

Chapter 5 GPU Architecture & Programming

Contents of this chapter

- *Why GPU ?*
- Types of GPU & using a GPU
- SIMD/T Models for GPU
- GPU Basics & Software (CUDA) architecture
- Example 1 – Vector addition on GPU
- Example 2 - Matrix Multiplication on GPU
- Example 3 – Parallel Reduction using CUDA

References:

- Gerassimos Barlas, "Multicore and GPU Programming-An Integrated Approach", Morgan Kaufmann, ISBN: 978-0-12-417137-4, 2015
- NVIDIA pages & whitepaper

Basics of GPU Architecture & Programming

- Single core/CPU Performance essentially getting saturated;
- Multi-core CPU based chips – supportive to a large class of application problems until 2011/2012;
- Sudden increase in demand for computational power – Big Data, HPC needs, medical and financial data processing, high-precision engineering;

Solution ?

- **Parallelization** – Code and Data;
- What is the architecture that supports this realization?
- What is the programming model?

Graphics Processing Unit (GPU)

Graphics processing is “inherently parallel” and there is a lot of parallelism that can be exploited; Typically we can say, **O(pixels) is the computational demand;**

Initial designs targeted two aspects:

- Graphics applications;
- Hardwired (less programmable) to provide speed and parallelism;

~2005 onwards GPUs started getting attention for non-graphic computations and accommodated programming!

Throughput performance - GPU

Year 2006 – **Nvidia** releases “**CUDA**” language for GPUs and started fully supporting non-graphic applications;

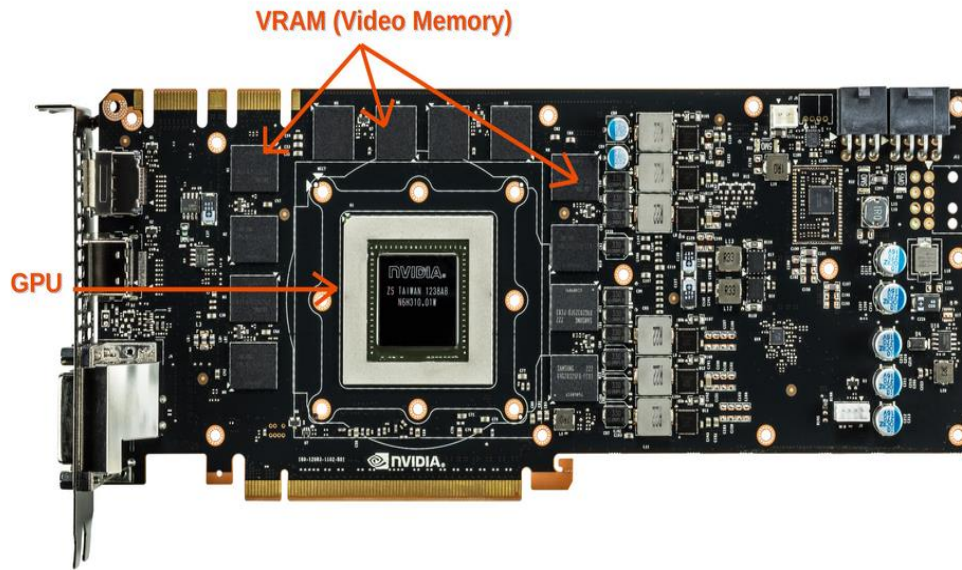
For ML/DL Applications – Some performance comparison

- **Maximum Batch size** – Refers to the largest number of samples that can be fit into the GPU memory
- **Throughput** - Refers to the number of samples that can be processed per second by a GPU
- **Throughput per \$** - Refers to a normalized price of the GPU. This tells us how "cost-effective" the GPU is regarding the computation/purchase price ratio

Model / GPU	A100 80GB SXM4	RTX A6000	RTX 3090	V100 32GB	RTX 8000	Titan RTX	RTX 2080Ti	3080 Max-Q
ResNet 50	925	437	471	368	300	306	275	193
ResNet 50 FP16	1386	775	801	828	646	644	526	351
SSD	272	135	137	136	116	119	106	55
SSD FP16	420	230	214	224	180	181	139	90
...								

Ampere GPU

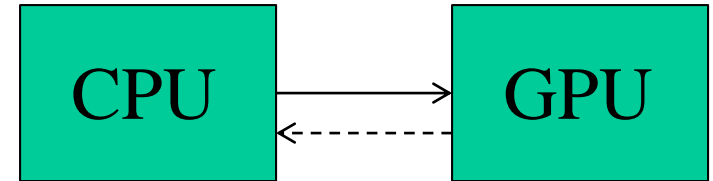
Quick glimpse on a GPU!



- Graphics card – This is a printed circuit board that holds - a processor, a memory, and a power management unit.
- Graphics card also has a BIOS chip, which retains the card's settings and performs startup diagnostics on the memory, input, and output
- For a graphics card, the GPU is connected to the memory unit Video Random Access Memory (VRAM) via a Bus called the memory interface bus
- GPU's memory bandwidth - Determines how fast a GPU can move data from/to VRAM to the computation cores. It is determined by the data transmission speed between memory and computation cores

Three types of GPU

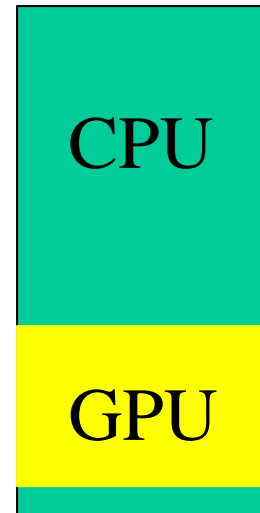
1. **CPU+GPU** combo; Offers more computational power and memory bandwidth; Discrete type;



2. Types 2 & 3 – Integrated GPUs

Shared-memory System;

Energy considerations





SFU - The Fermi GPUs have **Special Function Units** (SFUs) to (NVIDIA White Paper on Fermi) execute transcendental instructions such as, *sine*, *cosine*, *reciprocal*, and *square root*. Each SFU executes one instruction per thread, per clock".

DP Unit - Double Precision unit to carry out complex algebraic and scientific operations

L1 – can be partitioned

How to use a GPU?

