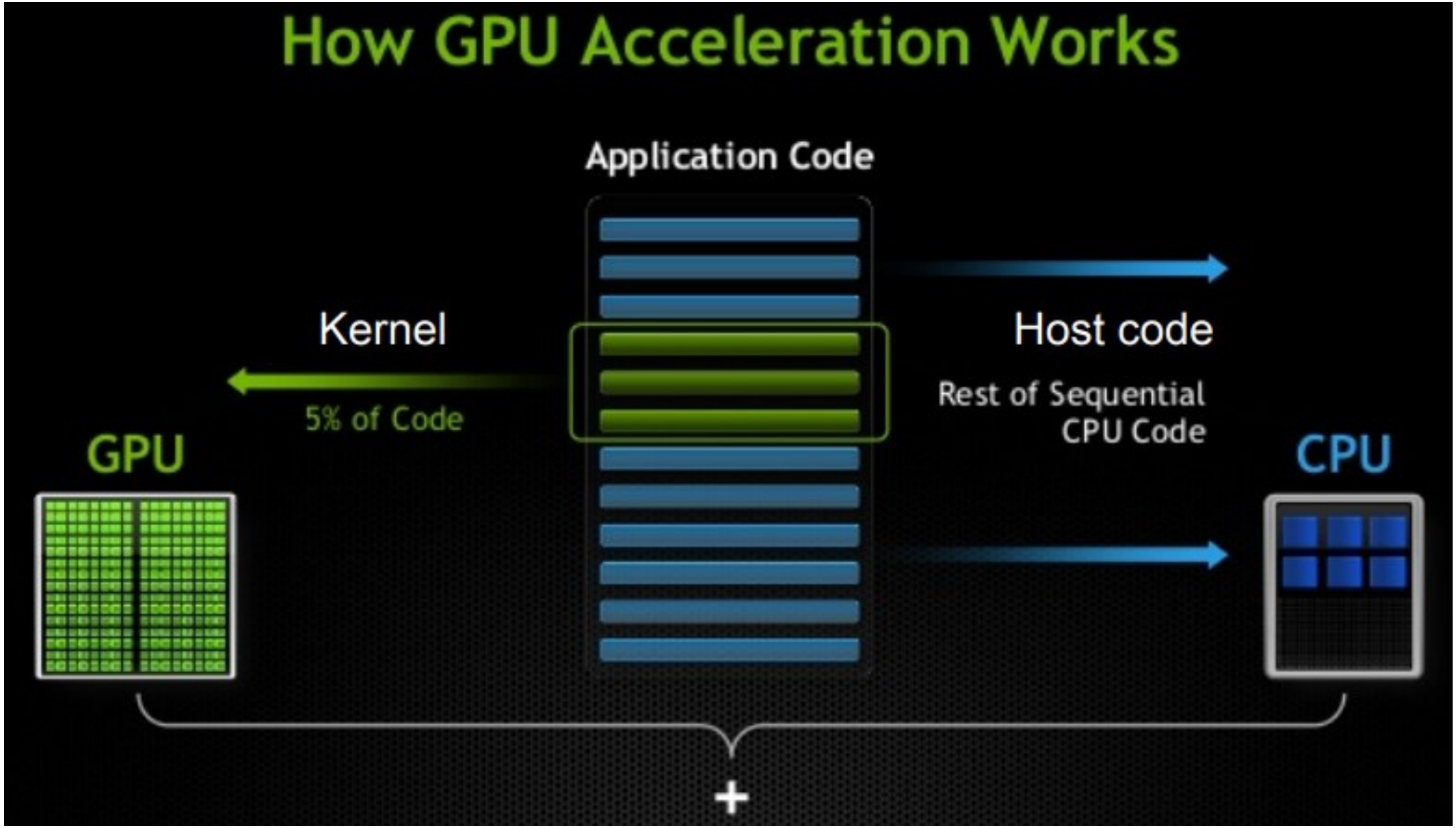
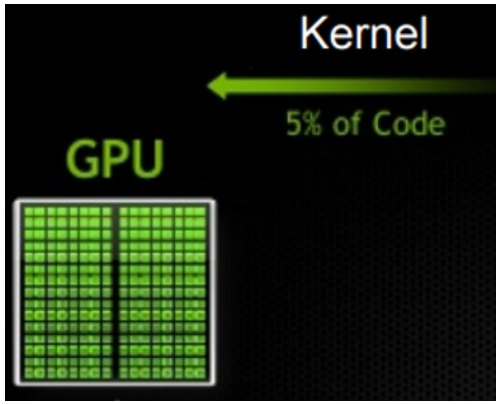



**FIGURE 3.2**

Overview of the compilation process of a CUDA program.


