Parallel Processing Using Nvidia CUDA C Library

Prasanna Paithankar

Introduction

Part 01

Background

Parallel Computing

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters and grids use multiple computers to work on the same task. Specialized parallel computer architectures(Like GPU's) are sometimes used alongside traditional processors, for accelerating specific tasks.

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing

The Problem

What we want to solve

Some problems can be parallelized, i.e. can be broken down into smaller portions which are independent of the other portion. We here focus on such programs. We will review such programs of running loops whose loop body instructions may not interfere with the next iteration, like addition of two matrices.

The performance barrier

The CPU has been engineered to carry sequential tasks with extreme speeds and precisions. In a condition if we have a parallelizable program it would be better to have processing unit which may shed some speed for a sequential task in a thread but will have many such threads. This leads to net gain in processing power for parallel programs.

Objectives

What will be explored

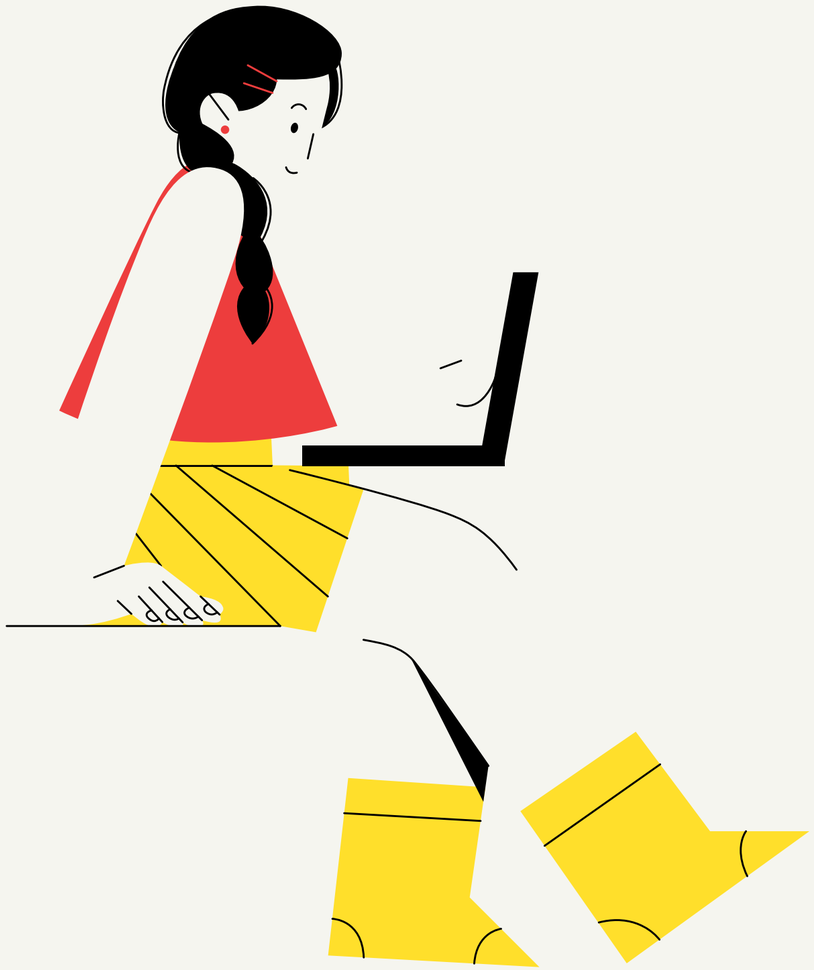
Types of Parallelism

General architectural differences between CPU and GPU

CUDA Parallel Programming Paradigm

Some basic program flow and method calls used in CUDA

Task demonstration

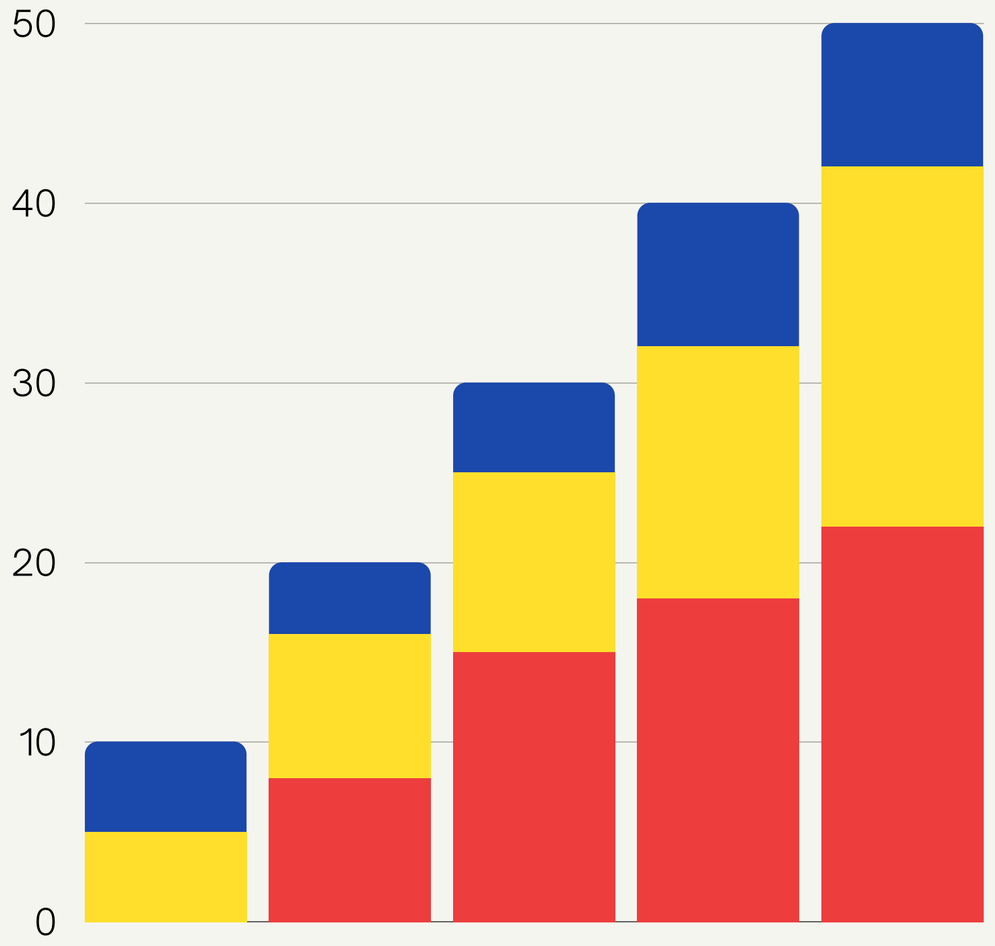


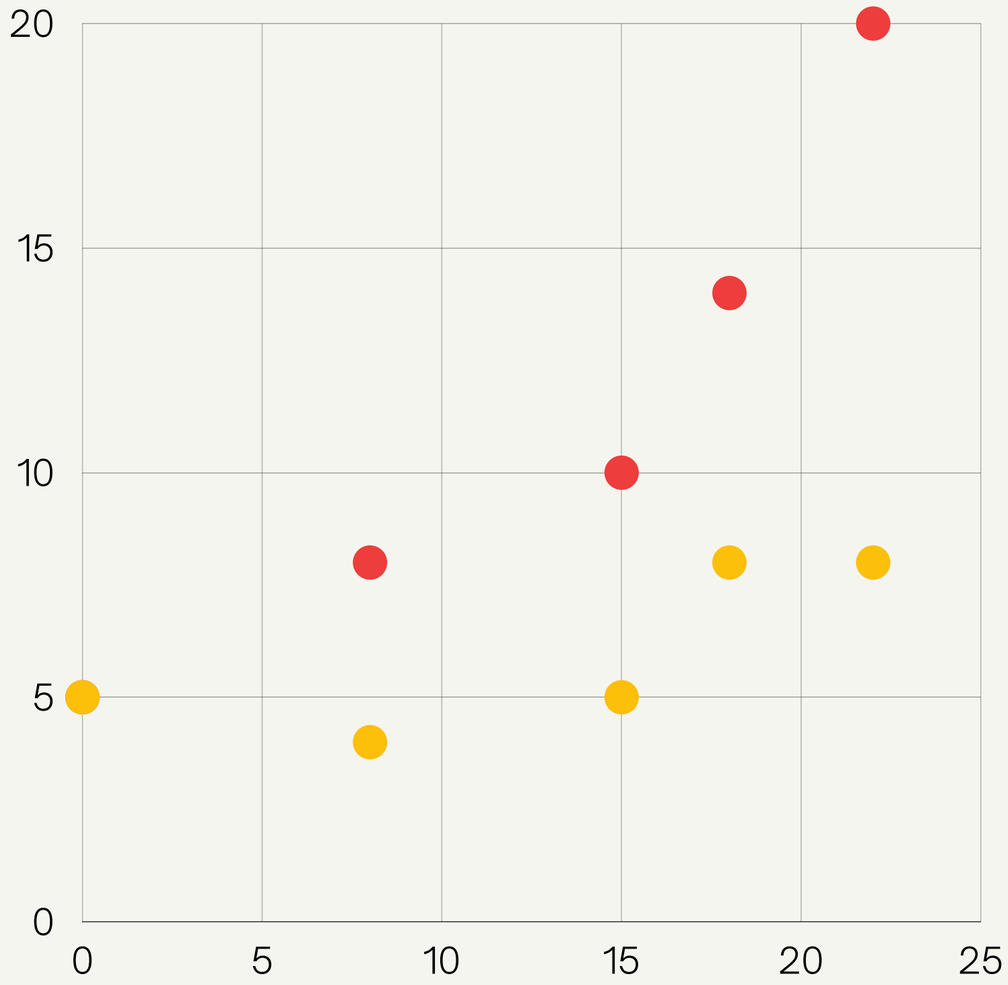
Parallelism

Part 02

Bit Level Parallelism

Speed-up in computer architecture can be driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.





Instruction Level Parallelism

Instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism.