Some common golden rules:

1. You must **retarget** code for the GPU
2. The working set must fit in GPU RAM
3. You must copy data to/from GPU RAM
4. Data accesses should be **streaming**
5. Use **scratchpad** as user-managed cache
6. **Lots of parallelism preferred** (throughput, not latency)
7. **SIMD-style parallelism best suited**
8. **High arithmetic intensity** (FLOPs/byte) preferred

Remarks: Caches vs Scratchpads

CPU caches are "automatically" managed by the hardware together with OS- when the requested memory contents are not in the cache, fetches that data from main memory.

In modern architectures, caches are generally a hierarchy, with level 1 caches being very fast to access, but small, and level 2 and higher caches being larger and slower.

Most of the time, a programmer hopes that accesses hit in L1 or L2 caches, so the processor spends little or no time waiting on memory.

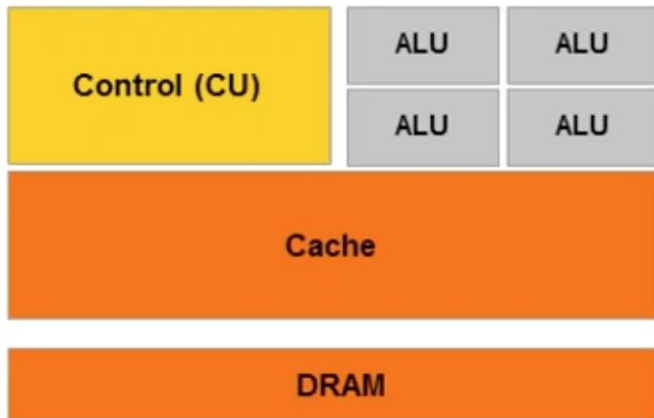
Scratchpad memories are "manually" managed: a program explicitly addresses the memory, writing results and retrieving them.

Scratchpads are usually relatively small, on the order of L1 or L2 caches, fast (1-2 cycle access), and often more importantly, exhibit deterministic behavior/performance;

This means, if you write a scratchpad, the data are always there and ready to go, unlike a cache, where the contents might have been evicted and need to be retrieved.

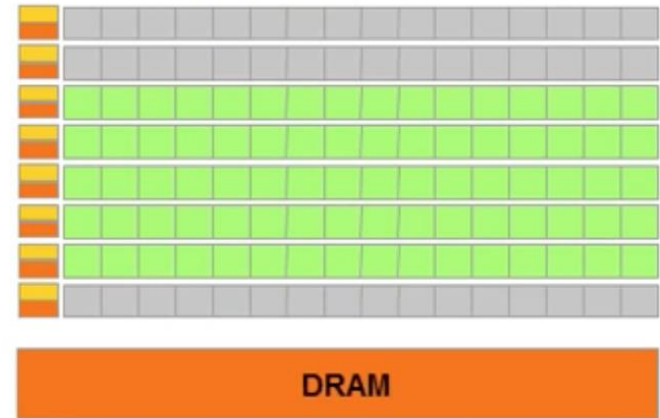
CPU vs GPU

Multi Core Latency Oriented CPU



- Less no. of core
- Higher Clock Freq.
- Lower Bandwidth
- Less FLOPS/watt
- Less Peak Performance (TFLOPS)
- More total Memory

Many thread Throughput Oriented GPU



- More no. of core
- Lower Clock Freq.
- Higher Bandwidth
- More FLOPS/watt
- More Peak Performance
- Less total Memory
- Heterogeneous: GPU attached to CPU via slow PCIe

SIMD/T model for GPUs

Data Parallel Computation

- All cores execute single instruction at any given clk cycle;
- Each instruction operates on different data elements;
- **SIMT** - Version of SIMD used in GPUs;
- GPUs use a thread model to achieve very high parallel performance and to hide memory latency;
- Multiple threads, each execute the same instruction sequence; On a GPU, a very large number of threads (10K) possible;
- Threads mapped onto a available processors on GPU (1000's of processors all executing same program sequence)

Basics of GPU CUDA Programming

CUDA - Compute Unified Device Architecture - **Heterogeneous model**

- Architecture & programming model - NVIDIA, 2007
- Enables GPUs to execute programs written in C, Fortran, etc
CUDA C – special APIs, syntax and kernel functions added to C to in the design of **CUDA C**
- CUDA code has two components – **Host** & **Device**;
Host component (executes on CPU) & Device component (on GPU)
- Within C programs, we can call SIMT “kernel functions” that are executed on GPU; Kernels – data parallel functions that aid execution on GPUs;

Basics of GPU Programming

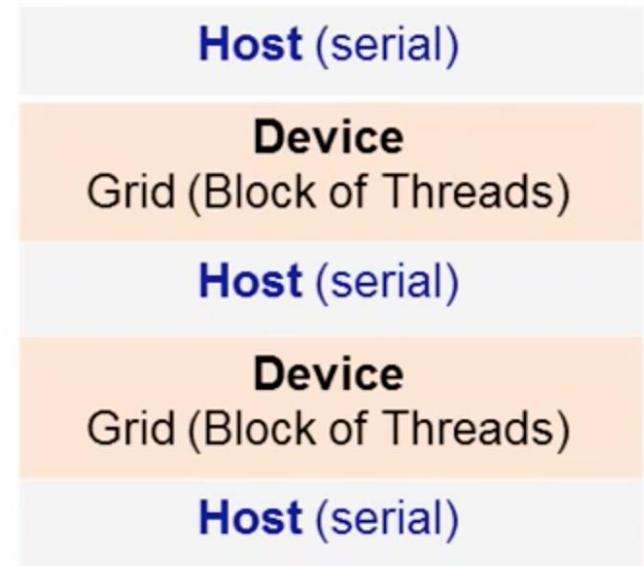
Heterogeneous :

