Introduction to accelerated computing

14th Inverted CERN School of Computing

Charis Kleio Koraka

Tuesday March 7th 2023

Overview

- Hardware accelerators and heterogeneous computing
- The GPU
- GPU applications in HEP
- The CUDA programming model

Hardware accelerators

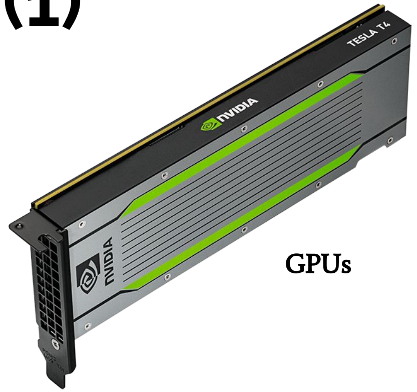
- Devices built for **executing specific tasks more efficiently** compared to running on the standard computing architecture of a CPU

- Come in many flavors :
    - GPUs / FPGAs / TPUs …

- Part of our everyday lives :
    - Encryption, video stream decoding, 3D graphics acceleration, pattern/object recognition, machine learning, AI and many more

Some types of hardware accelerators (1)

- **GPU** (Graphic Processing Unit)
  - Initially developed for graphics processing
  - Optimized for parallel processing of floating-point operations & used in a variety of tasks



GPUs

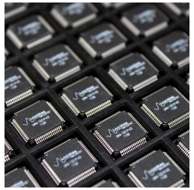
- **FPGA** (Field-Programmable Gate Array)
  - Integrated circuit (IC) configurable by the user and provides interface flexibility
  - FPGAs can be reprogrammed to suit the needs of the application or required functionality

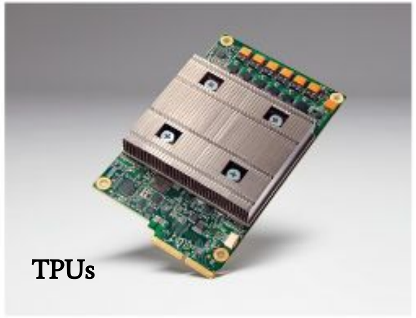
FPGAs

Image sources [i] ,[ii]

Some types of hardware accelerators (2)

- **ASIC** (Application-Specific Integrated Circuit)
  - IC chip customized for a particular use
  - i.e. lower precision and/or optimised memory usage to maximize throughput
- **TPU** (Tensor Processing Unit)
  - Optimised to perform matrix-multiplication operations / used in e.g. NN and RF training
- **VPU** (Vision Processing Unit)
  - Used to accelerate machine vision algorithms, i.e. CNNs , AI etc.

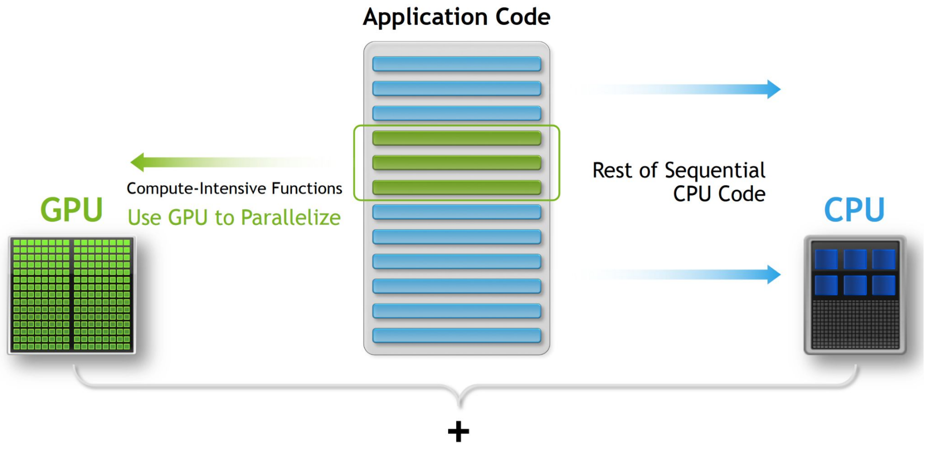

ASIC


TPUs

VPUs

Image sources [i] , [ii], [iii]

How are hardware accelerators used?