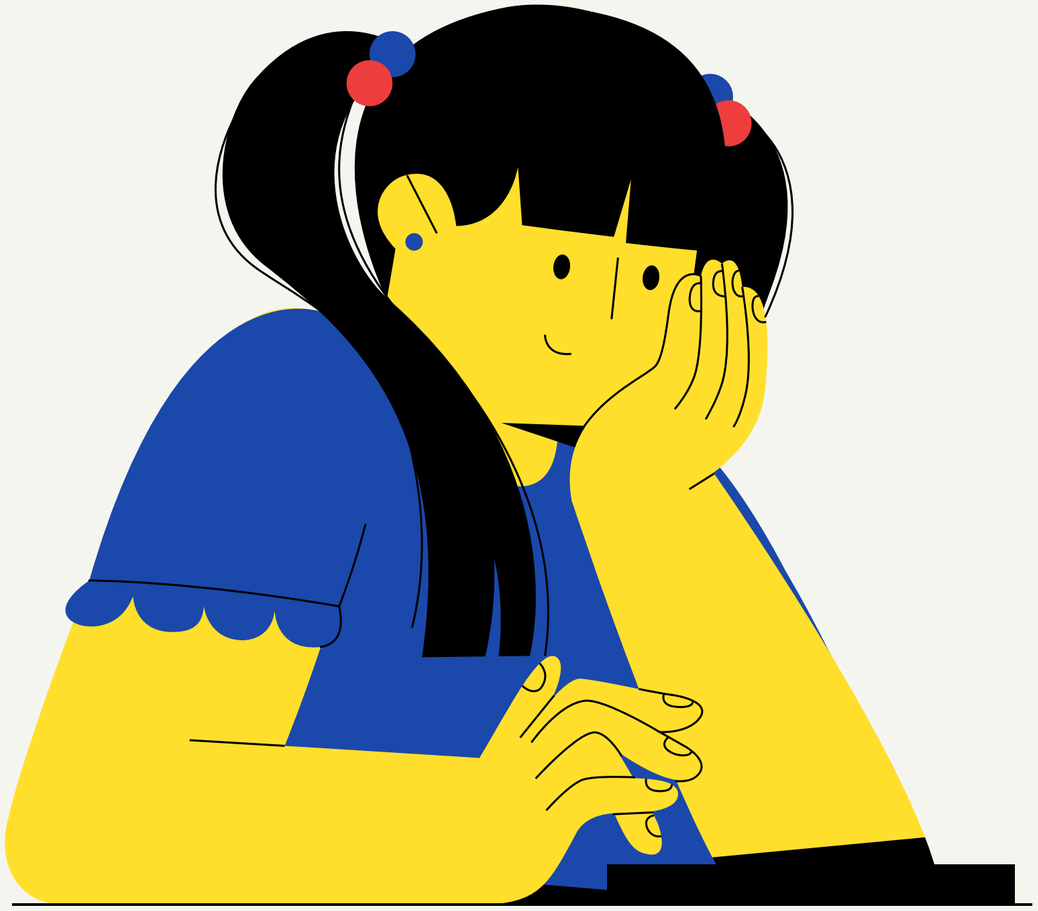
Instructions can be grouped together only if there is no data dependency between them.

All modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion and thus can issue one instruction per clock cycle (IPC = 1).

Combining execution unots with pipelining we can issue more than one instruction per clock cycle (IPC > 1). These processors are known as superscalar processors.

Task Parallelism

Task parallelisms is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data". This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism involves the decomposition of a task into sub-tasks and then allocating each sub-task to a processor for execution. The processors would then execute these sub-tasks concurrently and often cooperatively. Task parallelism does not usually scale with the size of a problem.



Hardware Architecture

Part 03

CPU

CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel

The CPU is designed such that more transistors are devoted to data caching and flow control

It emphasizes on low latency and normally needs more memory

Requires large cache and complex flow control which is expensive in terms of transistors