CPU → host (own memory) traditional compiler

GPU → device (own memory) CUDA compiler

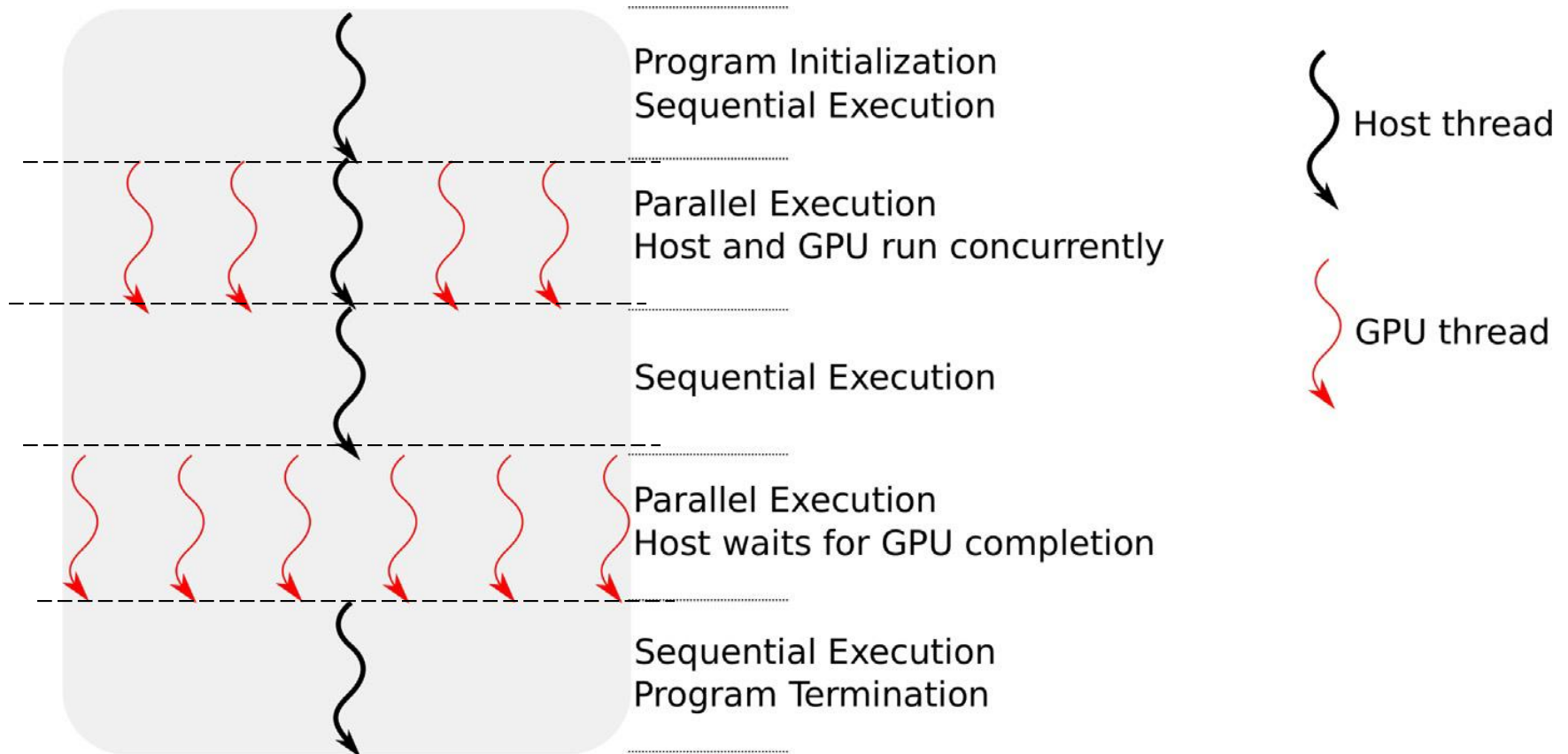
Functions of CPU:

1. Execute serial code;
2. Allocate GPU memory;
3. Copy data CPU to GPU;
4. Launch “kernel” on GPU;
5. Copy data GPU to CPU;
6. Error handling;



Kernels (data parallel portion) → serial program executed by threads
Possible to launch hundreds/thousands of thread (on cores)

CUDA model of execution



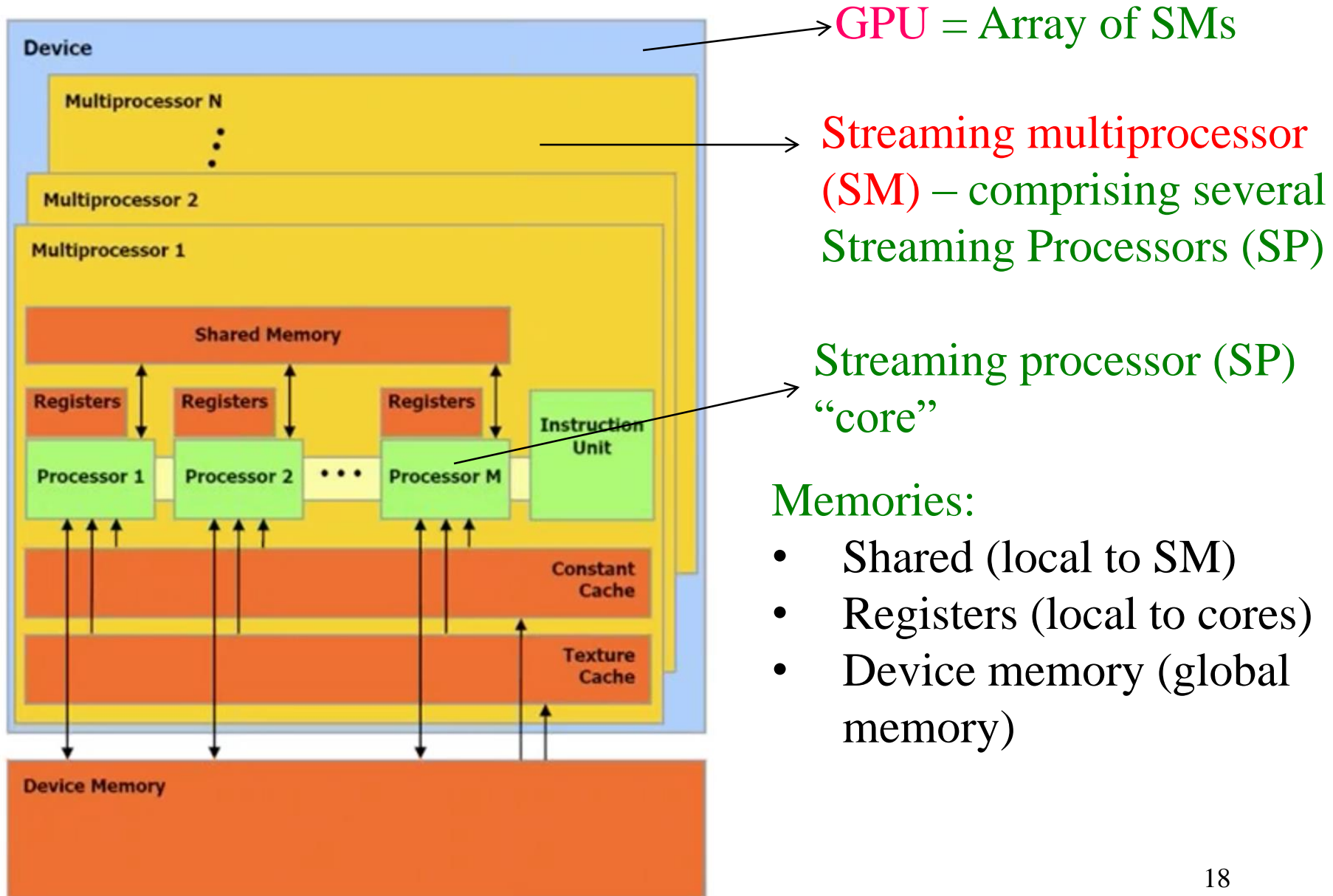
Basics of GPU Programming

Written in C with extensions - CPU code (runs on CPU) & GPU code (runs on GPU)