- In accelerated computing we take the compute intensive parts of the application code and parallelize that for execution on e.g. a GPU
  - Typically integer or floating-point mathematical operations
- The remainder of the code (usually the vast majority) remains on the CPU
  - The part of code that remains on the CPU is ideally serial code
- Data between the CPU and the accelerator has to be transferred:
  - This is performed via interconnect i.e. PCIe (*Peripheral Component Interconnect Express*), NVLink, Ethernet etc.

Image source [i]

Heterogeneous computing

- Heterogeneous computing involves using multiple different types of processors to accomplish a task
- Code can run on more than one platform concurrently
- A heterogeneous system can consist of :
  - Different types of CPUs (i.e. combine compute powerful with less compute powerful but more power efficient CPU cores)
  - Hardware accelerators

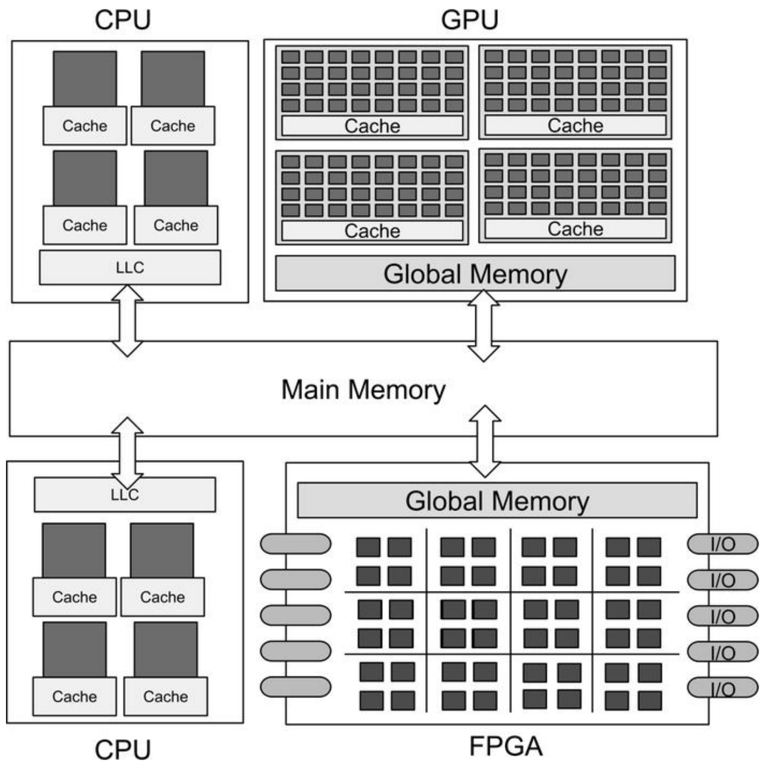


Image source [i]

The GPU

The Graphic Processing Unit (GPU)

GPUs are similar to CPUs :

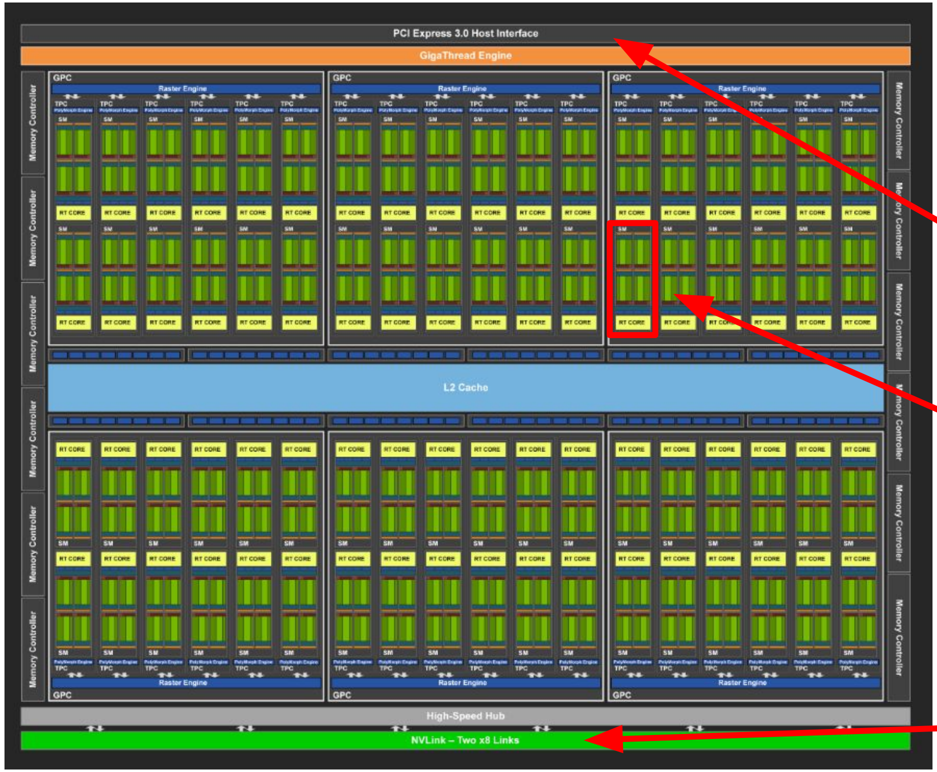
- Silicon based micro-processor that contain cores, registers, memory, and other components.