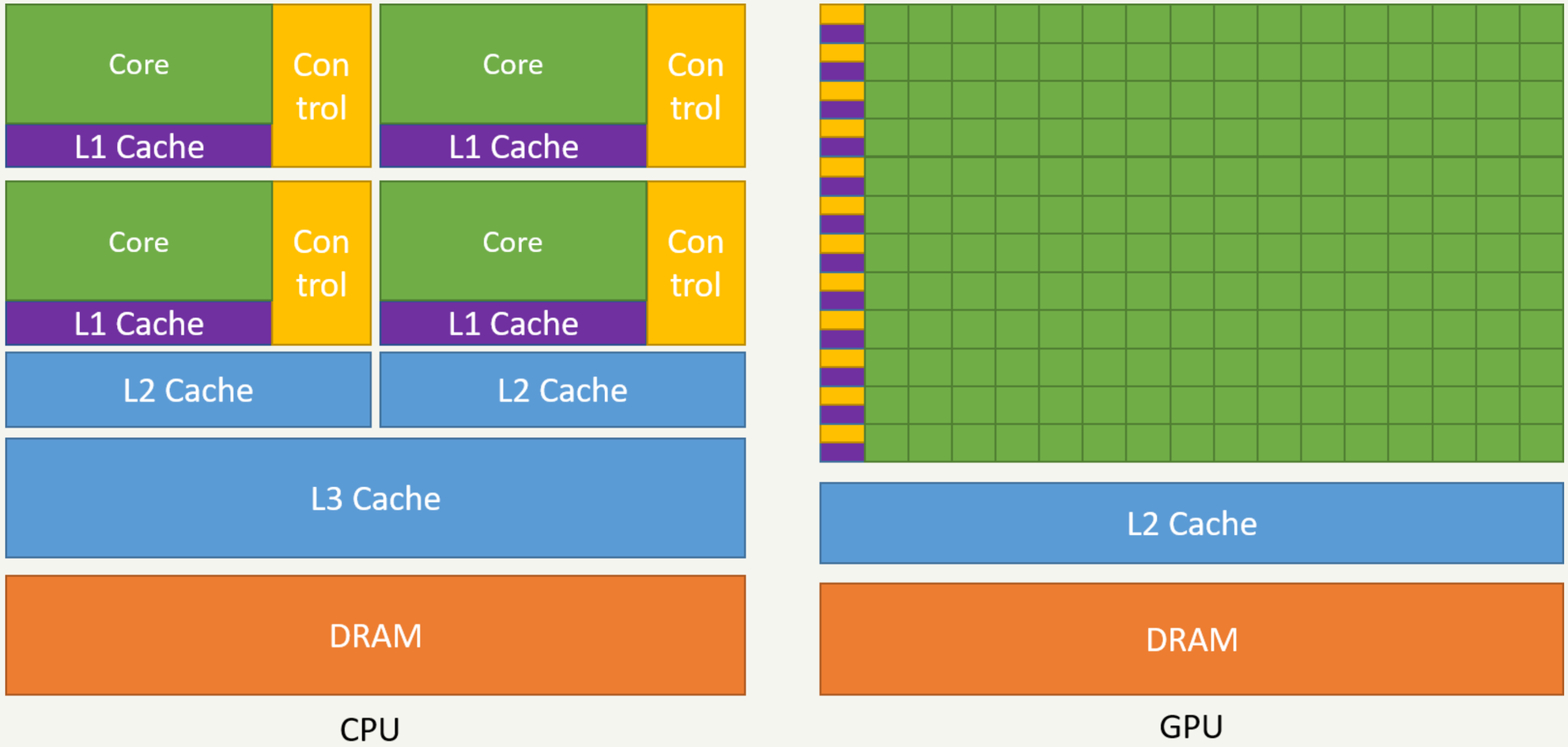
GPU

The GPU is designed to excel at executing thousands of them in parallel leading to slower single-thread performance to achieve greater throughput

The GPU is designed such that more transistors are devoted to data processing

It emphasizes on high throughput and normally requires less memory

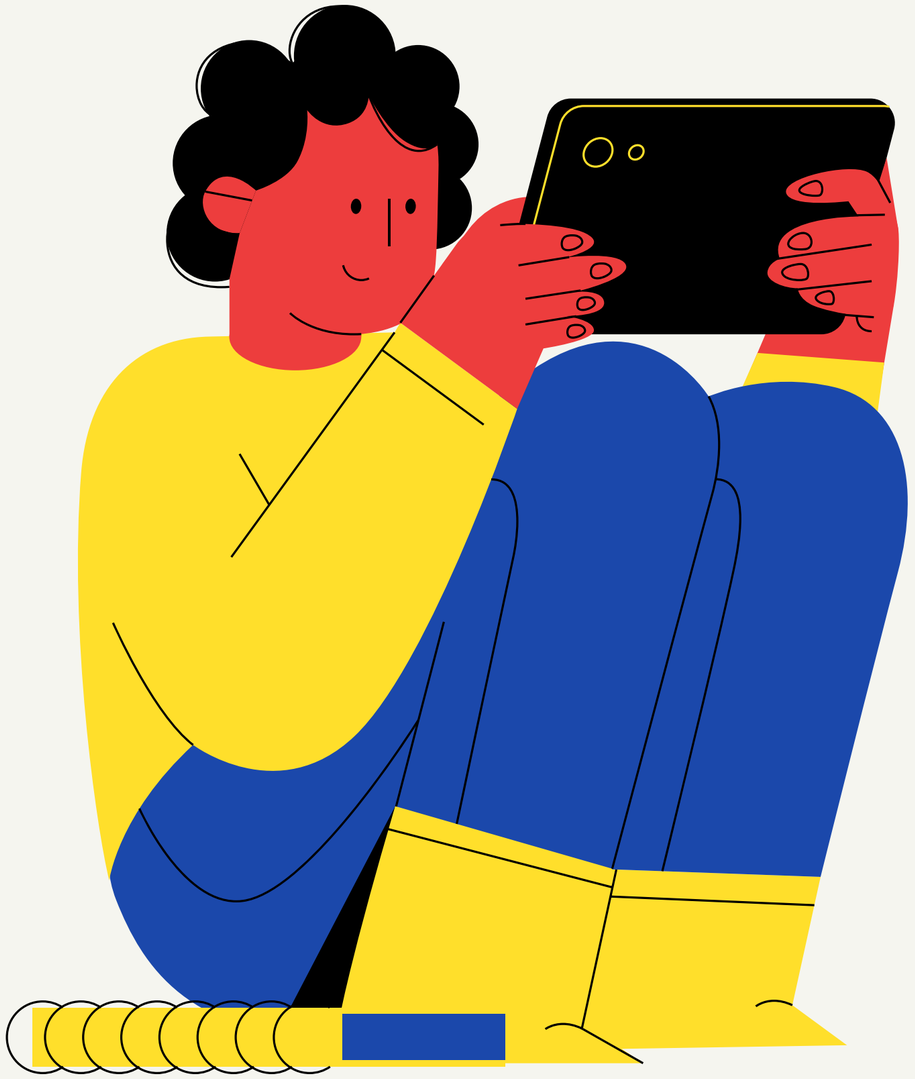
Requires minimal cache and less control in trade for less single thread performance