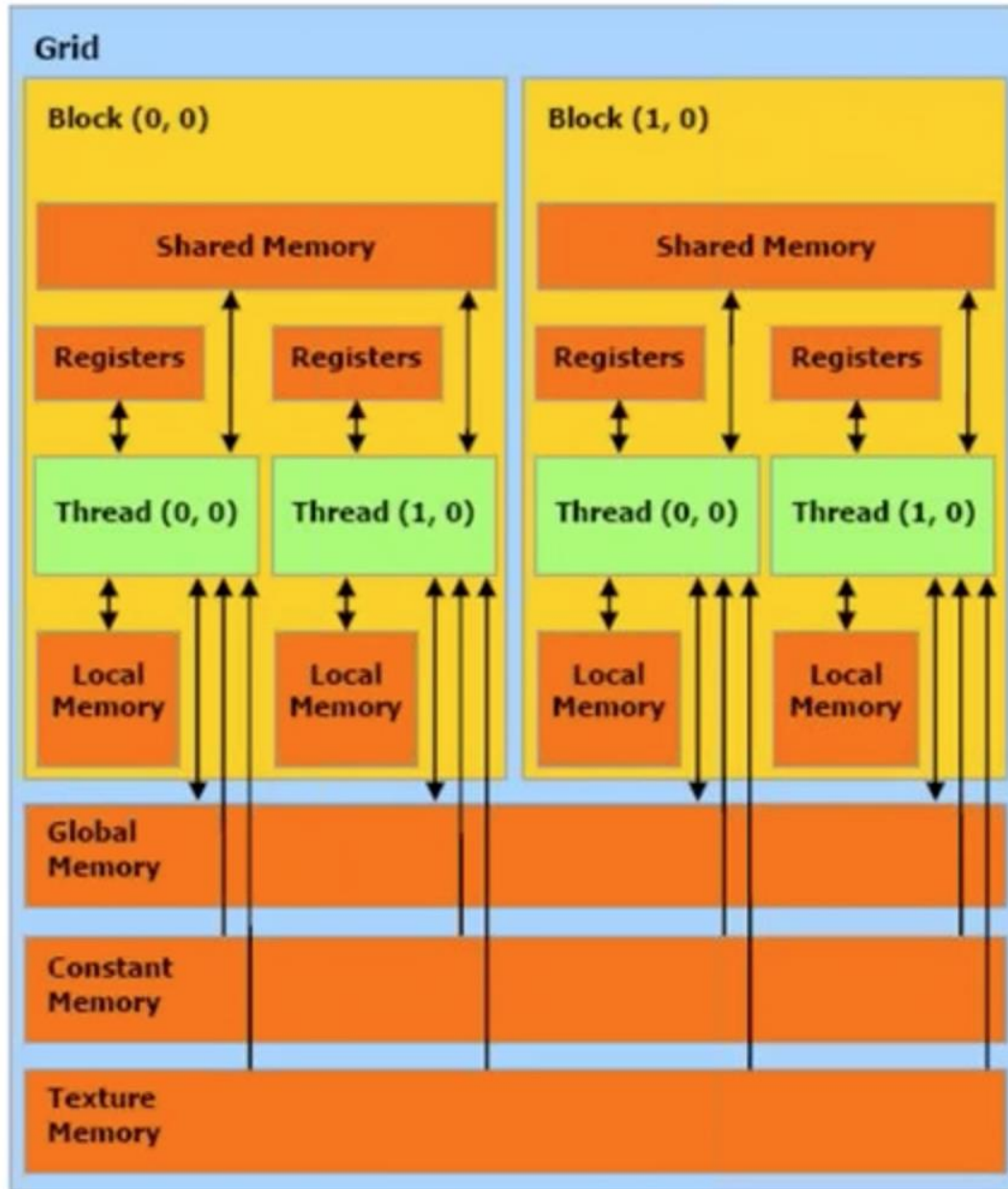
CUDA keywords separate the host and device codes;
Execution steps in order:

- CPU allocates storage on GPU - `cuda malloc()`
- CPU copies input data from CPU to GPU – GPU memory;
- CPU launches kernels on GPU to process data - kernel launch;
- GPU runs lots of threads in parallel
- CPU copies results back to CPU from GPU - `cud memcpy()`;

