GPU Programming

Week # 14

CPU

Single Core CPUs

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

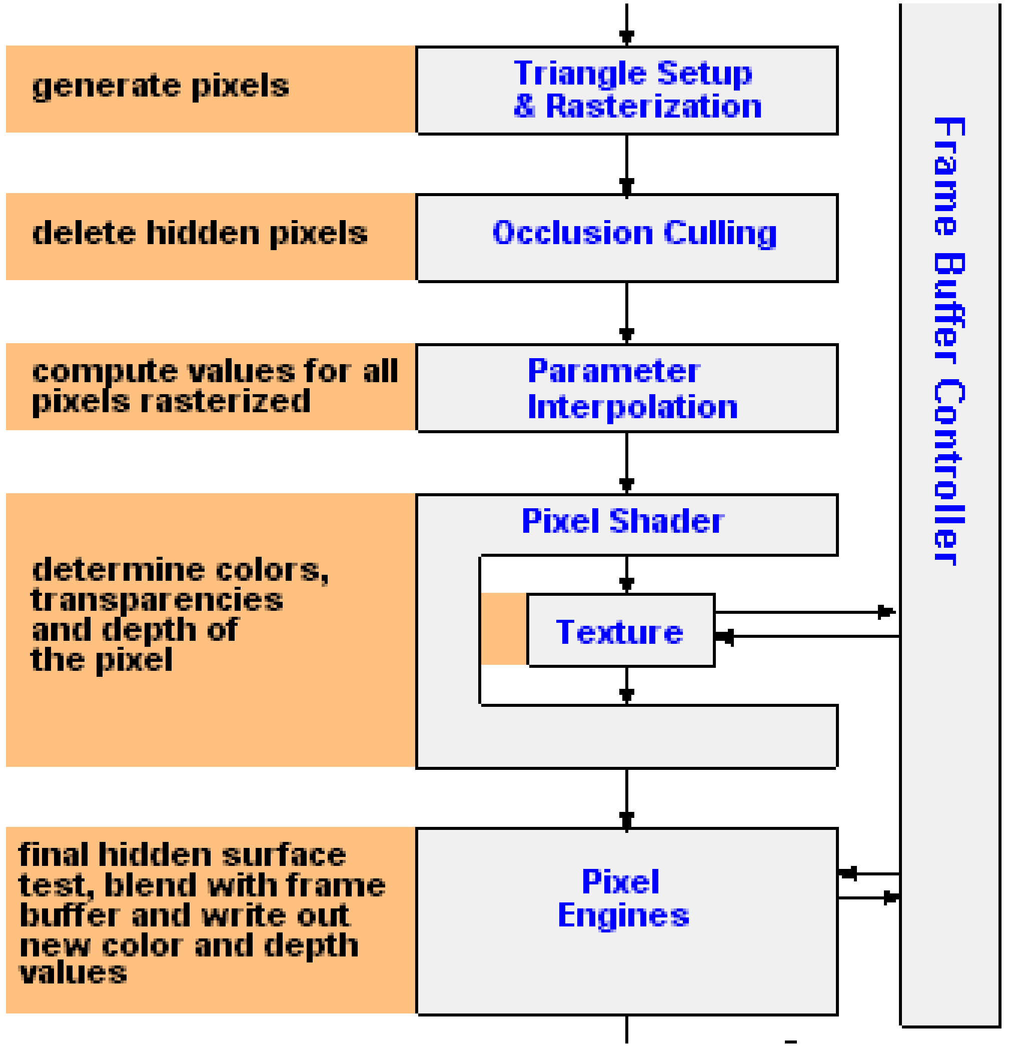
"Power Wall + Memory Wall + ILP Wall = Brick Wall"

Multi-Cores CPUs

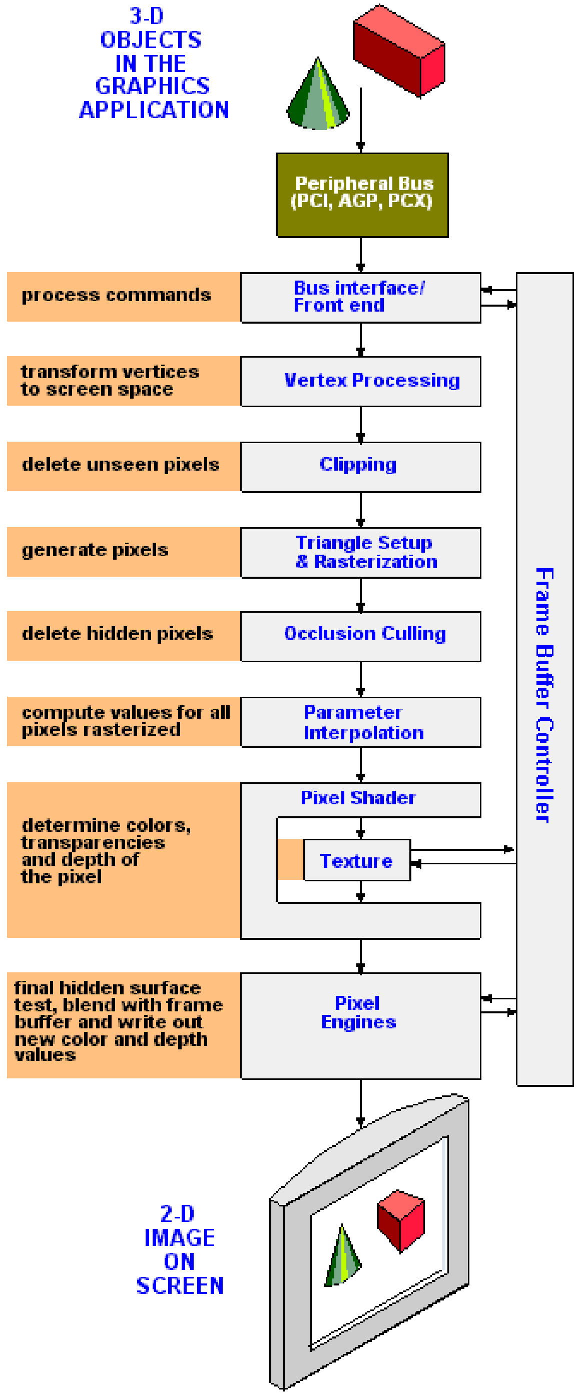
- Instead of adding complexity to a single core we scale cores in the same dimensions of the silicon
- Current Multi-Cores (Discuss)
- Do we need more?
  - Vector Processing Units (VPUs),
  - Graphic Processing Units (GPUs),
  - Associative Processing Units (APUs),
  - Tensor Processing Units (TPUs)
  - Field Programmable Gate Arrays (FPGAs),
  - Quantum Processing Units (QPUs)

Graphic Processing

Not part of syllabus



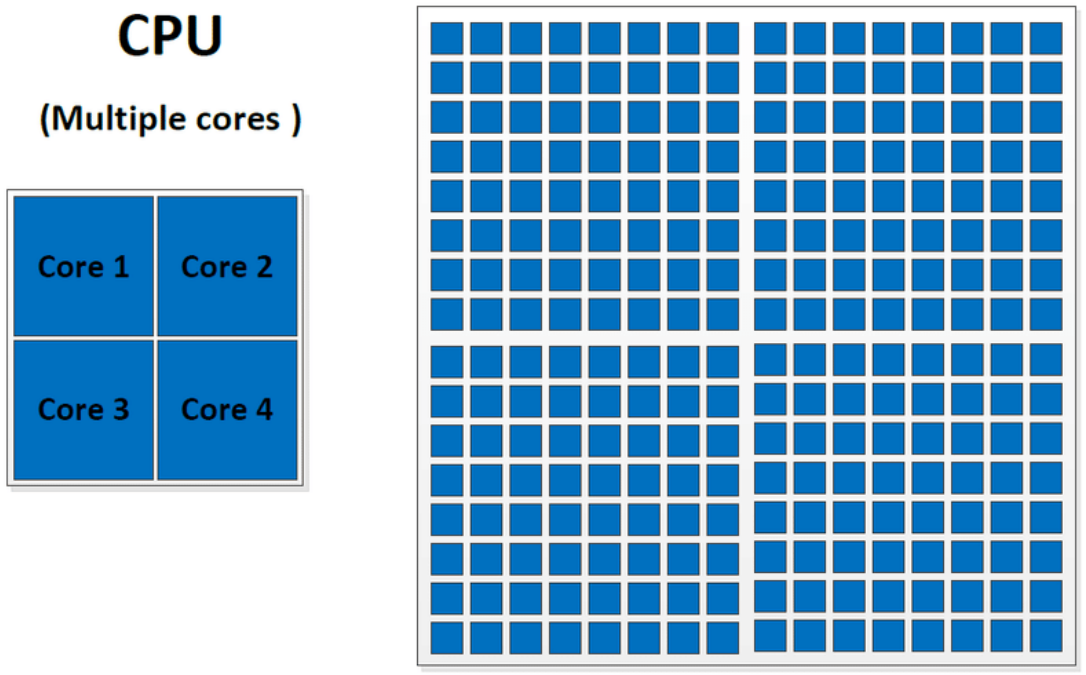
generate pixels
Triangle Setup & Rasterization
delete hidden pixels
Occlusion Culling
compute values for all pixels rasterized
Parameter Interpolation
determine colors, transparencies and depth of the pixel
Pixel Shader
Texture
final hidden surface test, blend with frame buffer and write out new color and depth values
Pixel Engines
Frame Buffer Controller