Reference: CUDA Toolkit Documentation

Kernels & Hierarchy

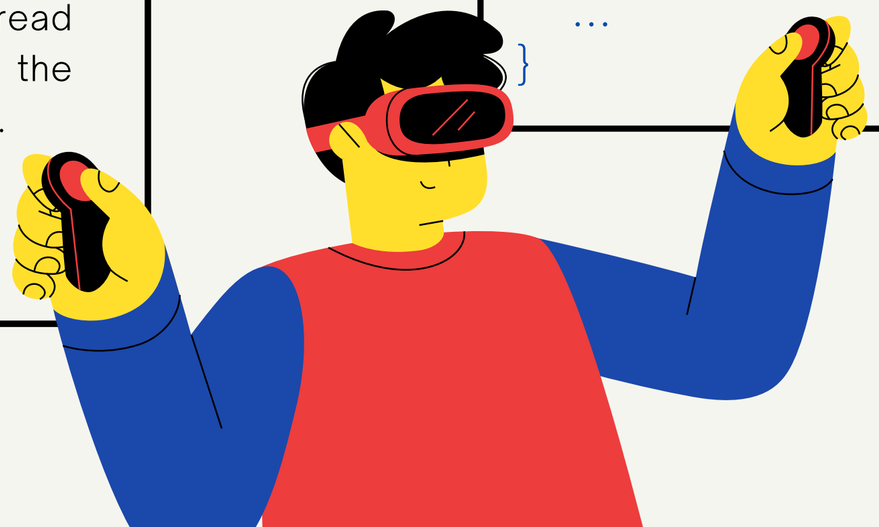
Part 04



We can define functions in CUDA called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions.
A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using <<<...>>>execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.

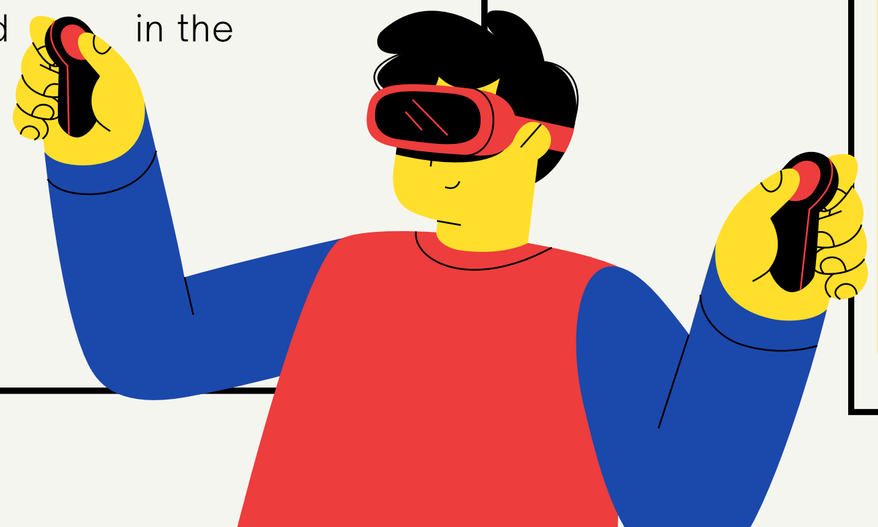
Compute Kernels

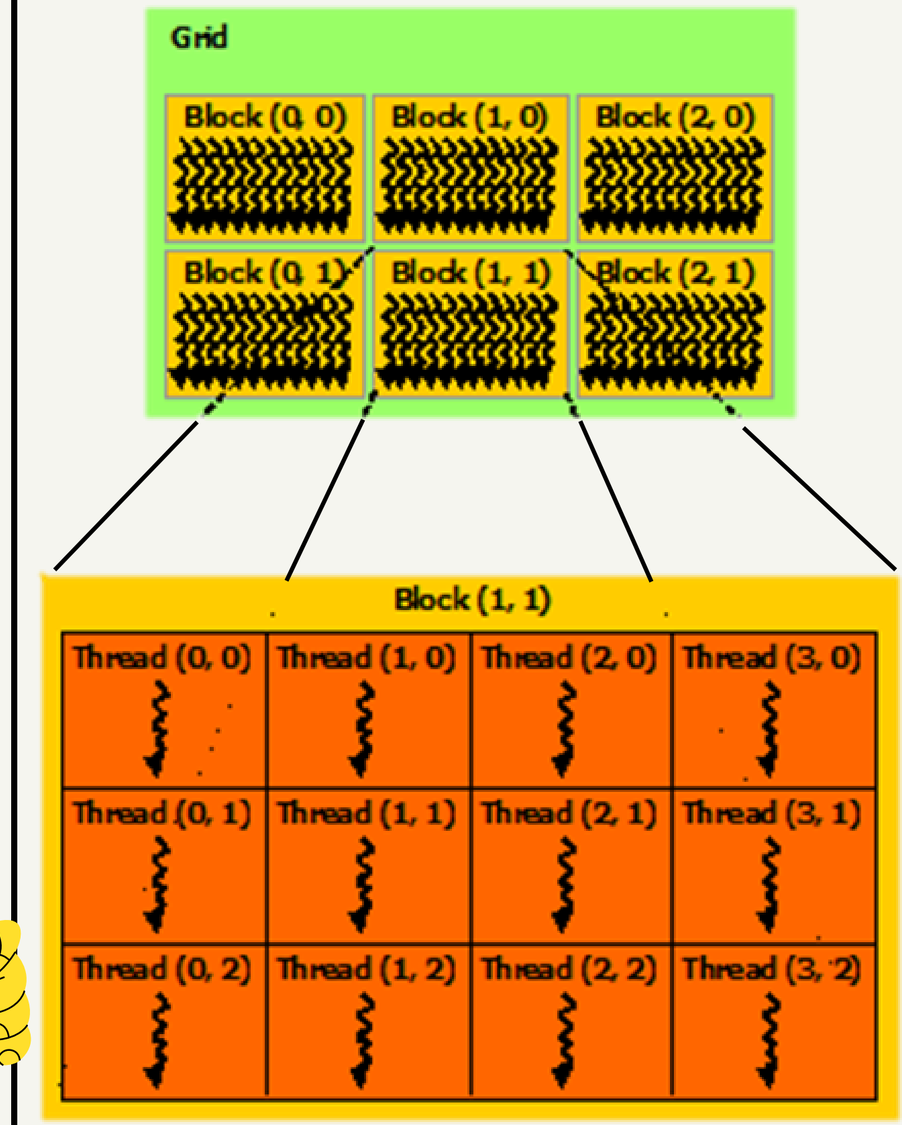
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```