How GPU Acceleration Works
Application Code
Kernel
5% of Code
Host code
Rest of Sequential
CPU Code
GPU
CPU
+

# Kernel Execution



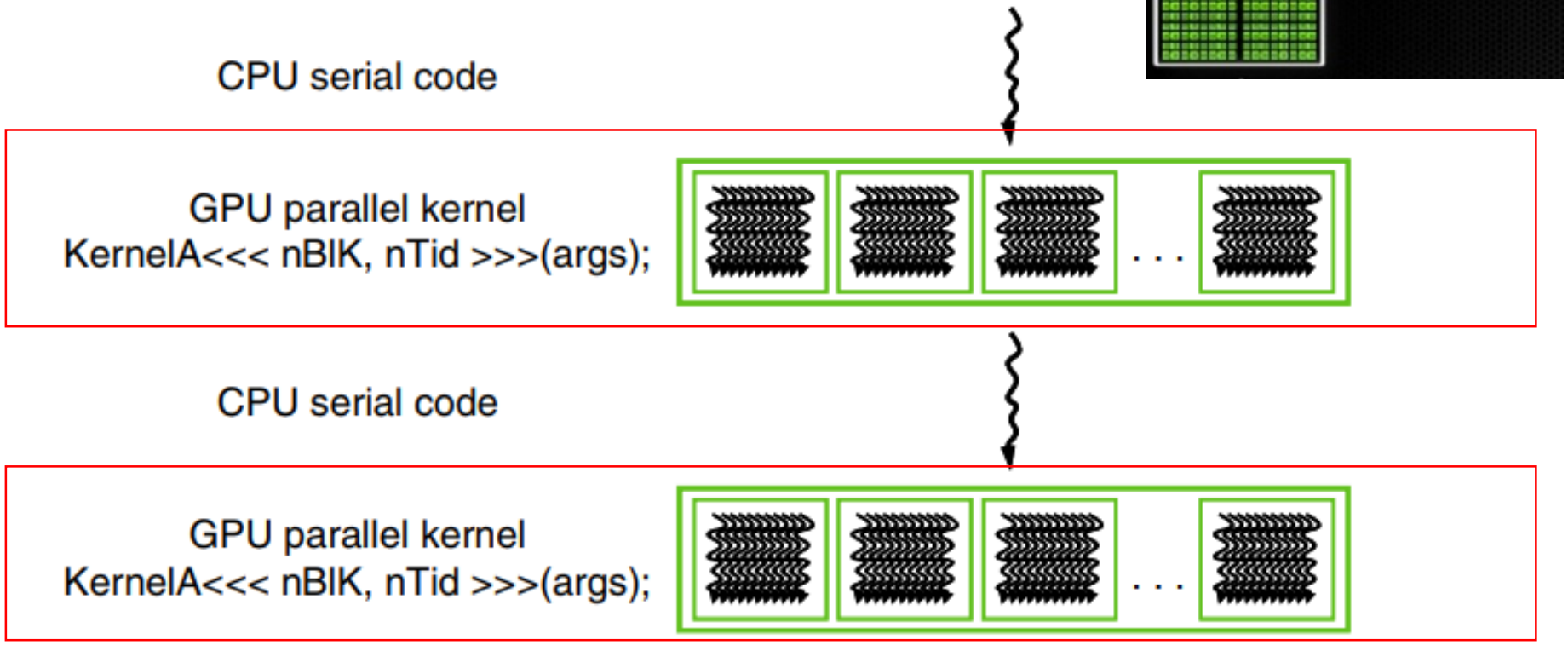




**FIGURE 3.3**

Execution of a CUDA program.