But also very different :

- **Many-core processor**
- Follows the **Single instruction, multiple threads (SIMT)** execution model
  - Asynchronous programming model where threads are not executed in lockstep
- GPU acceleration emphasizes on :
  - **High data throughput and massive parallel computing:** a GPU consist of hundreds of cores performing the same operation on multiple data items in parallel.

Image sources [i],[ii]

The NVidia GPU architecture



- The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).
- Each SM in a GPU is designed to support concurrent execution of hundreds of threads

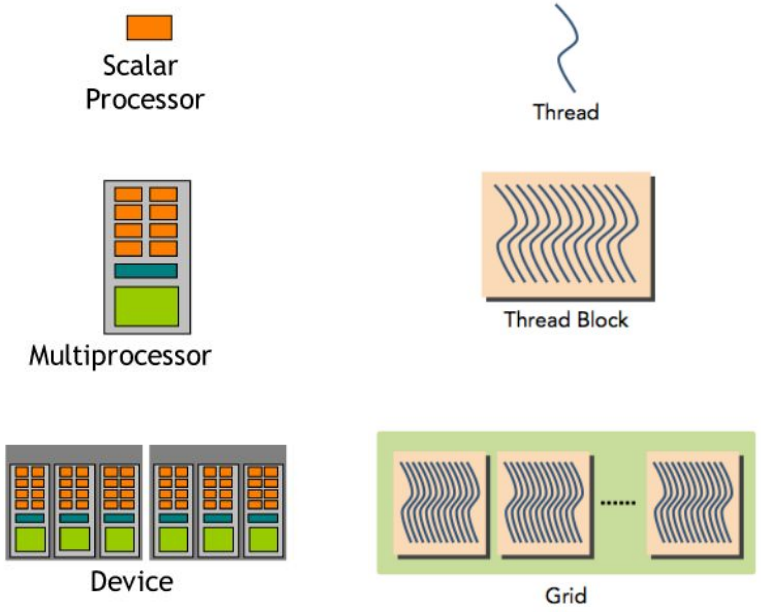
PCIe interconnect: Can be used for connecting GPU to host CPU

SM

NVlink : Can be used to connect to additional GPUs

Image source [3]

Hardware to software mapping



Hardware **Software**

- A scalar processor or CUDA core is equivalent to a software thread
- Scalar processors are grouped into a SM
- Each execution of a GPU function is done concurrently on a number of threads referred to as a thread block
- Each thread block is executed by one SM and cannot be migrated to other SMs in GPU
- The set of thread blocks executing the GPU function is called a grid.
- In CUDA terminology the GPU is referred to as the device

Image source [2]

GPU memory hierarchy

- **Registers**
  - Memory private to each thread
  - Fastest form of memory
- **L1 cache/Shared memory**
  - Fast accessible memory that can be accessed by threads in the same block and threads of different blocks in the same SM
- **Read-only**
  - Each SM has a constant/texture cache memory which is read-only to kernel code. Fast but limited in size
- **L2 Cache**
  - Memory that all threads in all blocks can access. Fast but limited in size.
- **Global memory**
  - GPUs DRAM memory
  - Slow but large