threadIdx is a 3-component vector, so that threads can be identified using a 1D, 2D, 3D thread index, forming a 1D, 2D, 3D block of threads, called a thread block. This provides a way to invoke computation across the elements in a domain such as a vector, matrix, or volume.
There is a limit to the number of threads per block. On current GPUs, a thread block may contain up to 1024 threads.
However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.
The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3.

Thread Hierarchy



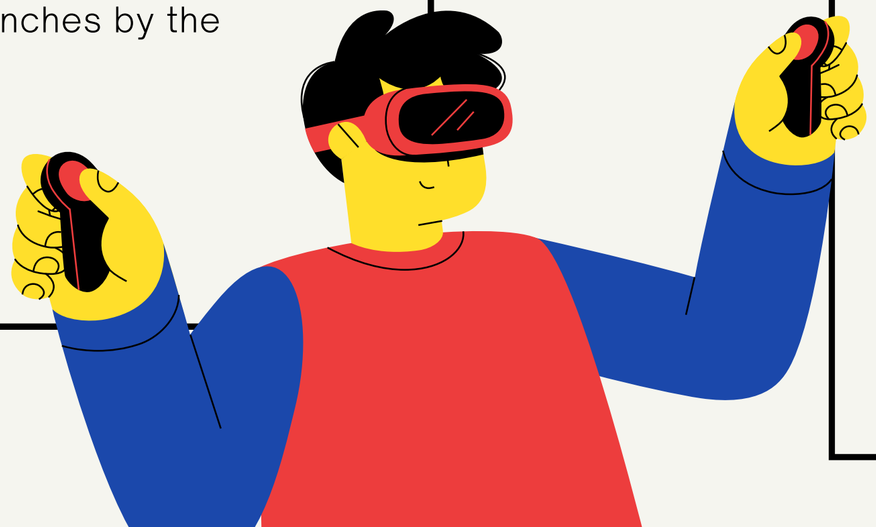


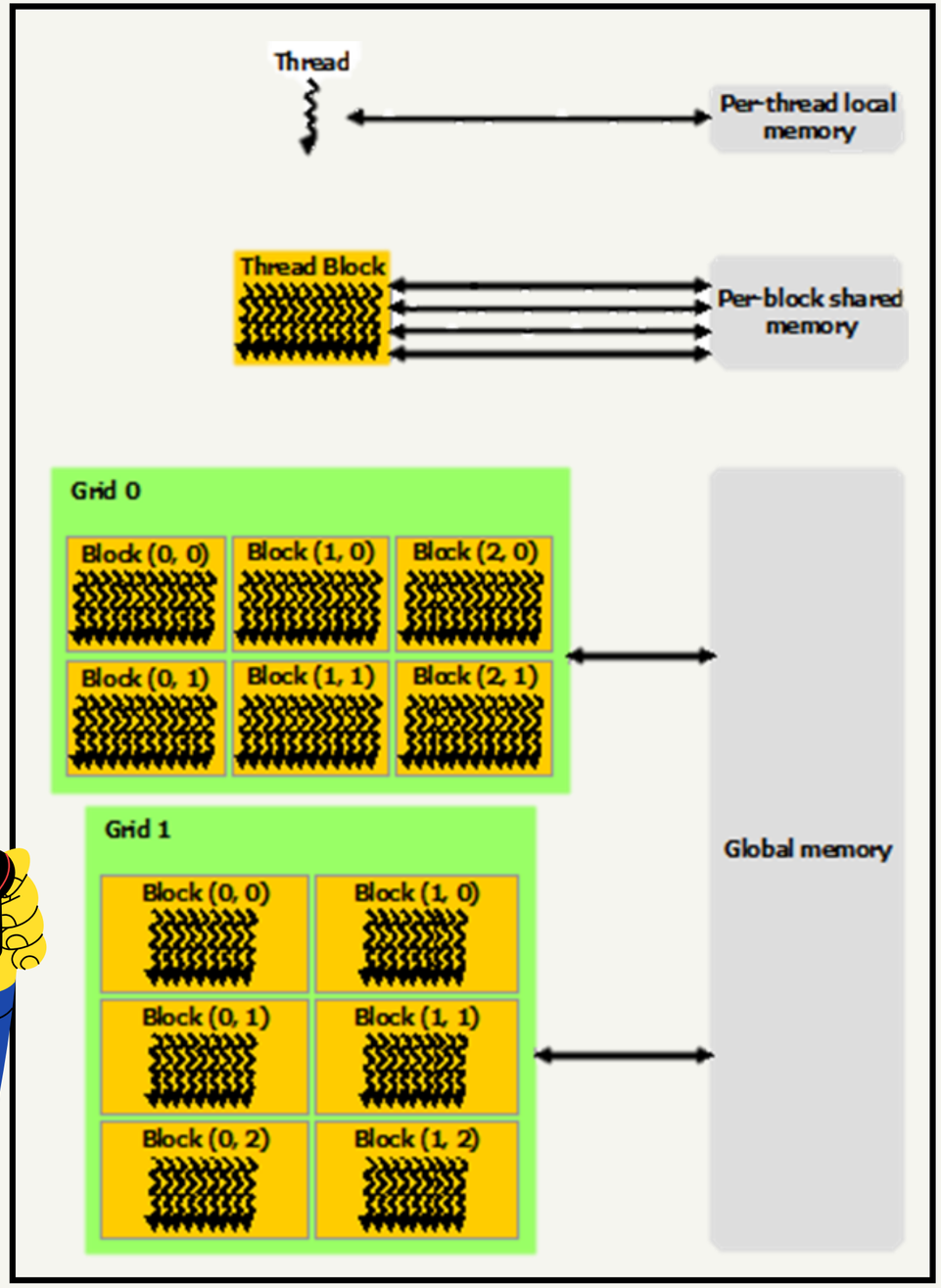
CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