Basics of GPU Software Architecture



Basics of GPU Programming



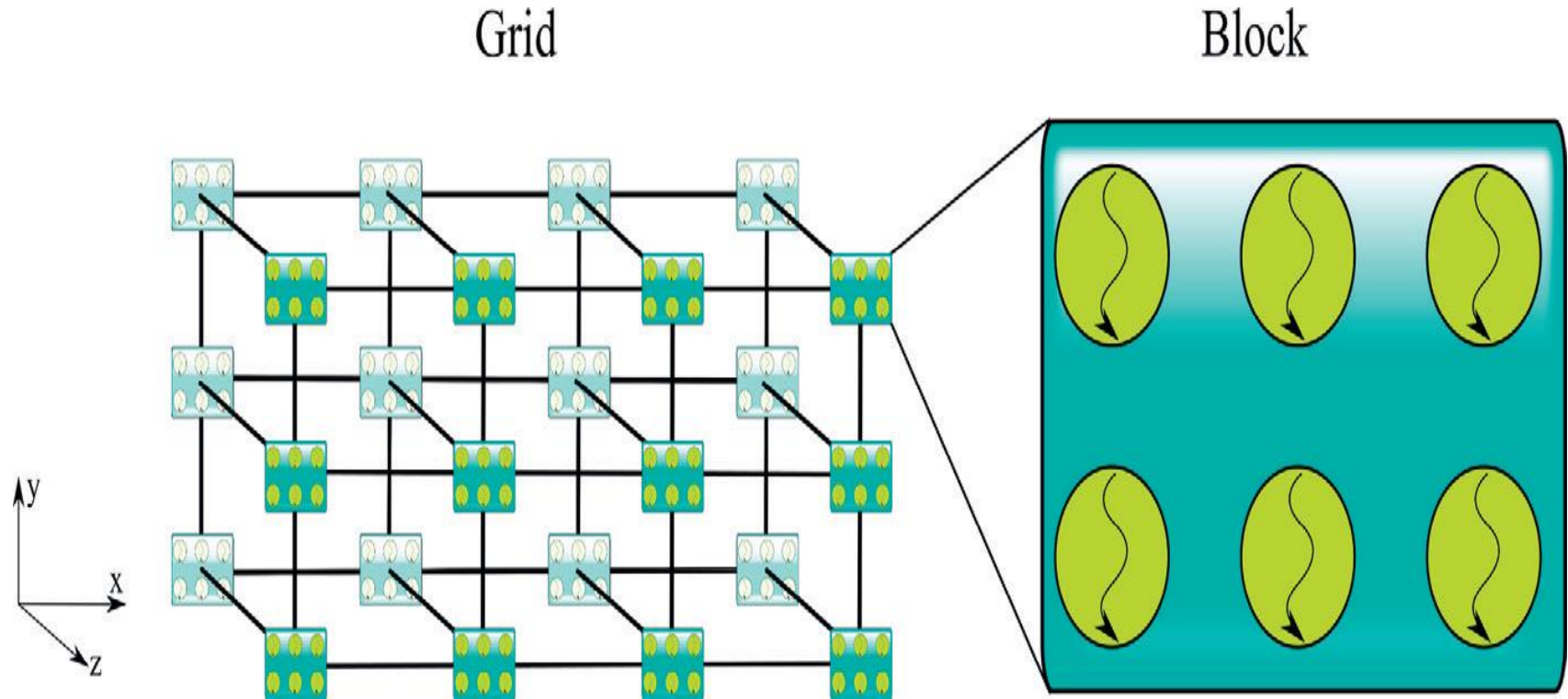
A thread corresponds to a core;

Collection of threads form a block; So, a block corresponds to a multiprocessor;

Collection of blocks forms a Grid; So grid is the kernel that is launched on the GPU by the CPU;

Observe “local mem” & “registers” per thread and a shared mem per block;

Grid-Block hierarchy



(Connections between blocks are only for illustration)

Basics of GPU Programming



- Each core can access any memory at
~100s Gb/s, but with different latencies;
- Shared memory - small latency;
- Device memory - ~100x slower than shared;
- GPU executes kernel grids;
- SMs executes one or more thread blocks;
- SM executes threads in groups of 32 threads called a
“warp”

Basics of GPU Programming

Thread hierarchy

Kernels composed of many **threads**;

All threads execute the same code independently;

Threads are grouped into “thread blocks”

Threads in the same block can cooperate;

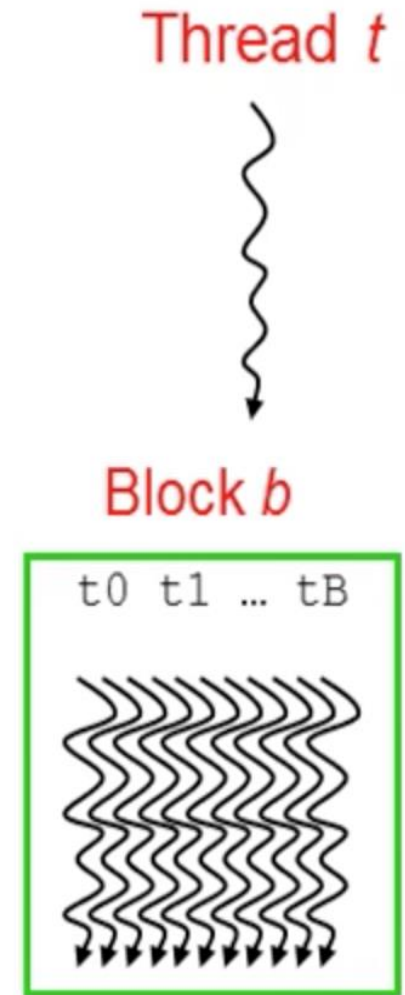
Blocks are grouped into a **Grid**

Threads/blocks have unique IDs;

Hardware virtualization mapping:

Thread – Virtualized SP/core;

Block – Virtualized SM



Dataset - Partitioned into blocks; Each block referred to as Thread Block (TB); Each TB itself is partitioned into smaller parts called as *warps*.

GPU H/W specifics:

- Broken into large # of **SIMT cores**;
- 1 Thread Block runs on 1 SIMT core;
- One SIMT core is responsible for executing a set of warps. So, there will be a Ready Warps Queue (RWQ) that will hold ready warps and in every cycle a warp is read from RWQ and executed on that SIMT core;
- Each SIMT core comprises a L1 cache; and a large *Reg File* (*not shown in the figure*) as context-switching needs to be supported between warps handled within a SIMT core!
- L1 cache shared by all the warps; If L1 has a miss then, using a crossbar interconnect, access L2 cache to get the data; If L2 gets a miss, then with the help of mem controller unit, we access external memory (high speed graphics interface tech is used)

Basics of GPU Programming

The “kernel” code is a regular C code, however, instead of usual array index, we use “thread IDs” as indices to access data;

```
index = ThreadID;
```

```
C[index] = A[index]+B[index];
```

How do we identify codes that are executed on CPU (host) and GPUs(device)?