3-D OBJECTS IN THE GRAPHICS APPLICATION
Peripheral Bus (PCI, AGP, PCX)
process commands
Bus interface/ Front end
transform vertices to screen space
Vertex Processing
delete unseen pixels
Clipping
generate pixels
Triangle Setup & Rasterization
delete hidden pixels
Occlusion Culling
compute values for all pixels rasterized
Parameter Interpolation
determine colors, transparencies and depth of the pixel
Pixel Shader
Texture
final hidden surface test, blend with frame buffer and write out new color and depth values
Pixel Engines
Frame Buffer Controller
2-D IMAGE ON SCREEN

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)

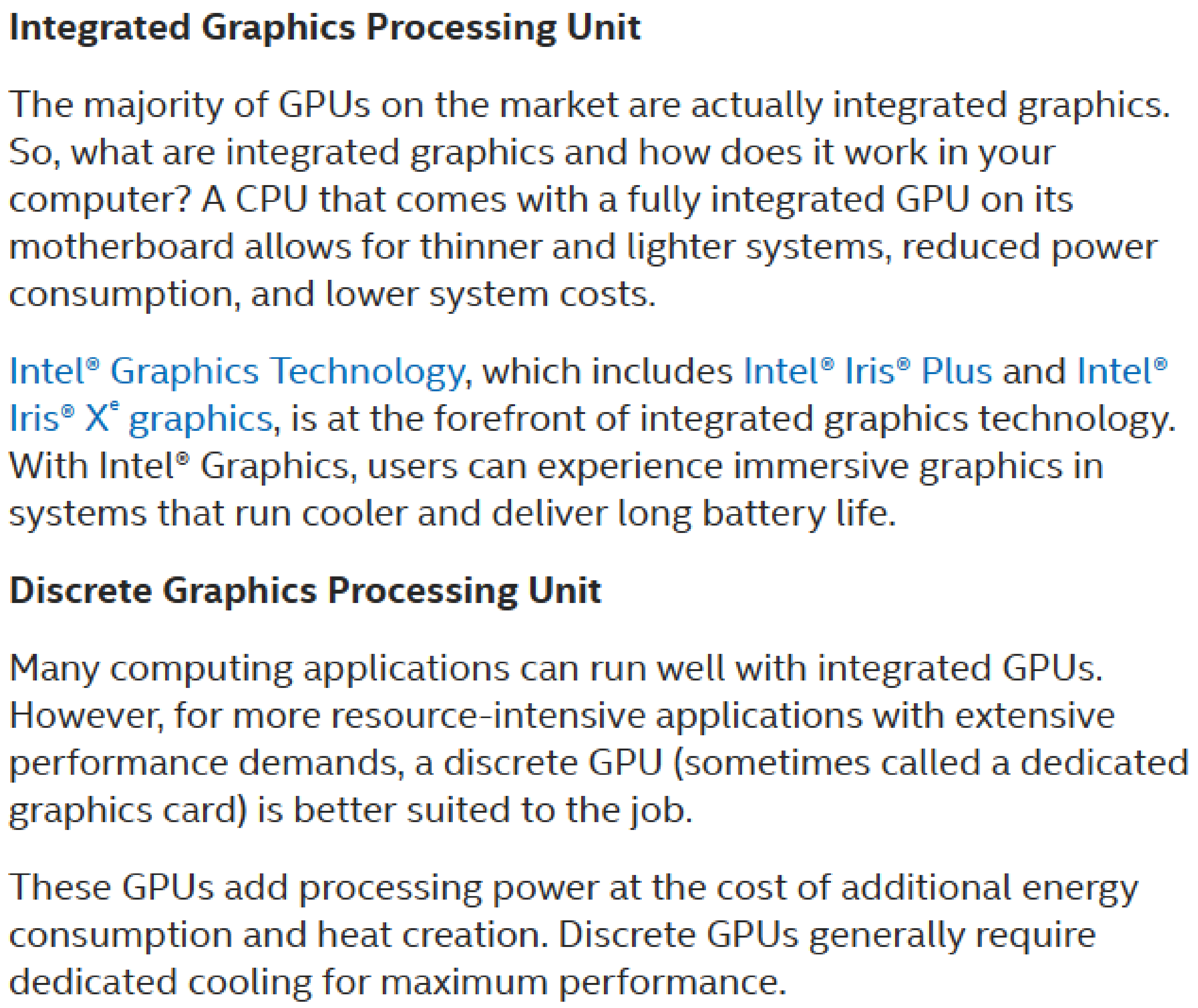
Your goals are not aligned with post office goals

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**



Integrated Graphics Processing Unit

The majority of GPUs on the market are actually integrated graphics. So, what are integrated graphics and how does it work in your computer? A CPU that comes with a fully integrated GPU on its motherboard allows for thinner and lighter systems, reduced power consumption, and lower system costs.

Intel® Graphics Technology, which includes Intel® Iris® Plus and Intel® Iris® X^e graphics, is at the forefront of integrated graphics technology. With Intel® Graphics, users can experience immersive graphics in systems that run cooler and deliver long battery life.

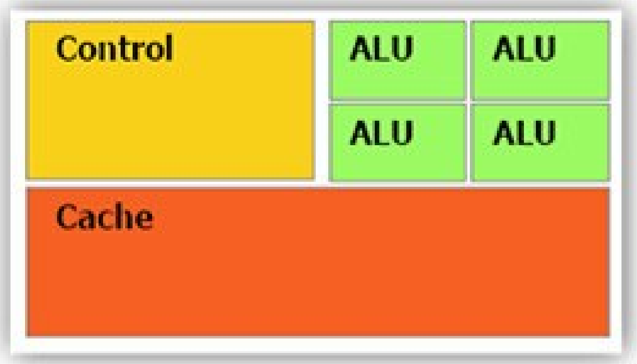
Discrete Graphics Processing Unit

Many computing applications can run well with integrated GPUs. However, for more resource-intensive applications with extensive performance demands, a discrete GPU (sometimes called a dedicated graphics card) is better suited to the job.

These GPUs add processing power at the cost of additional energy consumption and heat creation. Discrete GPUs generally require dedicated cooling for maximum performance.

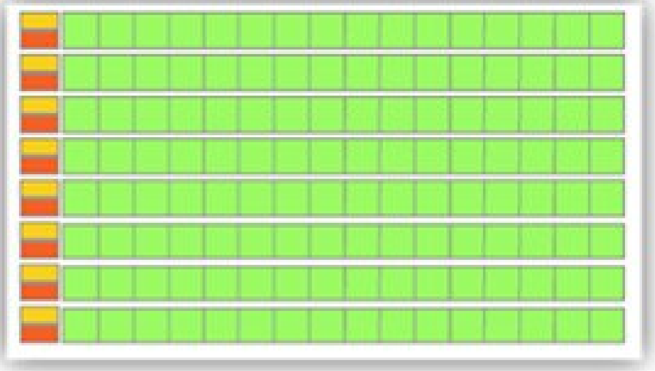
https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html

CPU



- Low compute density
- Complex control logic
- Large caches (L1$/L2$, etc.)
- Optimized for serial operations
  - Fewer execution units (ALUs)
  - Higher clock speeds
- Shallow pipelines (<30 stages)
- Low Latency Tolerance
- Newer CPUs have more parallelism

GPU



- High compute density
- High Computations per Memory Access
- Built for parallel operations
  - Many parallel execution units (ALUs)
  - Graphics is the best known case of parallelism
- Deep pipelines (hundreds of stages)
- High Throughput
- High Latency Tolerance
- Newer GPUs:
  - Better flow control logic (becoming more CPU-like)
  - Scatter/Gather Memory Access
  - Don't have one-way pipelines anymore

How GPU increases throughput?