Memory Hierarchy





Methods

Part 05

CUDA Basic Functions

cudaMemcpy()

__syncthreads()

cudaMalloc()

cudaFree()

cudaDeviceSynchronize()

cudaMalloc()

Allocating space for device copies
Equivalent to malloc() for CPU

When to

Memory allocation is performed if we want to keep a certain amount of memory reserved for a data structure, generally a vector or matrix during run time.

Function call

cudaMalloc((typecast) pointer, amount of space to be allotted);

e.g. cudaMalloc((void**)&a, sizeof(int));

Result of function

The amount of space is reserved in the GPU for the pointer variable just like malloc() does in classical C programs

cudaMemcpy()

Transferring data copies between Host (CPU) and Device (GPU)

When to

We generally initialize data structures in CPU, ehich to be processed need to be transferred to the GPU. Similarly the processed output needs to be transferred back to the CPU.

Function call

cudaMemcpy(receiver, source, transfer size, argument for direction of transfer);

e.g. cudaMemcpy(d, &a, sizeof(int), cudaMemcpyHostToDevice);

Result of function

The given data structure is copied to the receiver processing unit enabling further data operations to be carried out their.

cudaFree()

Manual garbage collector for GPU
Equivalent to free() for CPU

When to

After a data structure serves its purpose it could be manually destroyed in order remove redundant space blocking increasing the overall performance.

Function call

cudaFree(memory to be deallocated);

e.g. cudaFree(a);

Result of function

The given data structure is deallocated and sent to the heap clearing the active memory space for other variables.

Demonstration Code

```
__global__ void add(int *a, int *b, int *c) {
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

#define N 512
int main(void) {
int *a, *b, *c;
int *d_a, *d_b, *d_c;
int size = N * sizeof(int);

cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);

cudaMemcpy(d_a, a, size,
                    cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size,
                    cudaMemcpyHostToDevice);

add<<N, 1>>(d_a, d_b, d_c);

cudaMemcpy(c, d_c, size,
                    cudaMemcpyDeviceToHost);

free(a);
free(b);
free(c);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

__syncthreads()

Thread synchronization in a block

When to

It is used to prevent data hazards such as RAW/WAW/WAR.

Function call

```
__syncthreads();
```

Result of function

It does so by synchronizing all the threads in their respective blocks and making the conditional flow uniform across blocks.

cudaDeviceSynchronize()

Blocks the CPU to execute CUDA calls

When to

The code naturally runs on the CPU and the CUDA code is asynchronous with it. To maintain a controlled flow by the programmer the function can be called at appropriate instances.

Function call