## 3.2 CUDA PROGRAM STRUCTURE

Typical CUDA program

1.1 CPU allocates GPU memory (`cudaMalloc`)
1.2 CPU copies data to GPU (`cudaMemcpy`)
1.3 CPU launches kernel on GPU; parallelism expressed here
1.4 CPU copies results from GPU (`cudaMemcpy`)

# 3.3 A VECTOR ADDITION KERNEL

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  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 each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

**FIGURE 3.4**

A simple traditional vector addition C code example.

# 3.3 A VECTOR ADDITION KERNEL

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
  int size = n* sizeof(float);
   float *A_d, *B_d, *C_d;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

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

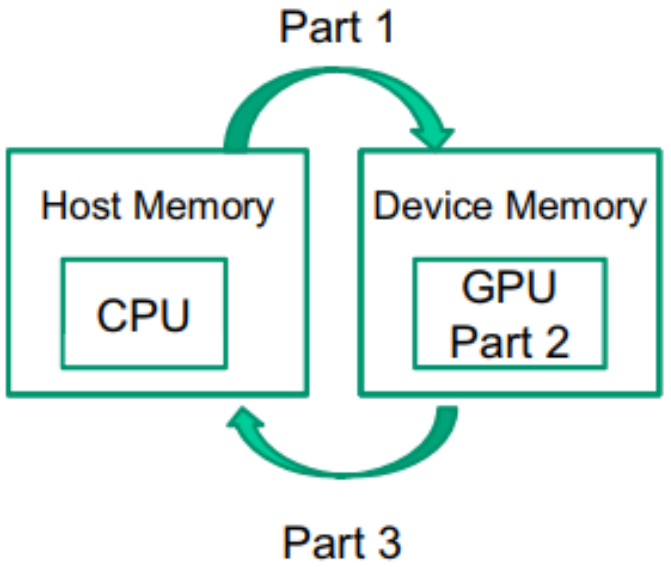



**FIGURE 3.5**

Outline of a revised `vecAdd()` function that moves the work to a device.

# How GPU Accelerate Compute?

Big Idea

1. Kernel looks like a serial program; says nothing about parallelism
2. Write as if run on one thread
3. Will actually run on many threads (CPU says how many—hundreds, thousands, millions!)
4. Each thread knows its thread index; can do different parts of the computation

# Conversion of Vector Addition C code to CUDA C

Serial C Code template

```c
#define N 10000000

void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

int main(){
    float *a, *b, *out;

    // Allocate memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }

    // Main function
    vector_add(out, a, b, N);
}
```

# CUDA C Code

```
#include <stdio.h>
#include <stdlib.h>
#define N 100000
__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];
}
void main(){
  float *a, *b, *out;
  float *d_a, *d_b, *d_out;

  a = (float*)malloc(sizeof(float) * N);
  b = (float*)malloc(sizeof(float) * N);
  out = (float*)malloc(sizeof(float) * N);
  for (int i=0; i < N; i++) { // fill vector a and b with random values
     a[i] = (float) rand(); b[i] = (float) rand();
  }
  cudaMalloc((void**)&d_a, sizeof(float) * N);   // Allocate device memory for d_a, d_b and d_out
  cudaMalloc((void**)&d_b, sizeof(float) * N);
  cudaMalloc((void**)&d_out, sizeof(float) * N);
  //
  cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice); // Transfer data from host to device memory d_a, d_b
  cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
  vector_add<<<1,1>>>(d_out, d_a, d_b, N); // CUDA kernel. This syntax <<< 1, 1 >> is for thread creation.
  cudaMemcpy(out,d_out, sizeof(float) * N, cudaMemcpyDeviceToHost); // Transfer data from device to host memory

  for (int i=0; i < N; i++) {
     printf ("%f,", out[i]);
  }
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_out); // Cleanup after kernel execution
  free(a); free(b); free(out); // clearnup allocated memory
}
```