Functions	Executed on the:	Only Callable from the:
<code>__device__ float DeviceFunc()</code>	DEVICE	DEVICE
<code>__global__ void KernelFunc()</code>	DEVICE	HOST
<code>__host__ void HostFunc()</code>	HOST	HOST

Basics of GPU Programming

Vector addition example using CUDA C

```
int main() {  
    // declare arrays A,B,C and initialize  
    for(i=0;i<n;i++)  
        C[i]=A[i]+B[i];  
}
```

1. Declare CUDA variables
`float *d_A,*d_B,*d_C;`
2. Allocate GPU memory for A,B,C using `cudaMalloc(...)`
3. Launch the **kernel** `<...>` to
Actually perform vector +
using a number of threads;
4. Copy back the results to CPU
using `cudaMemcpy(...)`

Basics of GPU Programming

Example 1: Vector addition example: Adding two one-D arrays;

1. Allocate memory space in “host” for data

```
// regular C
int *h_a, *h_b, *h_c;
...
h_a = (int*)malloc(size);
h_b = (int*)malloc(size);
h_c = (int*)malloc(size);
```

2. Allocate memory in “device”(GPU) for data

```
// we use CUDA malloc routines

int size = N*sizeof(int); //space for
N ints
int *devA, *devB, *devC; // device
pointers

cudaMalloc((void**) &devA,size));
cudaMalloc((void**) &devB,size));
cudaMalloc((void**) &devC,size));
```

Basics of GPU Programming

3. Transferring data from CPU to GPU

For this, we use cuda routine called: `cudaMemcpy()`

```
cudaMemcpy(devA, h_A, size,  
cudaMemcpyHostToDevice);
```

```
cudaMemcpy(devB, h_B, size,  
cudaMemcpyHostToDevice);
```

where, `devA` & `devB` points to destination in the device;

`h_A` & `h_B` are pointers to Host data; 4th parameter specifies the direction of transfer;

4. Declaring Kernel routine to run on GPU

Kernel call from Host code;

This contains information of threads using two parameters:

```
myKernel <<< n, m >>>(arg1,...);
```

where:

`n` - number of blocks to be used

`m` - number of threads in this block

For this example: we will set: `n=1, m=N`

`arg1,...` specifies arguments - typically pointers to device memory obtained from `cudaMalloc()` earlier;

Basics of GPU Programming

How do we write this kernel?

```
vecAdd<<<1,N>>>(devA,devB,devC);  
// Grid has 1 block with N threads in a blk;
```

How do we actually launch it? (launched from the Host)

```
__global__ void vecAdd(int *A, int *B, int *C);  
{  
    int i = threadIdx.x; // cuda structure that provides thread id in the block  
    C[i] = A[i]+B[i];  
}
```

Note:

Thread j: $\text{devC}[j] = \text{devA}[j] + \text{devB}[j]$, $j=0, N-1$

Basics of GPU Programming

5. Transferring data from GPU to CPU

For this, we use cuda routine called: `cudaMemcpy()` as before;

```
cudaMemcpy(C, devC, size,  
cudaMemcpyDeviceToHost);
```

where, `devC` is a pointer in the device; `C` is the pointer in the host;

4th parameter specifies the direction of transfer;

6. Free the memory!

Frees the object from device global memory; has only one parameter - pointer to the object to be freed;

```
cudaFree(devA);  
cudaFree(devB);  
cudaFree(devC);
```

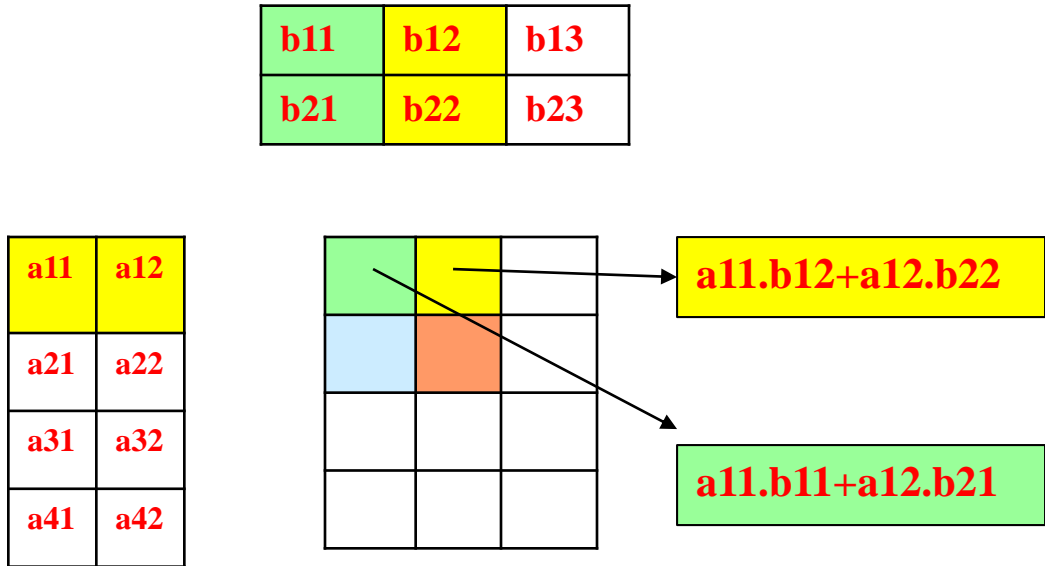
Complete code here!

```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
    int i = threadIdx.x;
    C[i] = A[i]+B[i];
}

int main(int *argc, char **argv[]) { /* run on Host */
    int size = N*sizeof(int);
    int a[N],b[N],c[N],*devA,*devB,*devC;
    cudaMalloc((void**) &devA,size);
    cudaMalloc((void**) &devB,size);
    cudaMalloc((void**) &devC,size);
    cudaMemcpy(devA, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(devB, h_B, size, cudaMemcpyHostToDevice);
    vecAdd<<<1,N>>>(devA,devB,devC);
    cudaMemcpy(C, devC, size, cudaMemcpyDeviceToHost);
    cudaFree(devA);
    cudaFree(devB);
    cudaFree(devC);
    return(0);
}
```

Example 2: Matrix Multiplication



Sequential Code:

```
For i loop:  
  For j loop:  
    For k loop:  
      compute C[i,j]
```

Order of complexities (for **n x n** matrices):

Time: $O(n^3)$

Space: $(2.n^2 + n^2) = 3n^2$

Cuda Solution approach – Computational steps for 2 x 2 case

- Load operation loads the data into a Shared Memory; A data loaded is accessible to all the threads

Iteration 0:

Thread 0: Load $\{a11, a12\}$, Compute $a11.b11, a11.b12$ (after Th2 loads the data)

Thread 1: Load $\{b11, b21\}$, Compute $a12.b21, a21.b11$ (after Th3 loads the data)