```
cudaDeviceSynchronize();
```

Result of function

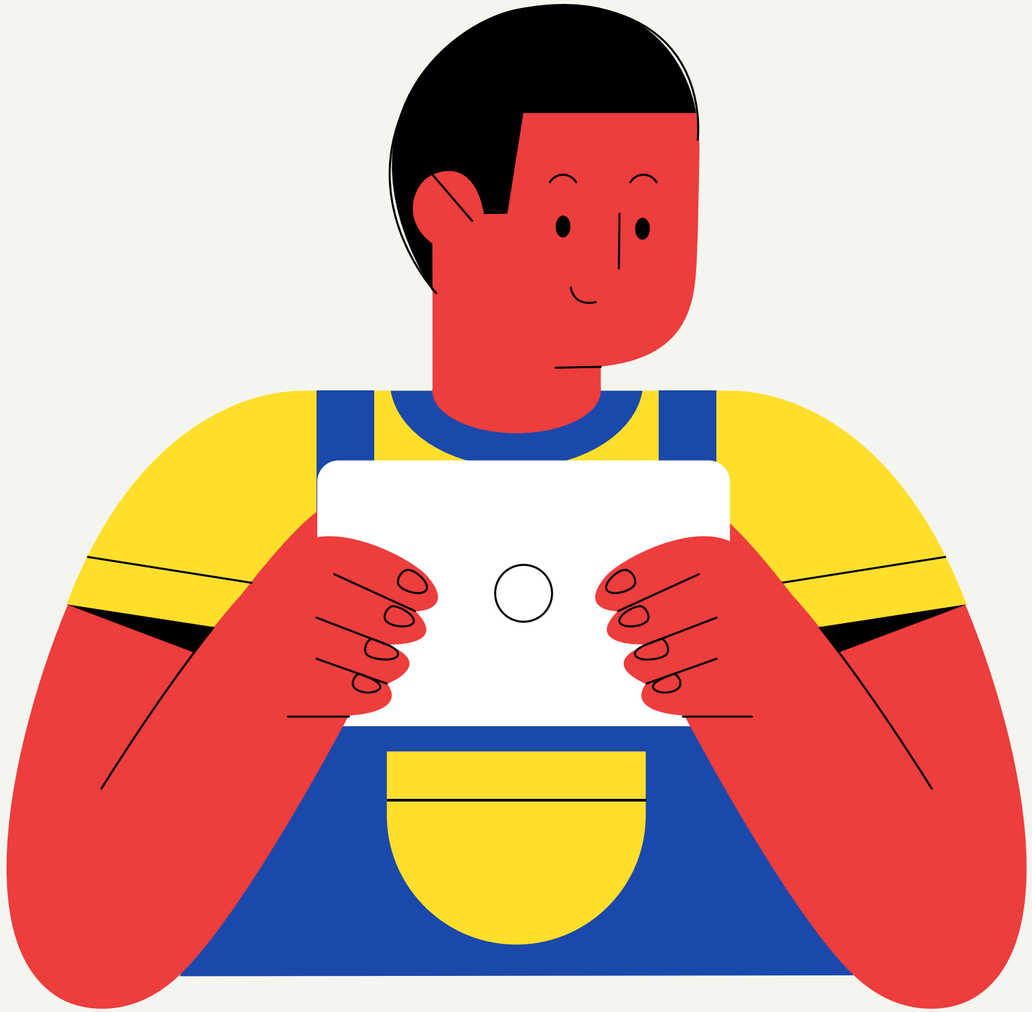
The CUDA calls start executing after this command and then the further processing of CPU commands start.

Demonstration Code

```
__global__ void add(int *a, int *b, int *c) {
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
__syncthreads();
}

#define N 512
int main(void) {
int *a, *b, *c;
int *d_a, *d_b, *d_c;
int size = N * sizeof(int);

cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);

cudaMemcpy(d_a, a, size,
                cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size,
                cudaMemcpyHostToDevice);

add<<N, 1>>(d_a, d_b, d_c);

cudaDeviceSynchronize();
cudaMemcpy(c, d_c, size,
                cudaMemcpyDeviceToHost);

free(a);
free(b);
free(c);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

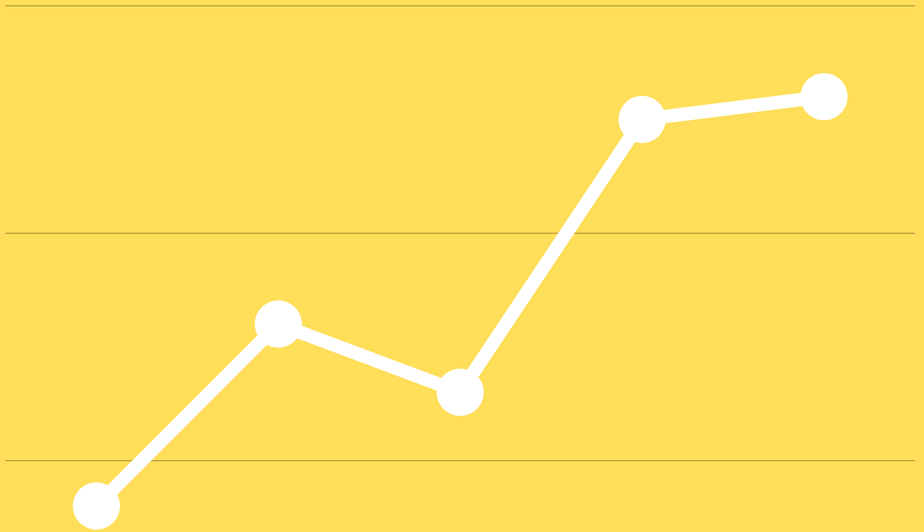
Task

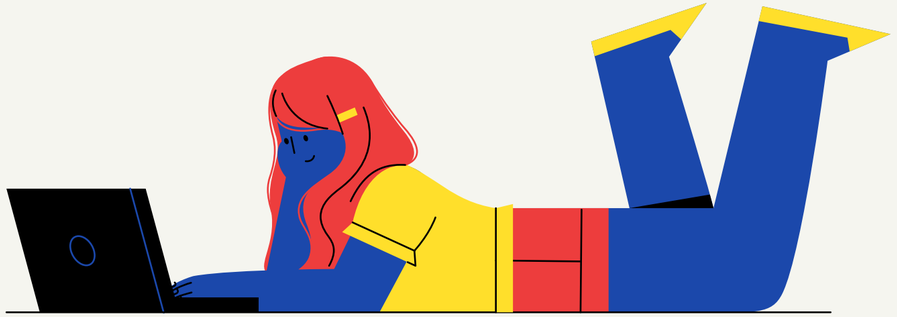
Part 06



Google Colaboratory

google.com





Thank You