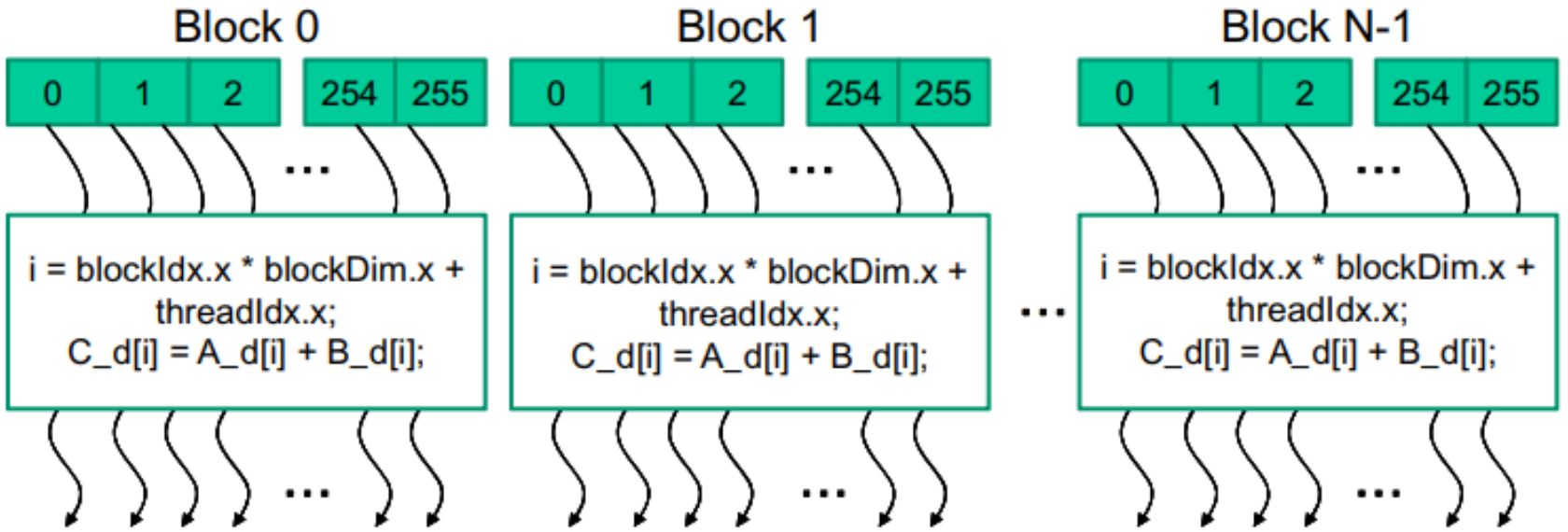



**FIGURE 3.10**

All threads in a grid execute the same kernel code.

- When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy. Each grid is organized into an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.
- The same kernel can be launched with different numbers of threads at different parts of the host code. For a given grid of threads, the number of threads in a block is available in the blockDim variable.
- Figure 3.10 shows an example where each block consists of 256 threads. The number of threads in each thread block is specified by the host code when a kernel is launched. The value of the blockDim.x variable is 256. In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons. We will revisit this later.

# CUDA Square C code Example

# Declare and initialize CPU arrays

cuda-square.cu

```
10  int
11  main(int argc, char **argv) {
12    const int ARRAY_SIZE = 64;
13    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
14
15    // Declare and initialize CPU arrays.
16    float h_in[ARRAY_SIZE];
17    float h_out[ARRAY_SIZE];
18    for (int i = 0;  i < ARRAY_SIZE;  i++) {
19      h_in[i] = float(i);
20    }
```

# Declare and Allocate GPU memory

cuda-square.cu

```
22 // Declare and allocate GPU memory.
23 float *d_in;
24 float *d_out;
25 cudaMalloc((void **)&d_in, ARRAY_BYTES);
26 cudaMalloc((void **)&d_out, ARRAY_BYTES);
```

# Specify Kernel code in your program

cuda-square.cu

```
3  __global__ void
4  square(float *d_out, float *d_in) {
5    int idx = threadIdx.x;
6    float f = d_in[idx];
7    d_out[idx] = f * f;
8  }
```

**Double Underscore or Dunder**

# Execute Kernel code on GPU

Copy Over - Compute - Copy Back

cuda-square.cu

```
28  // Copy array to GPU.
29  cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
30
31  // Launch the kernel.
32  square<<<1, ARRAY_SIZE>>> (d_out, d_in);
33
34  // Copy results back from GPU.
35  cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

Clean Up

cuda-square.cu

```
43  // Release GPU memory.
44  cudaFree(d_in);
45  cudaFree(d_out);
```

Different GPU kernels can be coded and called from your code