Notes on CUDA Programming

Aashay Kulkarni
December, 2025

Contents

1. Introduction

Multicore CPUs enable you to maximize execution speed of sequential programs, while many-core GPUs allow greater execution throughput of parallel applications.

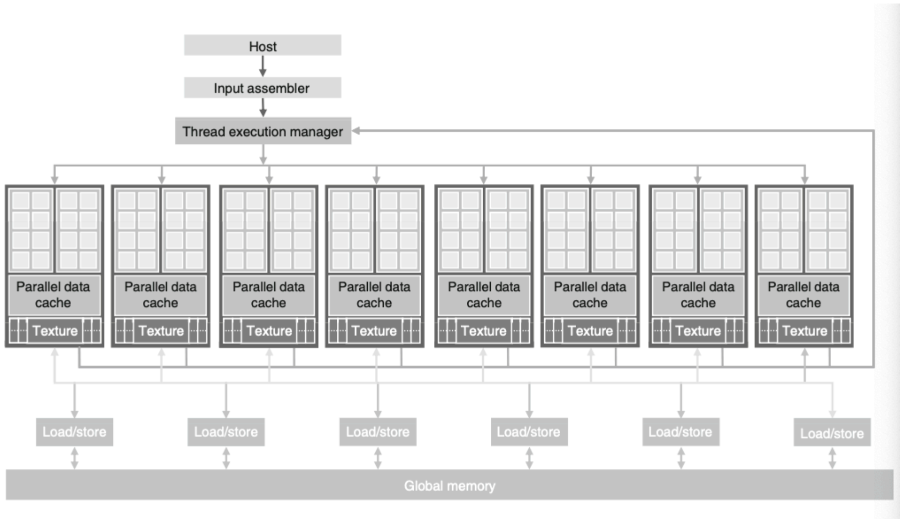
Today, there is a **large** performance gap between parallel and sequential execution.

Why?

- design of a CPU is optimized for sequential code performance
- memory bandwidth is another issue. GPUs move data much faster in and out of its DRAM.
- developers move computationally intensive parts of software to GPUs
- video games require massive number of floating point calculations per video frame, being executed in parallel and GPUs have been used for this purpose.

1.1. CUDA (Compute Unified Device Architecture)

- programming model created by NVIDIA to support joint CPU/GPU execution of an application.
- CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs). SMs combine to form a building block. SMs have a number of streaming processors (SPs) that share control logic and instruction cache.



- To experience speedup offered by parallelization, a large part of the application's execution time must be in the parallel portion.
- Certain applications have portions better suited to CPUs and hence a combined CPU/GPU parallel computing capability is required. This is precisely what CUDA promotes.
- Key steps in parallel computing:
  - identifying parts of application programs to be parallelized
  - isolating the data to be used by the parallelizing code by using API functions to allocate memory on the parallel computing device
  - using API functions to transfer data to the parallel computing device
  - developing kernel functions that will be executed by individual threads in the parallel part
  - launching kernel functions for execution by parallel parts
  - transferring data back to the host processor with API function calls

2. History of GPU Computing

2.1. GPGPU

- General Purpose Computing on GPUs.
- GPU processor array and frame buffer memory were designed to process graphics data and were too restrictive for general numerical applications.
- writes were extremely difficult -> could only be emitted as a pixel color value and configure the frame buffer operation to write.
- the handful of useful applications created with general computations on a GPU -> this field was called GPGPU.

2.2. GPU Computing

- NVIDIA developed Tesla GPU Architecture.
  - programming paradigm to think of GPU like a processor.
- programming approach involved explicit declaration of data-parallel aspects of their workload.
- no longer need to use graphics API to access parallel computing capabilities.