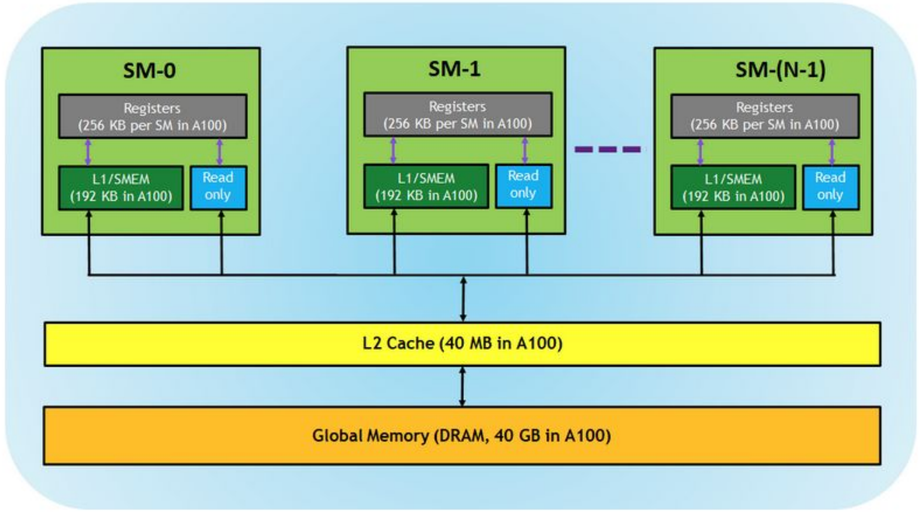
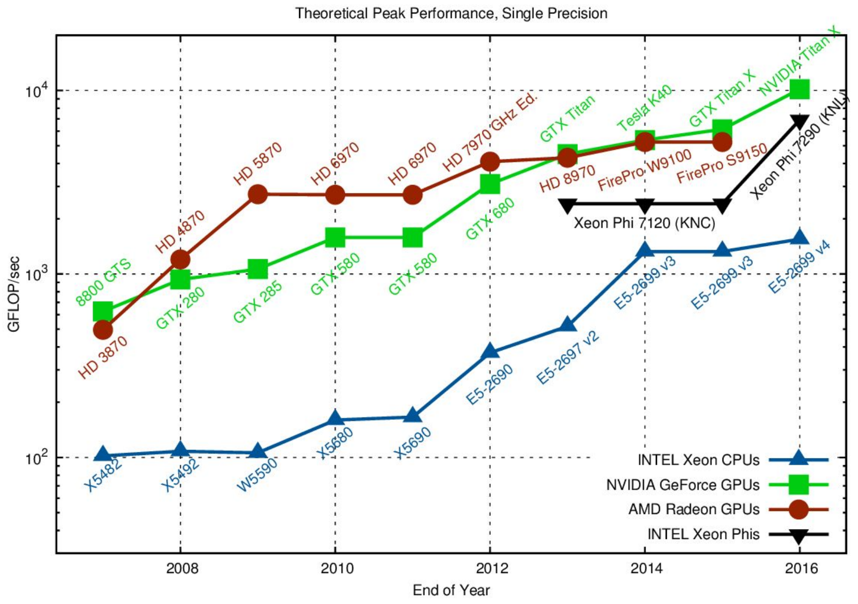


Image source [i]

Performance comparison of CPUs and GPUs (1)

FLOPS : Floating-Point Operations per Second

- Measure of computing performance useful in fields that require floating-point calculations (such as HEP)
- GPUs can deliver more FLOPS compared to CPUs



Performance comparison of CPUs and GPUs (2)

FLOPS per Watt :

- Rate of floating-point operations performed per watt of energy consumed

Important since power consumption is limiting factor in hardware manufacturing/usage:

- Peak performance constrained by the amount of power it can draw and the amount of heat it can dissipate

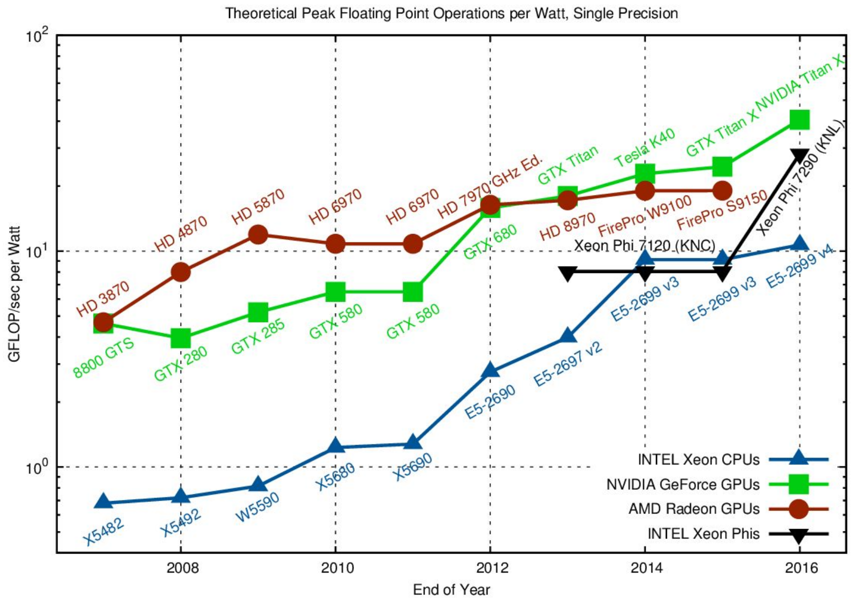
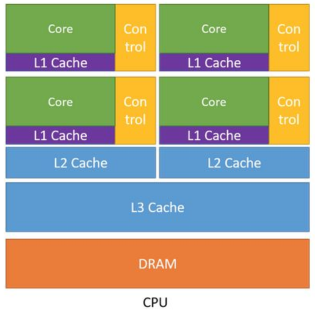