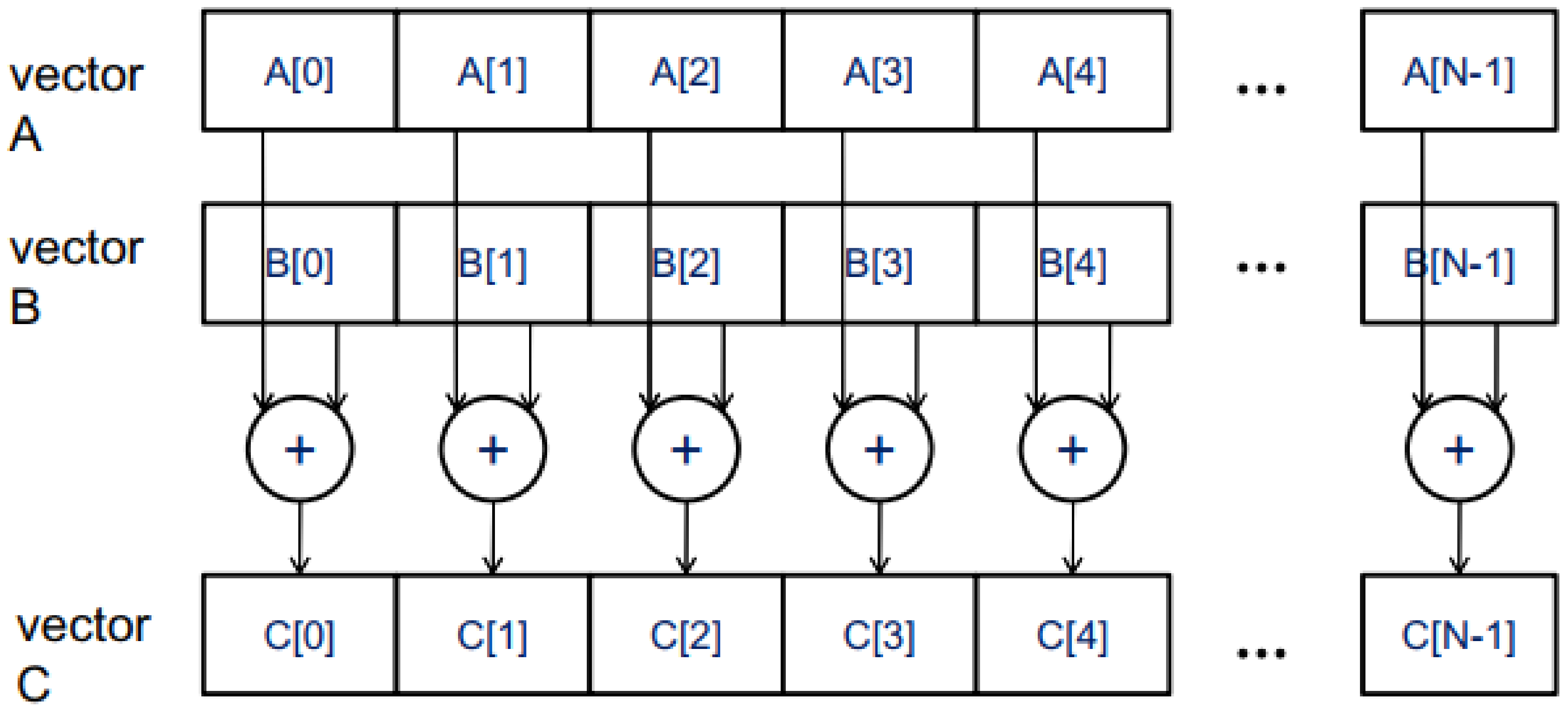
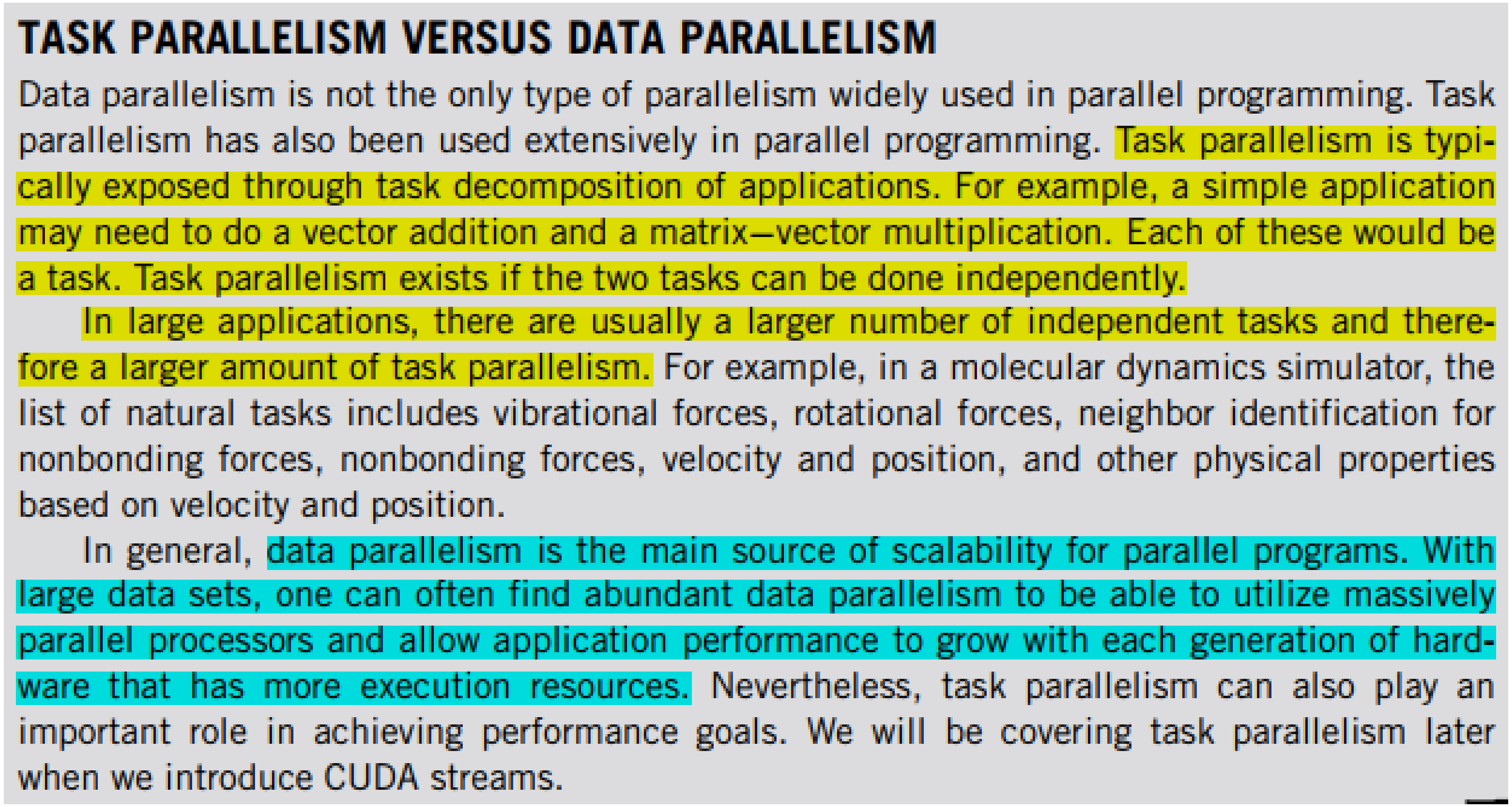


FIGURE 3.1

Data parallelism in vector addition.



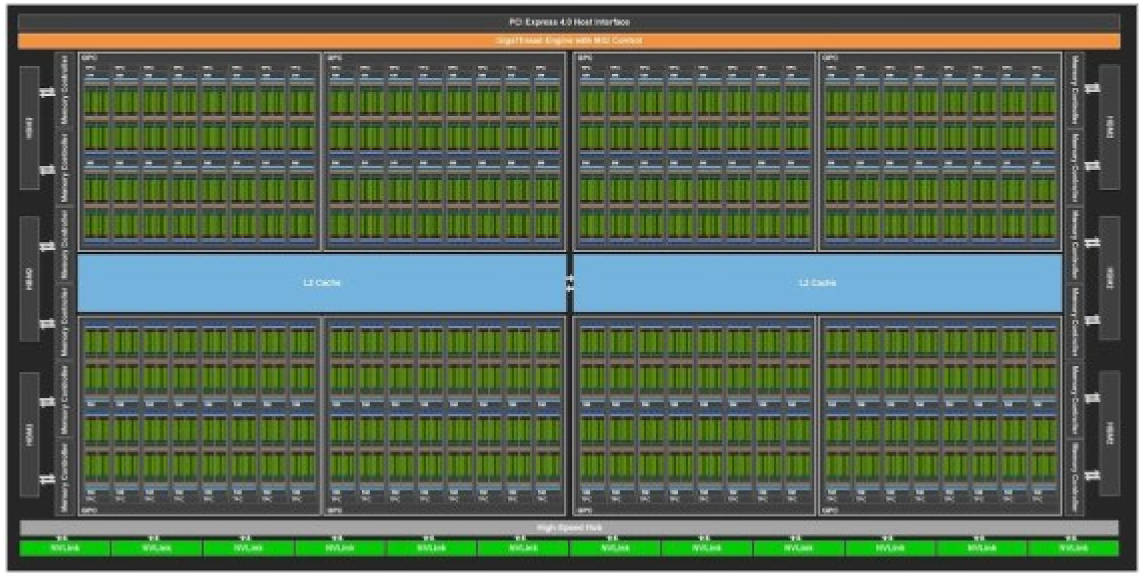
TASK PARALLELISM VERSUS DATA PARALLELISM

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.

NVIDIA Tesla A100 with 54 Billion Transistors


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
L2 Cache
L2 Cache
HBM2
Memory Controller
High-Speed Hub
NVLink