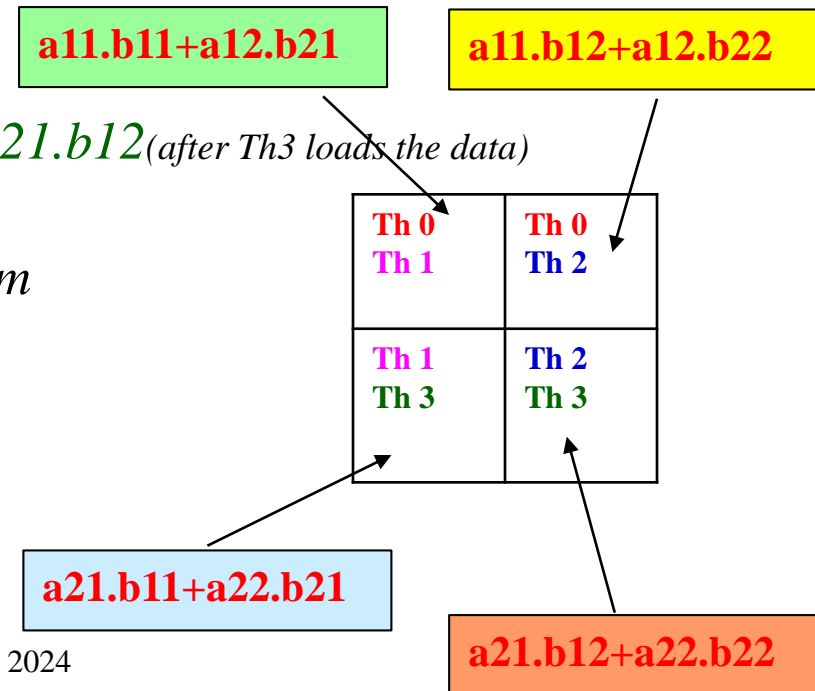
Iteration 1: (Here SM already has $\{a11, a12\}$)

Thread 2: Load $\{b12, b22\}$, Compute $a12.b22, a21.b12$ (after Th3 loads the data)

Iteration 2: (Here SM already has $\{b12, b22\}$)

Thread 3: Load $\{a21, a22\}$, Compute $a21.b21, a21.b12$ (after Th3 loads the data)

Then, in each cell, one of the threads can perform sums of the respective products



Cuda Solution alternative approaches (general case):

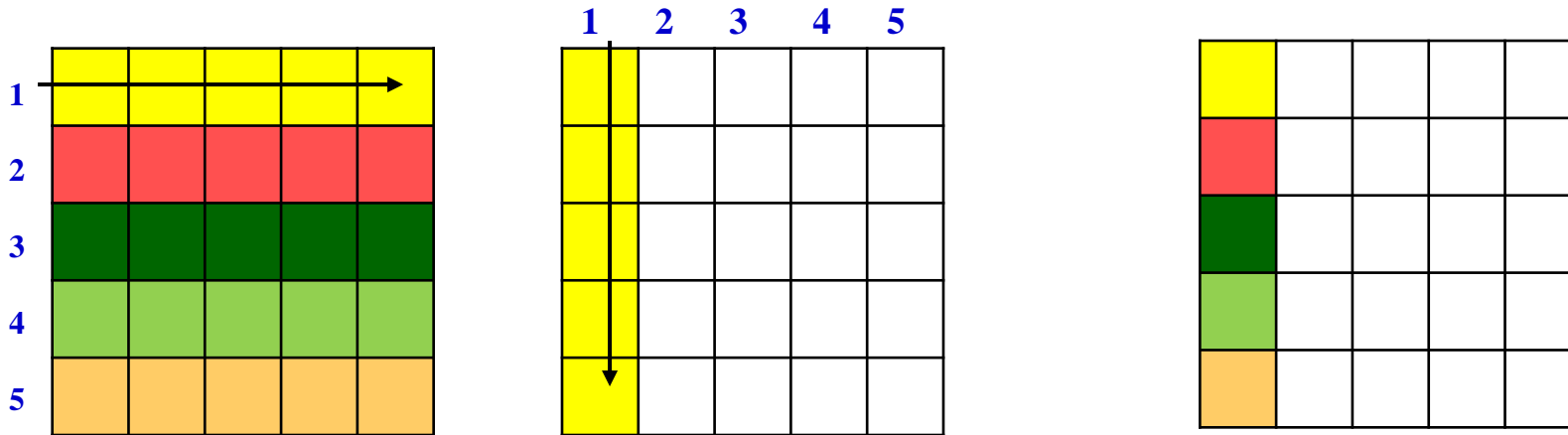
Reference: *Complete code and Step-by-step demo of code development available via:*

>> <https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/>

>> <https://debuggingsolution.blogspot.com/2021/11/matrix-multiplication-in-cuda.html>

An $n \times n$ Matrix-Matrix Multiplication – Understanding the time complexity with a given number of threads

Claim: Given n Threads, time complexity can be $\sim O(n^2)$



Total time: Shared memory loading time + Computational time

SM loading time: Time to write 1 row (or col) takes $O(n)$ time for 1 thread. A and B are loaded one after other; We have n threads; This takes $O(2n)$ time to load both A & B in the shared memory using n threads. All n threads will do loading of rows / cols concurrently.

Computational time: Each thread can compute $C_{i,j}$ and it takes $\sim O(2n)$ [n mults and n adds]. So, one entire column (n entries) takes $O(2n)$ using n independent threads. Thus, for n columns, it takes $\sim O(n^2)$. Hence, the claim.



DIY! *Some free food for thought!*

(i) *Is there any thread divergence issue in the above solution?*

(ii) *To achieve $O(n)$ complexity – how many threads are needed? What is your strategy?*

(iii) *Is $O(1)$ possible? If so, how many threads are needed and what is your strategy?*