Image source [i]

CPU vs GPU - overview of main differences

CPU

- ~O(10) powerful cores
  - Larger instruction set
- Low latency
- Serial processing
- Complex operations
- Higher clock speeds



GPUs

- ~O(1000) of less powerful cores
  - Smaller instruction set
- High throughput
- Parallel processing
- Simple operations
- Better per-watt performance

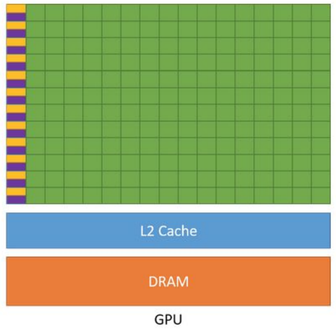


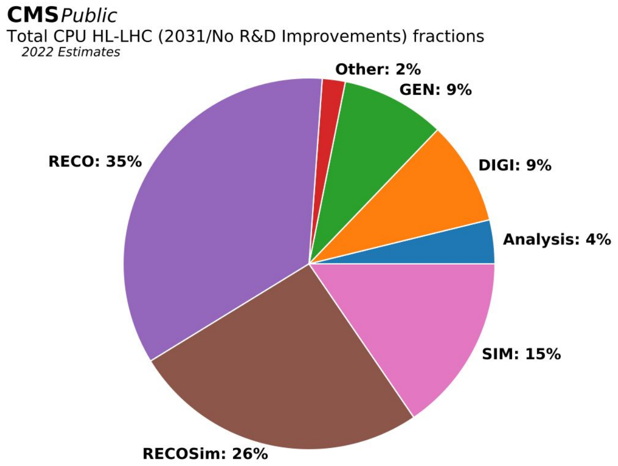
Image source [4]

Applications of GPUs in HEP

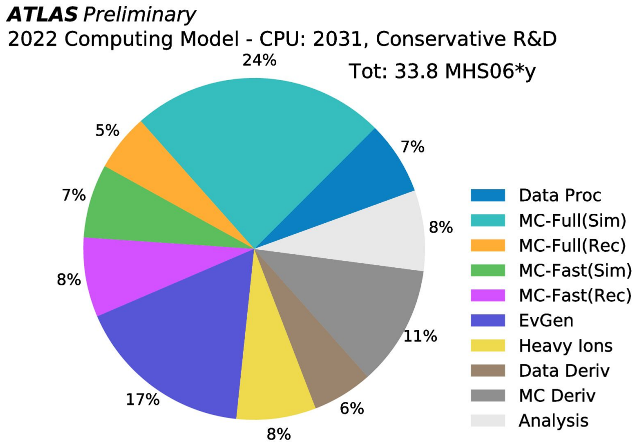
Computing needs in HEP

- Event generation
- Simulation
- Event reconstruction
- Event post-processing
- Data analysis

[link]



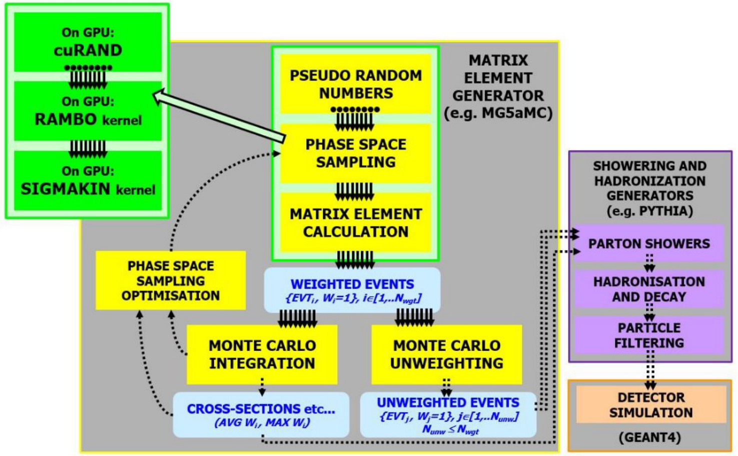
[link]



How can GPUs help?