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

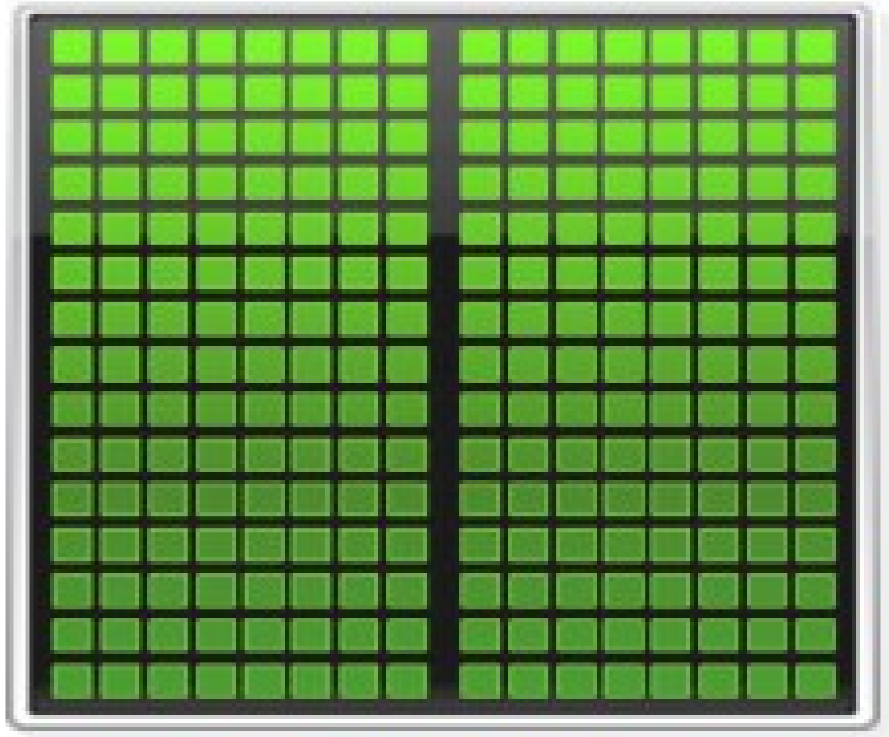
CPU

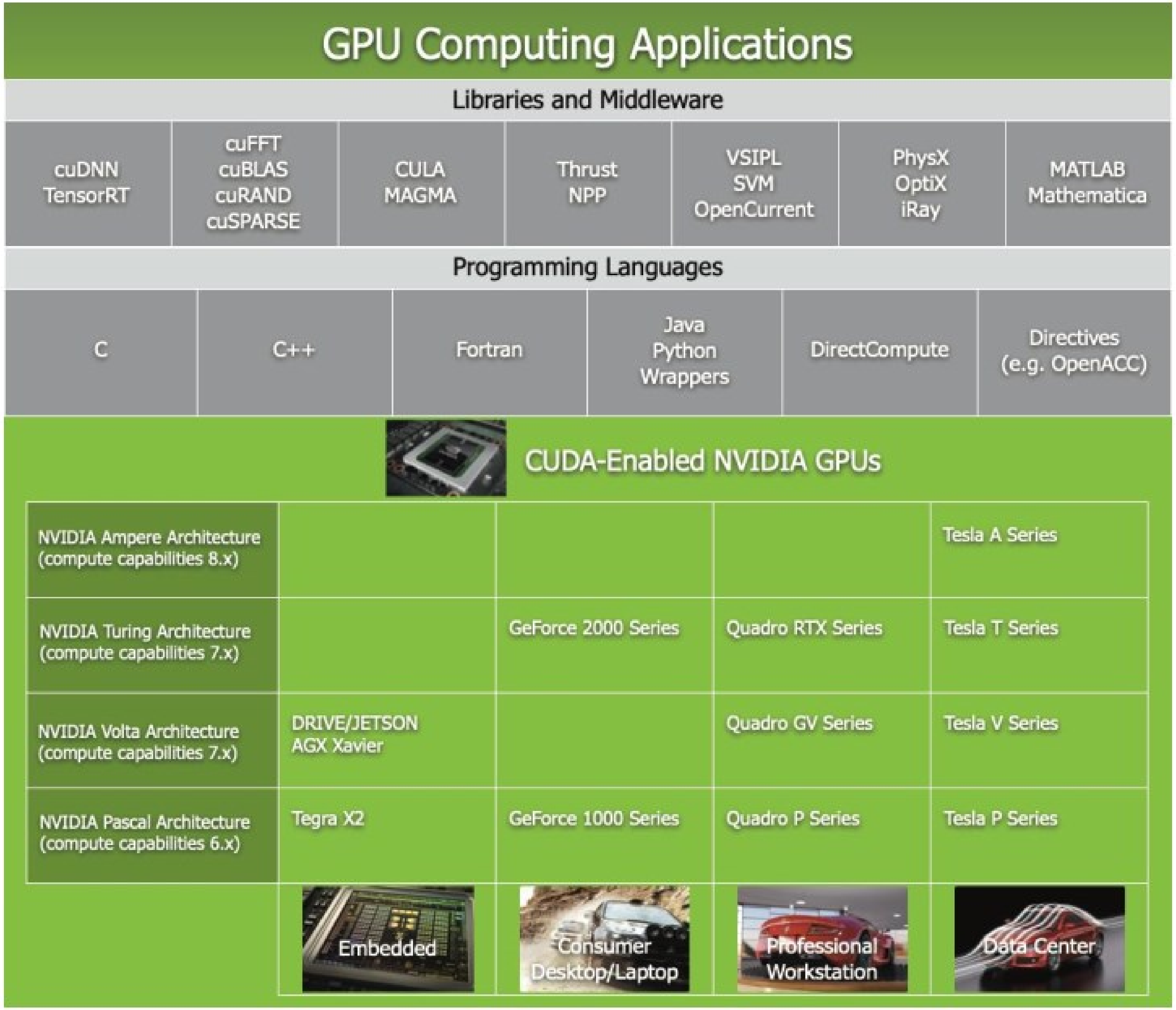
Optimized for Serial Tasks

+

GPU Accelerator

Optimized for Many Parallel Tasks





GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

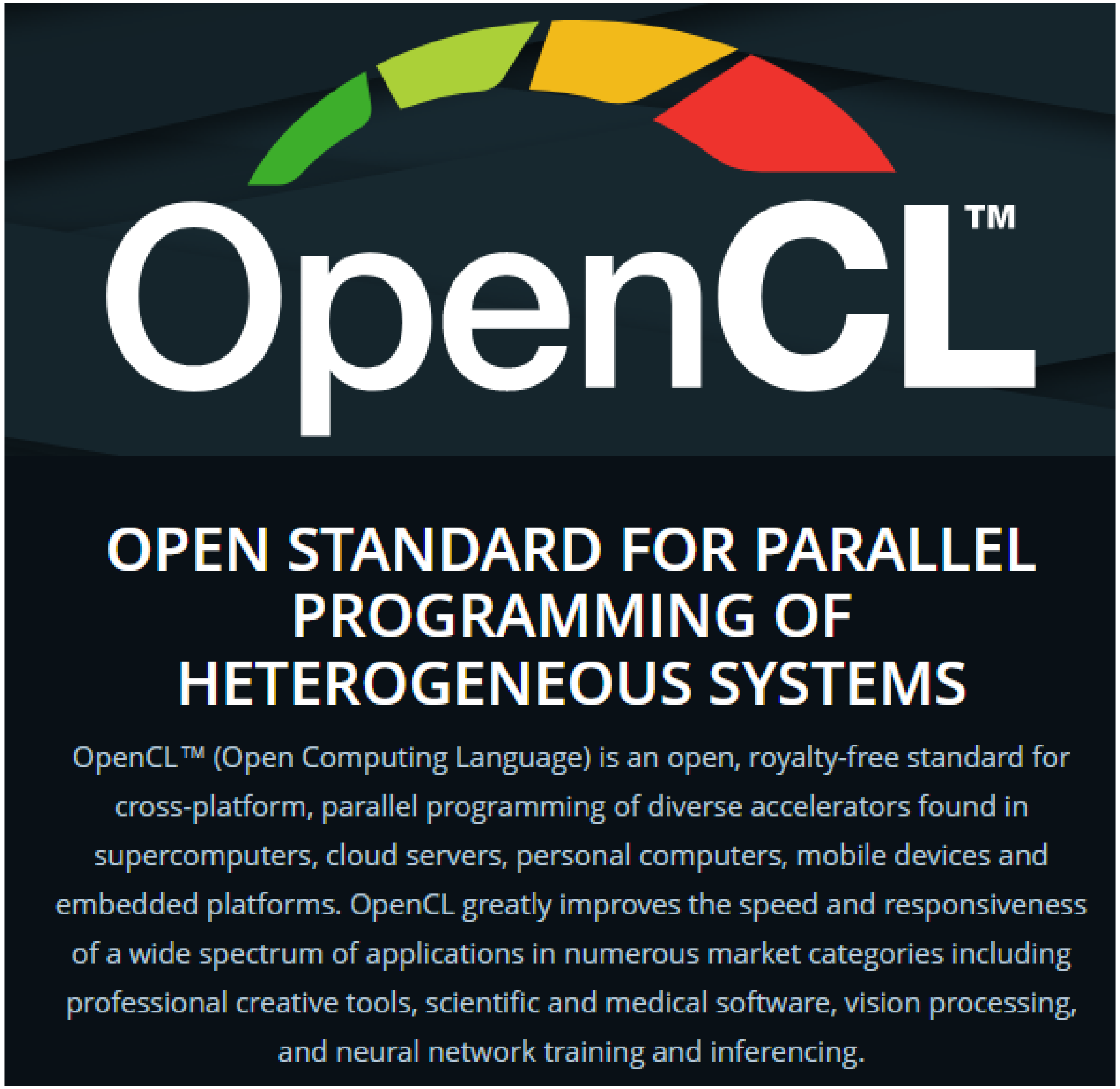
CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

- CUDA Platform for NVIDIA GPUs
- OpenCL – An Open platform for Compute Accelerators
- OpenCL vs CUDA
- How GPU accelerate Compute?
- A typical CUDA program
  - Write CPU code in C/C++ normally
  - Declare and allocate GPU memory
  - Specify Kernel code in your program – It will run on GPU
  - Execute Kernel code on GPU
  - Different Kernels can exists in your program and activated in your code

NVIDIA CUDA platform

- CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).
- With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.
- The CUDA Toolkit from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.