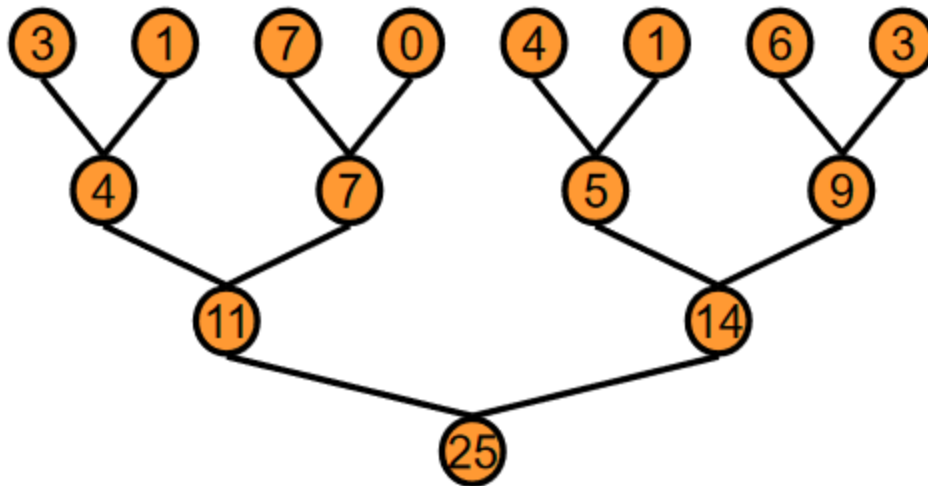
Let me know your solutions! 😊

Example 3: Parallel Reduction in Cuda

To combine an array of elements to produce a single value as a result

- Find the sum of values of an array
- Find the Max/Min in a given array

Find the sum of values of an array



*Not for your
exams!*

Reference Source: NVIDIA - Tree Based reduction within each thread block

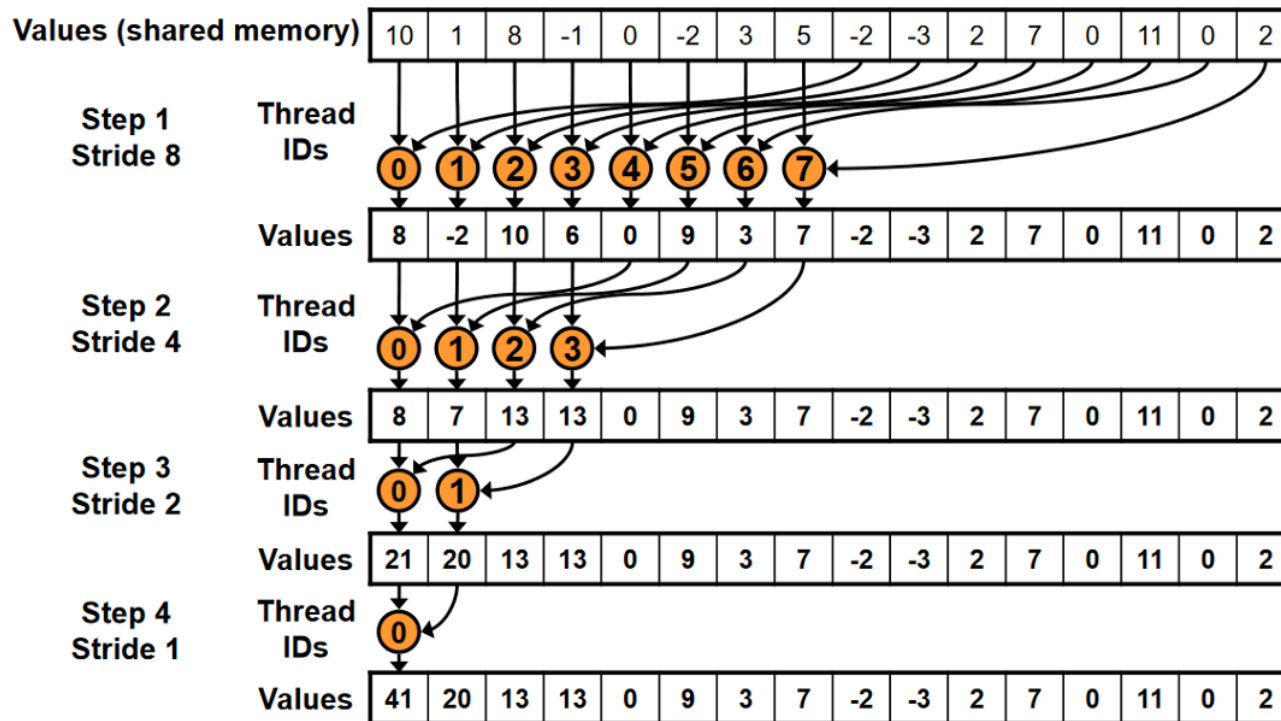
Parallel reduction - Find the sum of values of an array

Algorithm steps:

- Declare a stride length: $Stride = N/2$; # N :even
- Number of threads = stride length;
- Repeat until reduction to a single element:
Add index j & index $j+N/2$, $j=0, \dots, (N/2)-1$
 $Stride = Stride/2$; # every iteration this is done

Not for your exams!

Reference Source: NVIDIA - Strided indexing technique



Some prominent GPU programming development platforms

- CUDA - <https://developer.nvidia.com/category/zone/cuda-zone>
- OpenCL - <http://www.khronos.org/opencv/>
- OpenACC - <http://www.openacc-standard.org>
- Thrust - <http://code.google.com/p/thrust/>
- ArrayFire - <http://www.accelereyes.com/>
- C++AMP - <http://msdn.microsoft.com/enus/library/vstudio/hh265136.aspx>
- **Google Colab** - <https://programmersbranch.com/2021/12/23/free-gpu-for-your-python-code-and-deep-learning-model-with-google-colab/>

Concluding remarks

CUDA performance optimization depends on a number of factors;




We have two main time-consuming aspects:

Computation & Memory access

Major issues:

- synchronization via `__syncthreads()`;
- Overheads (communications, computations)
- Strategies to hide latency;
- Thread divergence issues;
- Memory access efficiency;

Annexure - Summary

1. Host code → C code
2. Special Cuda keywords → function or data declarations (for device)
3. Functions → rich in Data parallelism → kernels
4. Kernel launch → many threads → collection → grid (GPU parallel kernel); two level hierarchy (grid & blocks).
5. Kernel termination ---- serial code ---new kernel (grid); (overlap possible)
6. The execution of a thread is sequential (from users point).
7. Threads process different parts of data in parallel.
8. Stub function → launching a kernel → executed on the device
9. All threads in the grid execute the same kernel.
10. All blocks of a grid are of the same size.
11. Each block can contain specified max. number (1024) of threads. 
12. No. of threads in each block is specified when kernel is launched. 
13. Dimension of thread blocks should be multiple of 32 for good performance.
14. Each thread has a unique thread ID. 

GPU Programming

Compilation?

Download NVIDIA toolkit and it has the required compiler;

NVIDIA provides “nvcc” – compiler + driver

nvcc will automatically separate out the code for Host and Device; It uses:

gcc – for normal C/C++ on host;

nvcc – for the device code;