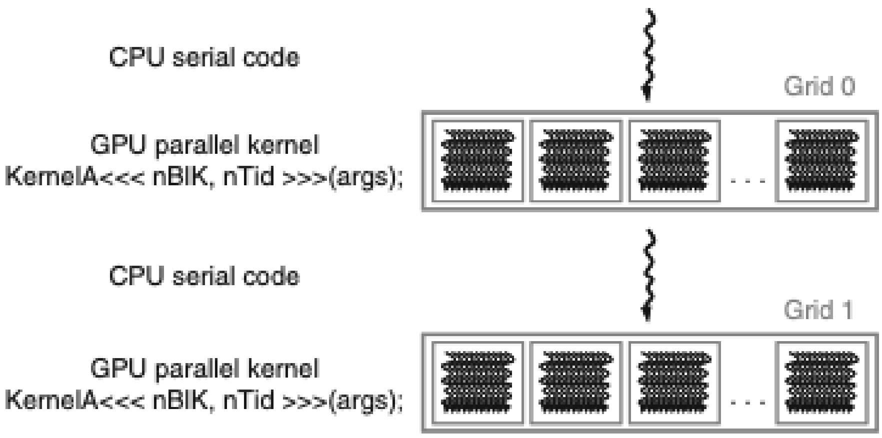
3. Introduction to CUDA

3.1. Data Parallelism

- computing system consists of host (CPU) & devices (massively parallel processors)
- CUDA devices accelerate execution of applications by harvesting a large amount of data parallelism.
- Matrix multiplication $P = M \times N$:
  - as every entry p_{ij} is independent of each other, a large amount of data parallelism can be performed.

3.2. CUDA Program Structure

- CUDA program comprises phases that are executed either by the host (CPU) or a device such as a GPU.
- CUDA program is a unified source code comprising both host & device code.
  - NVIDIA C compiler (nvcc): host code = ANSI C; device code = ANSI C extended with keywords for data-parallel functions, called kernels.
  - kernel functions generate a large number of threads to exploit data parallelism.
  - when no device is available, one can execute the kernel on a CPU using the CUDA SDK or MCUDA tool.
- CUDA threads are faster to generate and schedule than CPU threads due to efficient hardware support.



- CUDA program:
  - starts with CPU execution; when a kernel function is invoked, execution is moved to the device.
  - large numbers of threads are generated to take advantage of data parallelism, collectively called **grids**.
  - when all threads finish execution, the corresponding grid is terminated and execution moves back to the host.

3.2.1. MATMUL Example

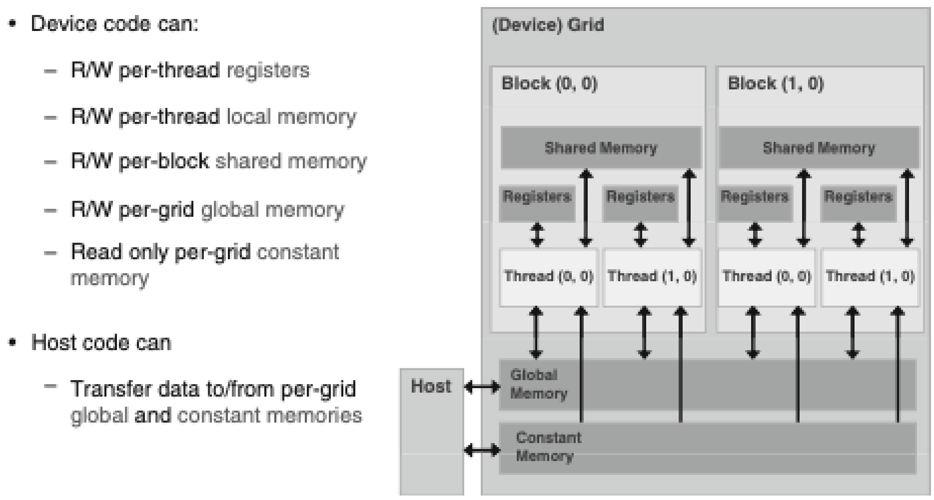
```
int main (void) {
    // 1. Allocate and initialize matrices M, N, P
    //    I/O to read input matrices M and N

    // 2. M * N on the device
    //    MatrixMultiplication(M, N, P, Width);
```