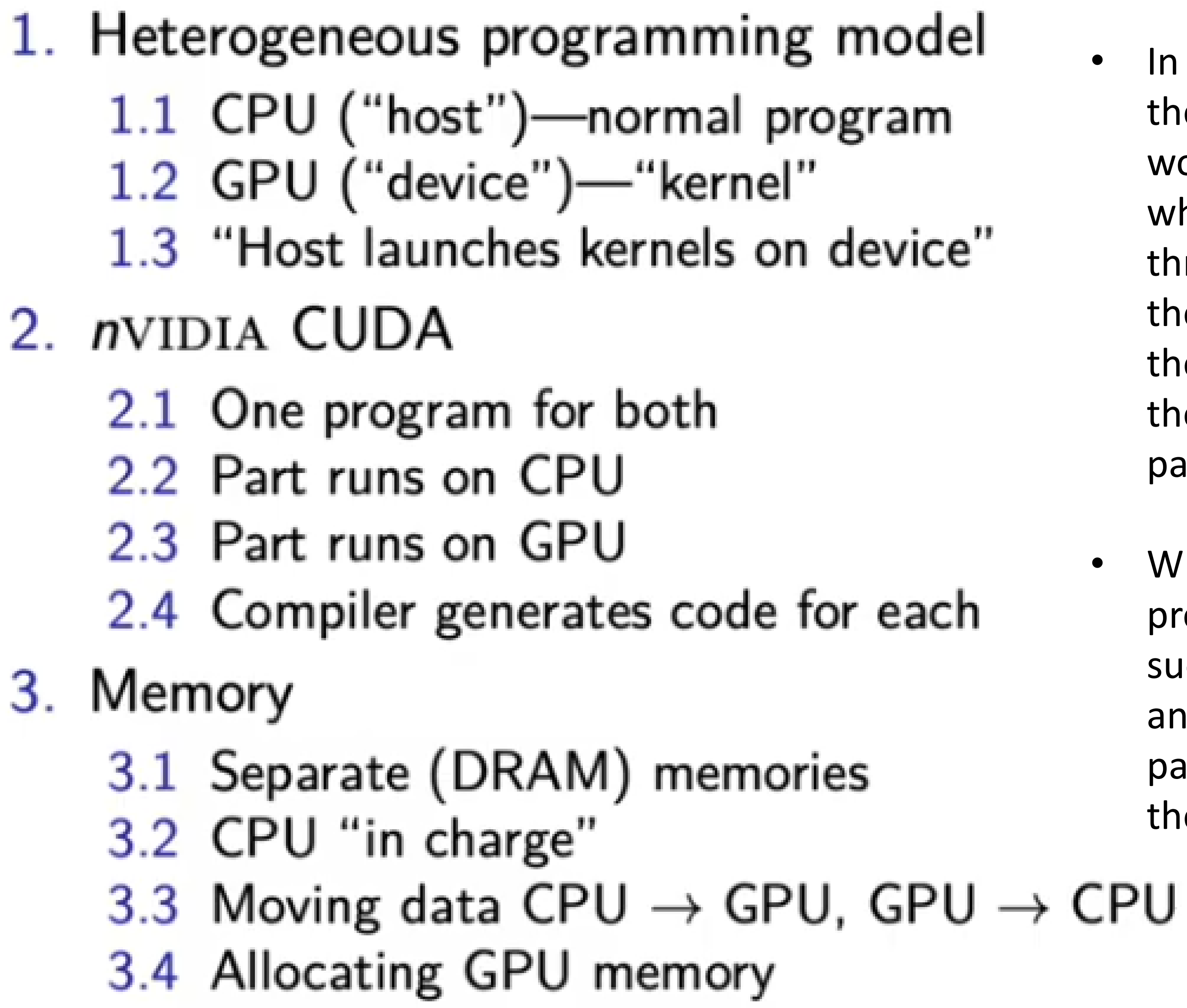
OPENCL™

OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS

OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including professional creative tools, scientific and medical software, vision processing, and neural network training and inferencing.

OpenCL vs CUDA

- You can write OpenCL programs which will function on Nvidia/AMD/Intel GPUs, AMD/Intel/ARM CPUs, FPGAs etc. and excellent compatibility across Windows, Linux and MacOS.
- If you use CUDA, you will have to use Nvidia GPUs only and re-write your code again in OpenCL or other language for other platforms.
- As long as you don't plan to use Nvidia-proprietary things like tensor / ray-tracing cores, OpenCL is just as fast as CUDA when properly optimized.
- A serious limitation of using CUDA and cause of serious waste of time in the long run.



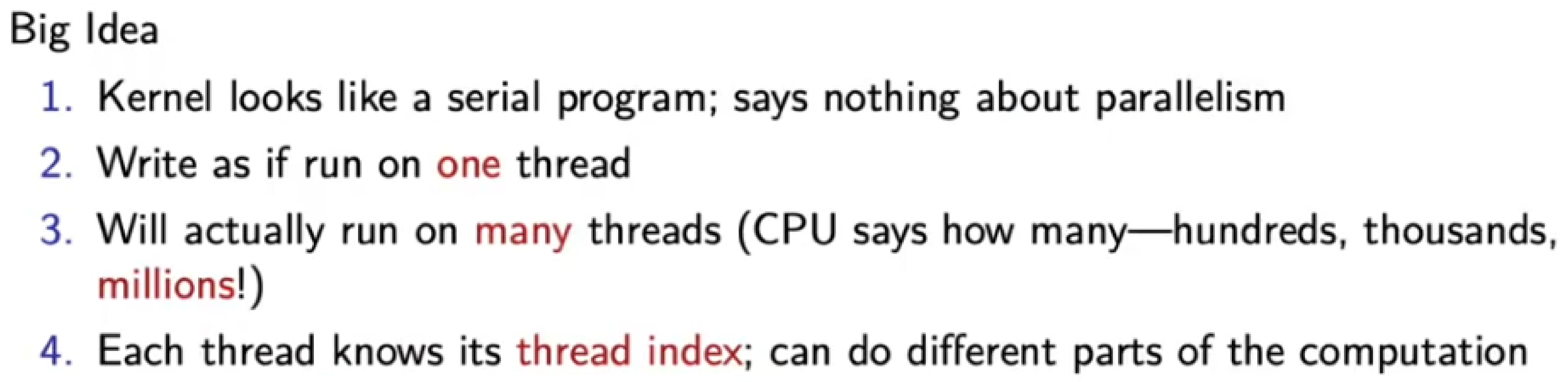
1. Heterogeneous programming model
   1.1 CPU (“host”)—normal program
   1.2 GPU (“device”)—“kernel”
   1.3 “Host launches kernels on device”
2. nVIDIA CUDA
   2.1 One program for both
   2.2 Part runs on CPU
   2.3 Part runs on GPU
   2.4 Compiler generates code for each
3. Memory
   3.1 Separate (DRAM) memories
   3.2 CPU “in charge”
   3.3 Moving data CPU $\to$ GPU, GPU $\to$ CPU
   3.4 Allocating GPU memory

- 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.

1. 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`)
2. Still have computation/communication tradeoff!
3. Can launch lots of threads efficiently (1,000+)
4. Can run lots of threads in parallel

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

cuda-square.cu

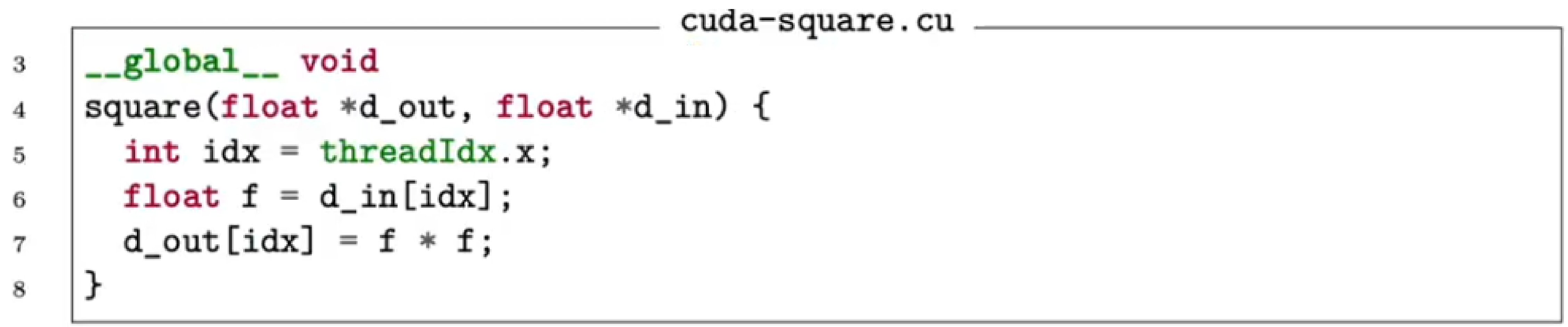
```
int
main(int argc, char **argv) {
  const int ARRAY_SIZE = 64;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // Declare and initialize CPU arrays.
  float h_in[ARRAY_SIZE];
  float h_out[ARRAY_SIZE];
  for (int i = 0;  i < ARRAY_SIZE;  i++) {
    h_in[i] = float(i);
  }
```

Declare and Allocate GPU memory

cuda-square.cu

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

Specify Kernel code in your program

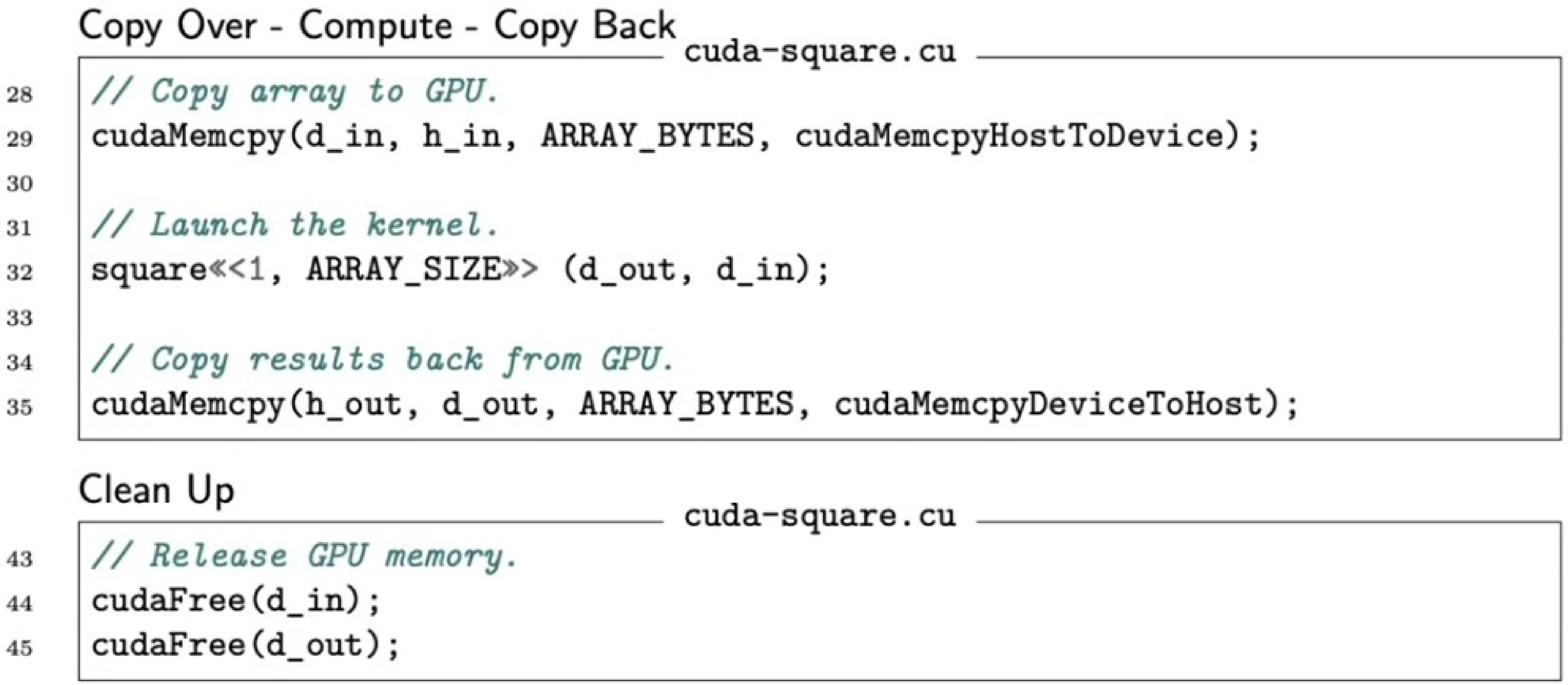


cuda-square.cu

```
__global__ void
square(float *d_out, float *d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f;
}
```

Double Underscore or Dunder

Execute Kernel code on GPU



Copy Over - Compute - Copy Back

cuda-square.cu

```
// Copy array to GPU.
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

// Launch the kernel.
square«<1, ARRAY_SIZE»> (d_out, d_in);

// Copy results back from GPU.
cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

Clean Up

cuda-square.cu

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

Different GPU kernels can be coded and called from your code cuda-square.cu

```
int
main(int argc, char **argv) {
  const int ARRAY_SIZE = 64;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // Declare and initialize CPU arrays.
  float h_in[ARRAY_SIZE];
  float h_out[ARRAY_SIZE];
  for (int i = 0;  i < ARRAY_SIZE;  i++) {
    h_in[i] = float(i);
  }
```

cuda-square.cu

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

cuda-square.cu

```
__global__ void
square(float *d_out, float *d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f;
}
```

Copy Over - Compute - Copy Back

cuda-square.cu

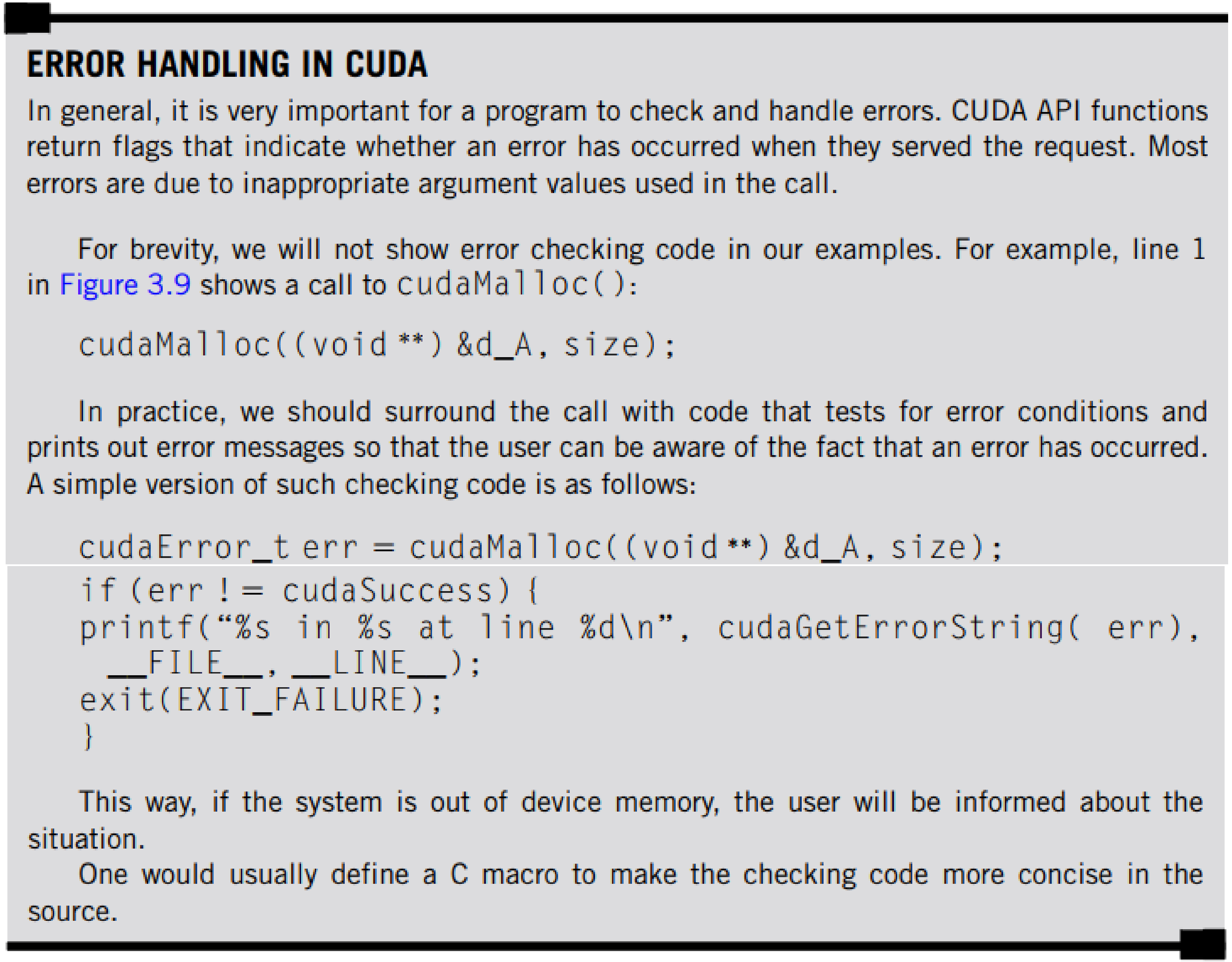
```
// Copy array to GPU.
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

// Launch the kernel.
square<<<1, ARRAY_SIZE>>> (d_out, d_in);

// Copy results back from GPU.
cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

Clean Up

cuda-square.cu

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

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



ERROR HANDLING IN CUDA

In general, it is very important for a program to check and handle errors. CUDA API functions return flags that indicate whether an error has occurred when they served the request. Most errors are due to inappropriate argument values used in the call.

For brevity, we will not show error checking code in our examples. For example, line 1 in Figure 3.9 shows a call to `cudaMalloc()`:

```
cudaMalloc((void **) &d_A, size);
```

In practice, we should surround the call with code that tests for error conditions and prints out error messages so that the user can be aware of the fact that an error has occurred. A simple version of such checking code is as follows:

```
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);
}
```

This way, if the system is out of device memory, the user will be informed about the situation.

One would usually define a C macro to make the checking code more concise in the source.

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

FIGURE 3.12

CUDA C keywords for function declaration.

Note that one can use both `__host__` and `__device__` in a function declaration. This combination tells the compilation system to generate two versions of object files for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This supports a common-use case when the same function source code can be recompiled to generate a device version. Many user library functions will likely fall into this category.

[6]Note that SPMD is not the same as SIMD (single instruction, multiple data) [Flynn1972]. In an SPMD system, the parallel processing units execute the same program on multiple parts of the data. However, these processing units do not need to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.