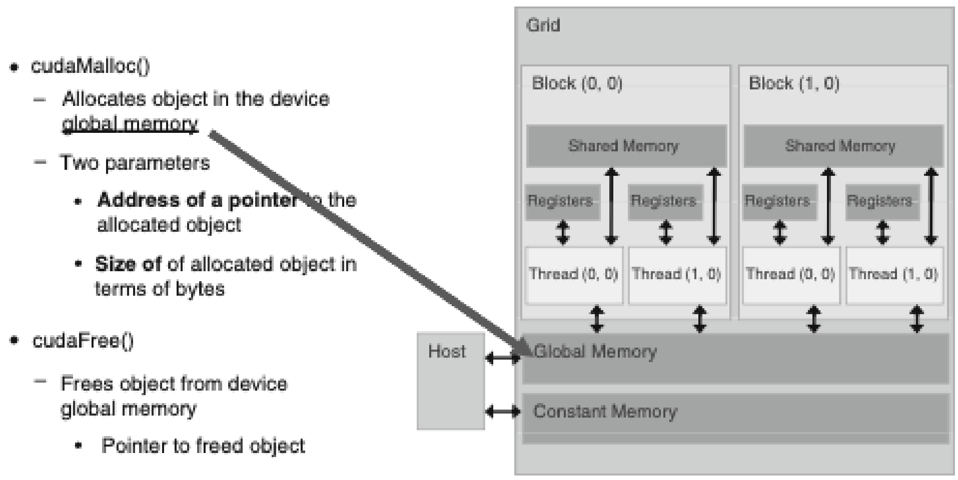
```
    // 3. I/O to write output matrix P
    //    Free matrices M, N, P

    return 0;
}
```

3.3. Device Memory & Data Transfer



- CUDA runtime system provides API functions to perform memory allocation and data transfer between host and devices.
- CUDA devices comprise global memory and constant memory; these are accessible from host code.
- Constant memory is read-only for device code.
- API functions cudaMalloc() and cudaFree() allocate and free global memory.
- API function cudaMemcpy() transfers data between host & device memory.



For cudaMalloc():

First parameter is a generic pointer (void), Second parameter is the size in bytes

Example:

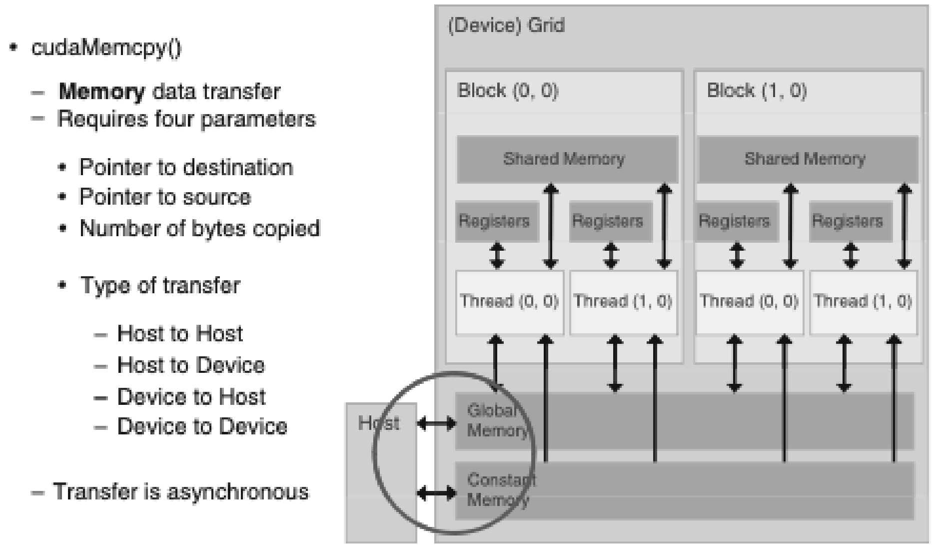
```
float *Md;
int size = Width * Width * sizeof(float);
cudaMalloc((void**)&Md, size);

// ...
cudaFree(Md);
```

For cudaMemcpy():

First argument is a destination pointer, Second is a source pointer, Third is number of bytes and Fourth is direction (host⟶device, device⟶host, device⟶device)

Note: cudaMemcpy() cannot be used for memory transfer in multi-GPU systems.