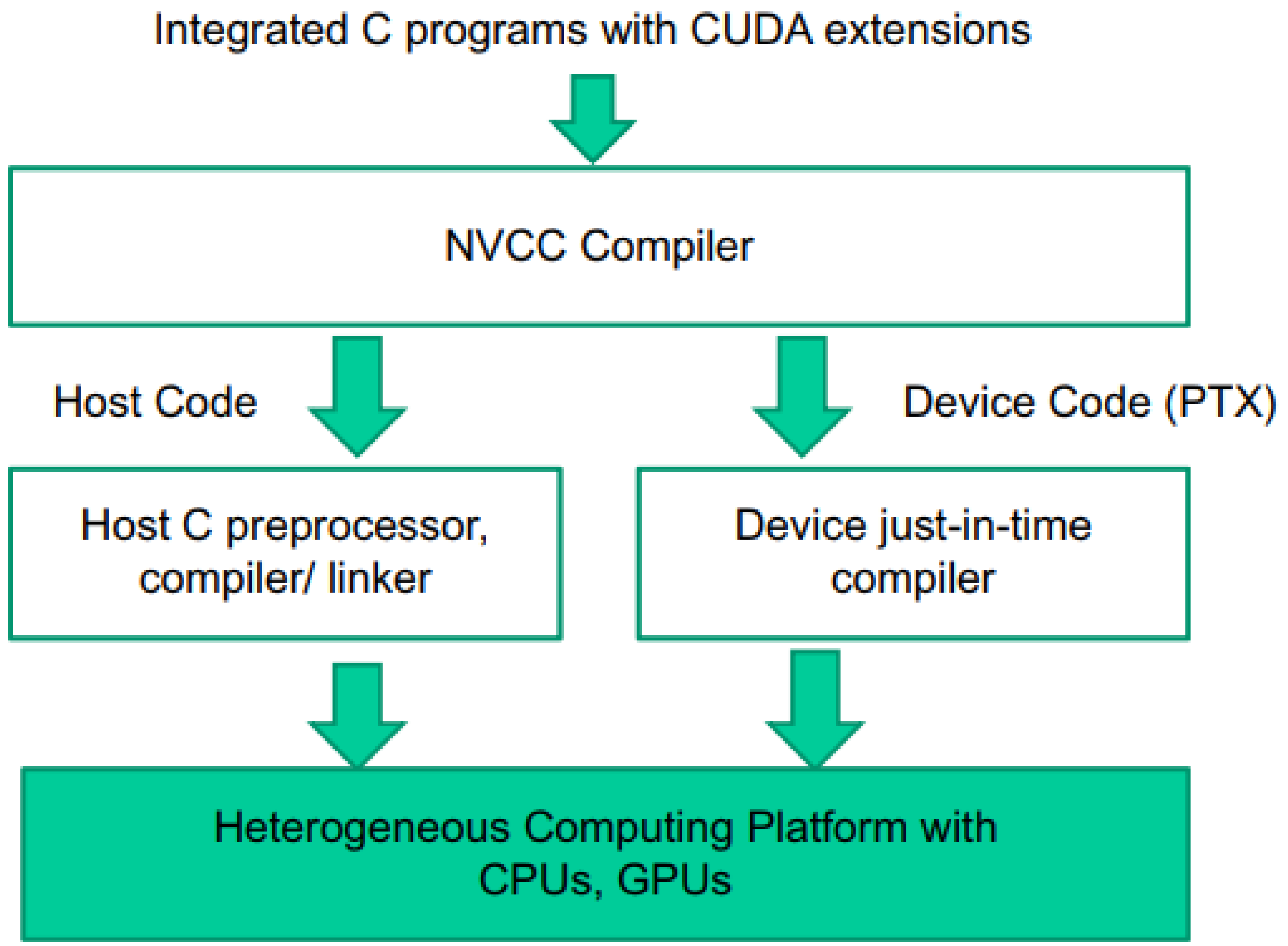

Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/ linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

FIGURE 3.2

Overview of the compilation process of a CUDA program.

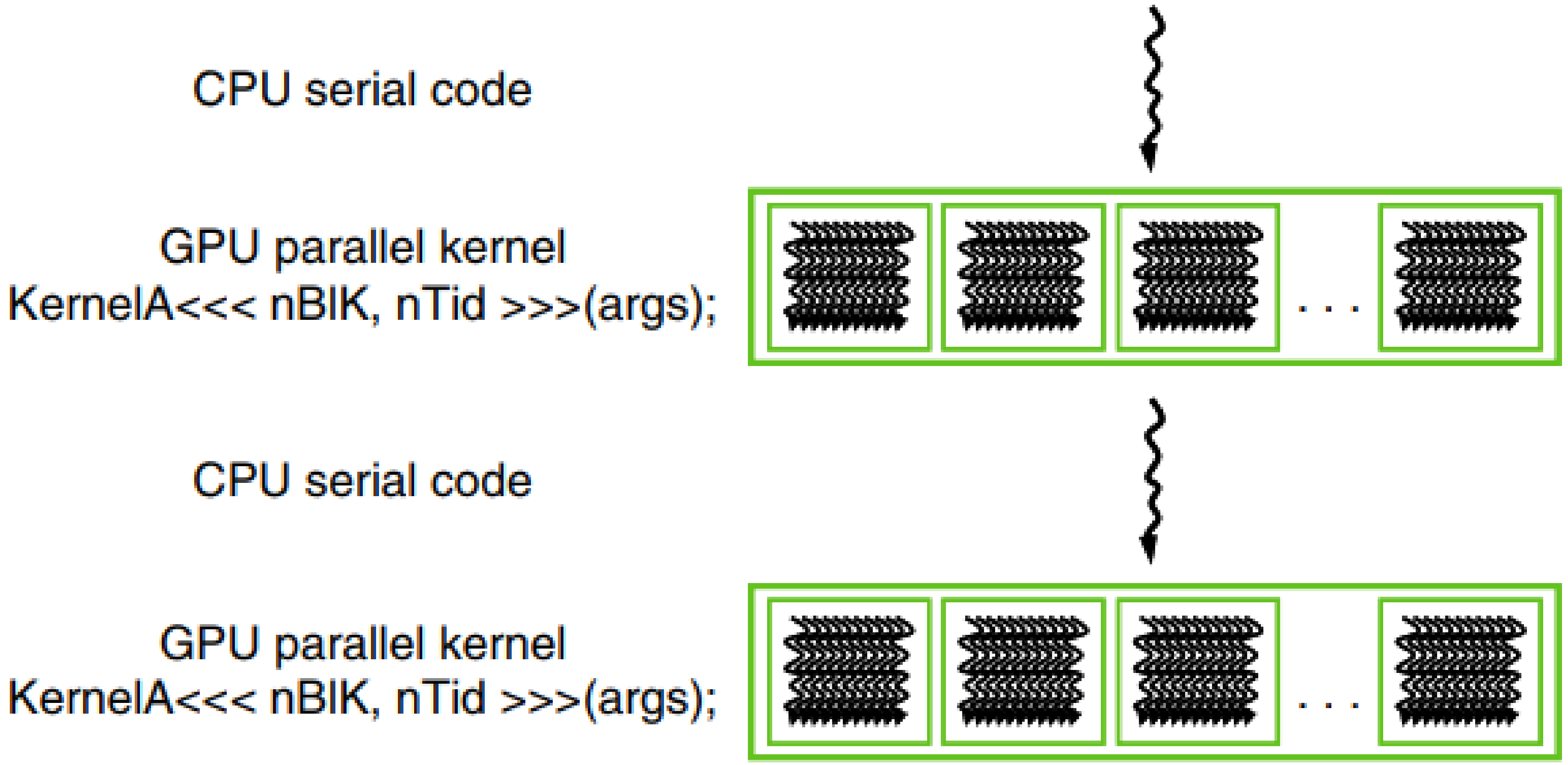


FIGURE 3.3

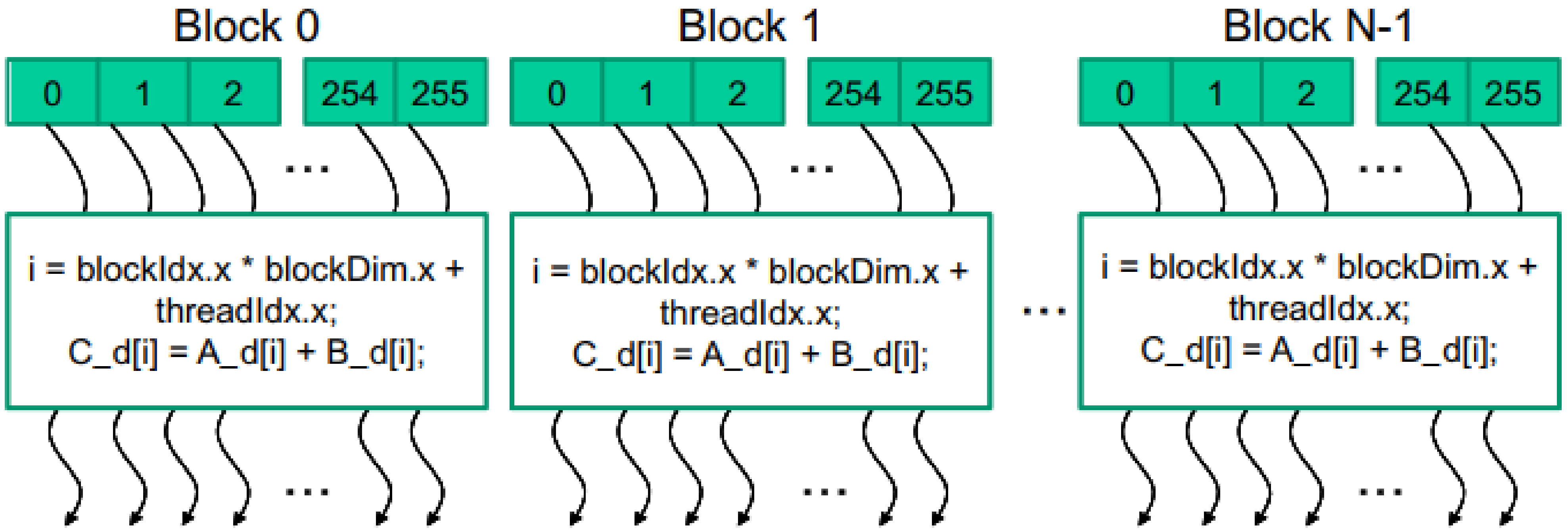
Execution of a CUDA program.

GPU Threads: Grid Block

- 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.
  - 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 same kernel can be launched with different numbers of threads at different parts of the host code.



Block 0
Block 1
Block N-1
0
1
2
254
255
i = blockIdx.x * blockDim.x + threadIdx.x;
C_d[i] = A_d[i] + B_d[i];

FIGURE 3.10

All threads in a grid execute the same kernel code.

The second notable extension to ANSI C in Figure 3.10 is the keywords `threadIdx.x`. `blockIdx.x`, and `blockDim.x`.

There needs to be a way for them to distinguish among themselves and direct each thread toward a particular part of the data. These keywords identify predefined variables that correspond to hardware registers that provide the identifying coordinates to threads. Different threads will see different values in their `threadIdx.x`, `blockIdx.x`, and `blockDim.x` variables. For simplicity, we will refer to a thread as $\texttt{thread}_{\texttt{blockIdx.x, threadIdx.x}}$. Note that the `.x` implies that there might be `.y` and `.z`.

There is an automatic (local) variable `i` in Figure 3.11. In a CUDA kernel function, automatic variables are private to each thread. That is, a version of `i` will be generated for every thread. If the kernel is launched with 10,000 threads, there will be 10,000 versions of `i`, one for each thread. The value assigned by a thread to its `i` variable is not visible to other threads.

GPU Threads: Grid Block

- For a given grid of threads, the number of threads in a block is available in the blockDim variable. In Figure 3.10, 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.

- Each thread in a block has a unique threadIdx value. For example, the first thread in block 0 has value 0 in its threadId.x variable, the second thread has value 1, the third thread has value 2, etc.

- This allows each thread to combine its threadIdx and blockIdx values to create a unique global index for itself with the entire grid. In Figure 3.10, a data index i is calculated as i = blockIdx.x * blockDim.x + threadIdx.x.

GPU Threads: Grid Block

- Since blockDim is 256 in our example,
  - the i values of threads in block 0 ranges from 0 to 255.
  - The i values of threads in block 1 range from 256 to 511.
  - The i values of threads in block 2 range from 512 to 767.
- That is, the i values of the threads in these three blocks form a continuous coverage of the values from 0 to 767.
- **Since each thread uses i to access d_A, d_B, and d_C, these threads cover the first 768 iterations of the original loop.**
- By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with n or more threads, one can process vectors of length n.

CUDA Vector Addition Code

```
// 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.

```
#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
}
```

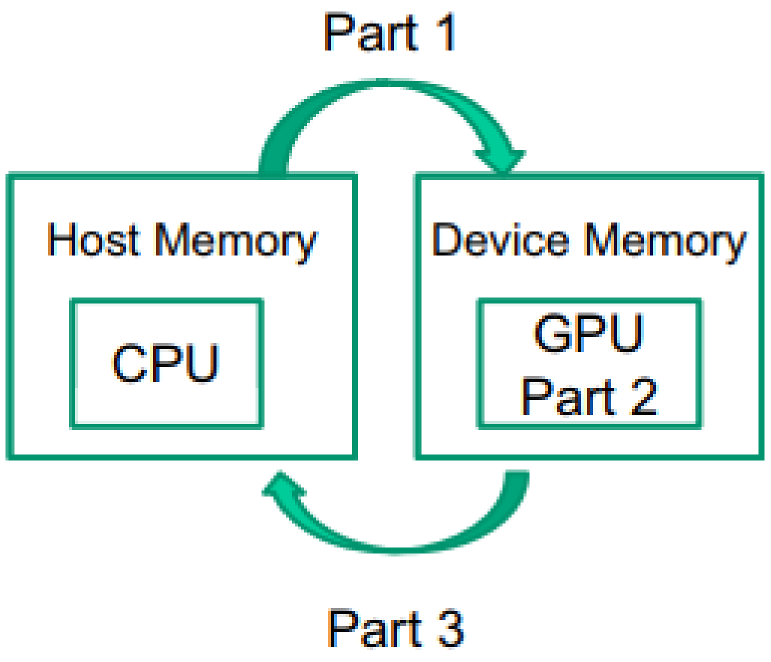


FIGURE 3.5

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

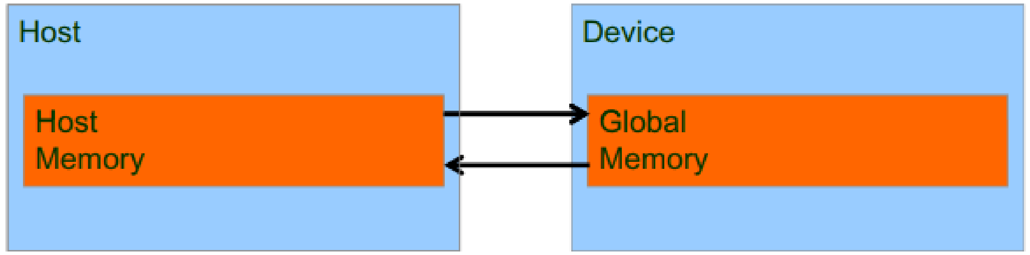


FIGURE 3.6

Host memory and device global memory.

```
void vecAdd(float* A, float* B, float* C, int n)
{
   int size = n * sizeof(float);
   float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
      // Free device memory for A, B, C
     cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 3.14

A complete version of `vecAdd()`.

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__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];
}
```

FIGURE 3.11

A vector addition kernel function and its launch statement.

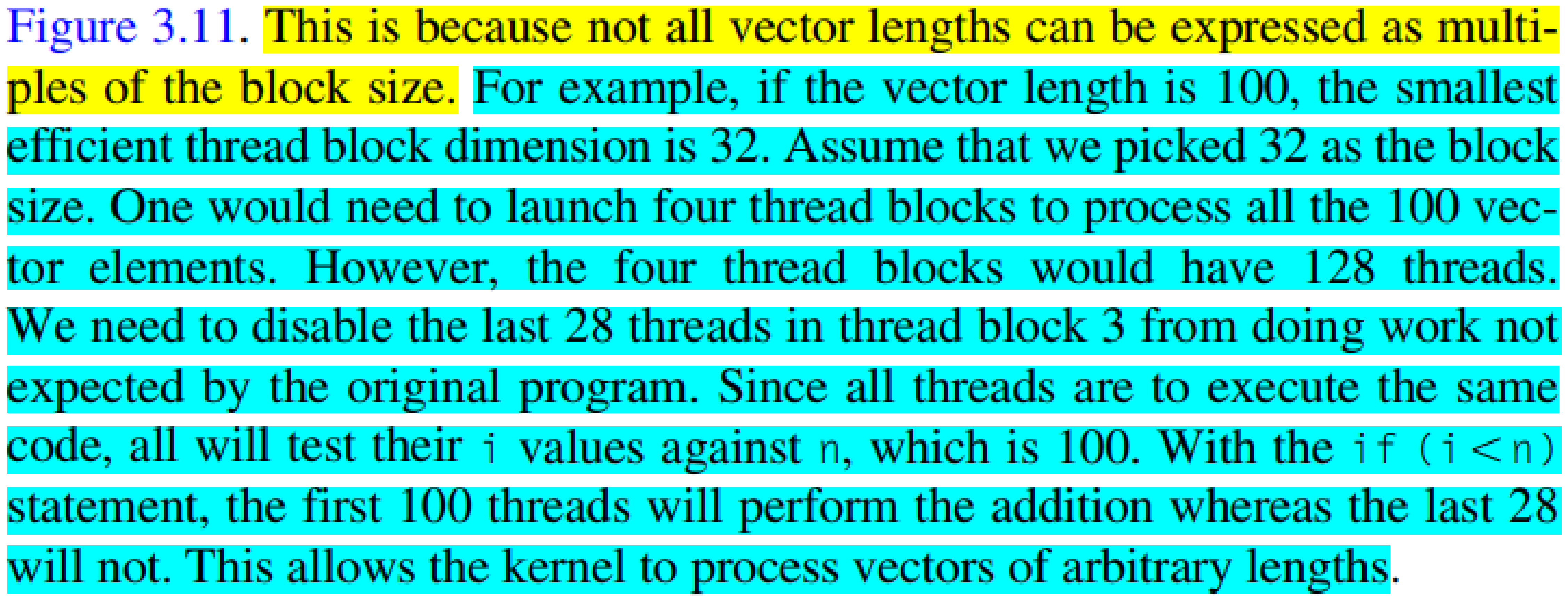


Figure 3.11. This is because not all vector lengths can be expressed as multiples of the block size. For example, if the vector length is 100, the smallest efficient thread block dimension is 32. Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their `i` values against `n`, which is 100. With the `if (i<n)` statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

Tasks suitable for GPUs – Throughput vs Latency

- It is important to point out that the vector addition example is used for its simplicity.

- In practice, the overhead of:
  - allocating device memory,
  - Input data transfer from host to device,
  - output data transfer from device to host, and
  - de-allocating device memory

  will likely make the resulting code slower than sequential code.

> - Kernels in real applications do much more work relative to the amount of data processed, which makes the additional overhead worthwhile.
> - They also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized.

- Why? … because kernel is doing small computation relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand.

Kernel executed as a Grid of Blocks of Threads

```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

No. of thread blocks

No. of thread per blocks

- To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly.

- For example, if we have 1,000 threads, we would launch ceil(1,000/256.0) 5 4 thread blocks. As a result, the statement will launch 4 3 256 5 1,024 threads.

```
vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
```

- n = 750 -> blocks = 3, n = 4,000 -> blocks = 16, n = 2,000,000 -> block = 7,813.
- Note that all the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order.
- A small GPU with limited execution resources may execute one or two of these thread blocks in parallel. A larger GPU may execute 64 or 128 blocks in parallel.
- This gives CUDA kernels scalability in execution speed with hardware. i.e. same code runs at lower performance on small GPUs and higher performance on larger GPUs.