Example:

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

In the MatMul example, the main() function calls MatrixMultiplication(). MatrixMultiplication() allocates device memory, performs data transfers, and invokes the kernel that computes the result. This type of host-side function is called a stub function.

```
void MatrixMultiplication(float *M, float *N, float *P, int Width) {
    int size = Width * Width * sizeof(float);
    float *Md, Nd, Pd;

    // Allocate device memory for M, N, P
    cudaMalloc((void**)&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Pd, size);

    // Kernel invocation (not shown)
    // ...

    // Copy P & free device memory
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

```
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

3.4. Kernel Functions and Threading

In CUDA, a kernel function is executed by many threads in parallel. CUDA programming follows the single program, multiple data (SPMD) model.

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

Device functions cannot include recursion or indirect calls.

A function can be annotated for both host & device, generating two versions.

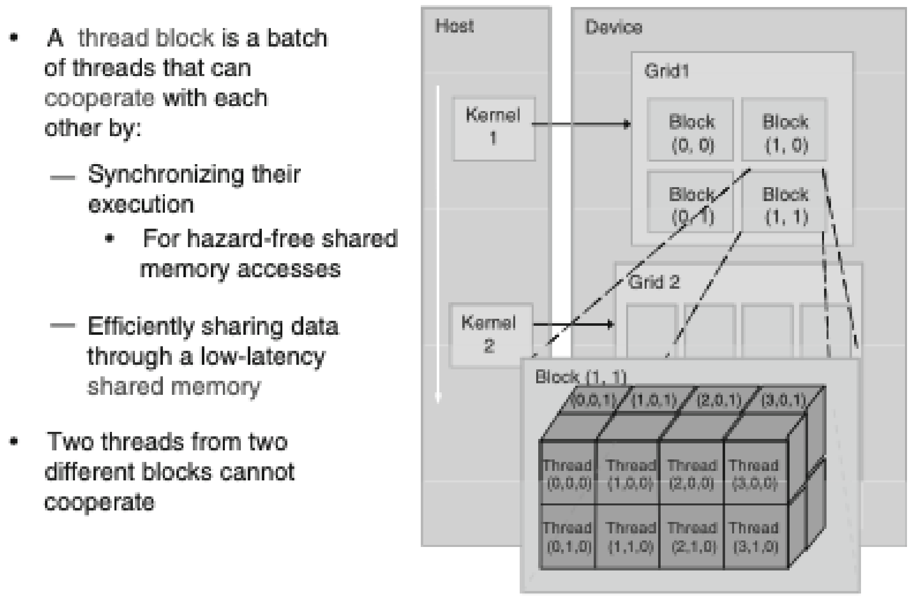
Variables like threadIdx.x and threadIdx.y give thread coordinates.

Thread notation: $\text{thread}_{\{\text{threadIdx}.x,\ \text{threadIdx}.y\}}$

Threads are organized hierarchically:

A grid contains one or more blocks

Each block has a unique 2D coordinate



A block is a 3D array of threads (max 512 threads): indexed by threadIdx.x, .y, .z

When launching a kernel, host code chooses grid & block size.

Example:

```
// Setup execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);
```

```
// Launch device computation threads!
c MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

4. CUDA Threads

4.1. CUDA Thread Organization

Threads in CUDA are organized in a two-level hierarchy using unique coordinates blockIdx and threadIdx. These are built-in, preinitialized variables, accesible in kernel functions.

In general, a grid is organized as a 2D array of blocks. Each block is organized into a 3D array of threads. The exact organziation is determined by execution configuration.