- Event generation
- Simulation
- Event reconstruction
- Event post-processing
- Data analysis

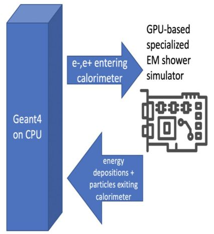
- GPU enabled event generator i.e. Madgraph [i]


On GPU: cuRAND
On GPU: RAMBO kernel
On GPU: SIGMAKIN kernel
PSEUDO RANDOM NUMBERS
PHASE SPACE SAMPLING
MATRIX ELEMENT CALCULATION
MATRIX ELEMENT GENERATOR (e.g. MG5aMC)
PHASE SPACE SAMPLING OPTIMISATION
WEIGHTED EVENTS {EVT_i, W_i≠1}, i∈[1,..N_wgt]
MONTE CARLO INTEGRATION
MONTE CARLO UNWEIGHTING
CROSS-SECTIONS etc... (AVG W_i, MAX W_i)
UNWEIGHTED EVENTS {EVT_j, W_j=1}, j∈[1,..N_unw] N_unw ≤ N_wgt
SHOWERING AND HADRONIZATION GENERATORS (e.g. PYTHIA)
PARTON SHOWERS
HADRONISATION AND DECAY
PARTICLE FILTERING
DETECTOR SIMULATION
(GEANT4)

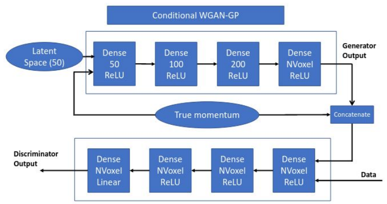
How can GPUs help?

- Event generation
- Simulation
- Event reconstruction
- Event post-processing
- Data analysis

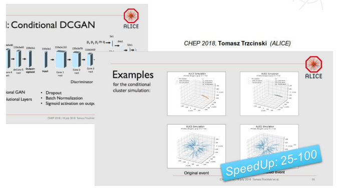
- **GPU based Geant4 application (i.e. AdePT) [i]**


Geant4 on CPU
e-,e+ entering calorimeter
GPU-based specialized EM shower simulator
energy depositions + particles exiting calorimeter

- **AI/ML enabled Fast Simulation (i.e. AltFast3 in ATLAS [ii] , DC-GAN in ALICE [iii])**



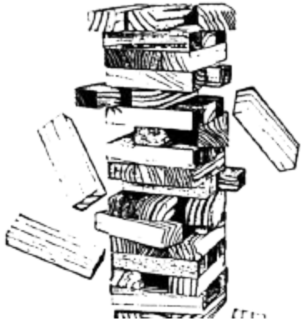
Deep Conditional Convolutional GAN


Conditional DCGAN
ALICE
CHEP 2018, Tomasz Trzcinski (ALICE)
Examples for the conditional cluster simulation:
SpeedUp: 25-100

How can GPUs help?

- Event generation
- Simulation
- Event reconstruction
- Event post-processing
- Data analysis

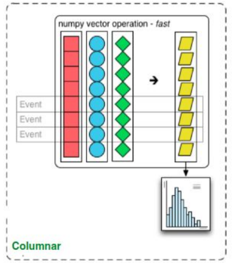
- **Track reconstruction, primary vertex reconstruction, raw data unpacking, clustering etc.**
- **Various efforts in different experiments (Patatrack track reconstruction [i], Allen project [ii], ALICE TPC track reconstruction [iii] etc.)**



How can GPUs help?

- Event generation
- Simulation
- Event reconstruction
- Event post-processing
- Data analysis

- **Training and inference of ML models**
- **Perform HEP analysis using columinar analysis paradigm tools (i.e. coffea [i])**



Introduction to CUDA

The CUDA programming model

CUDA → **C**ompute **U**nified **D**evice **A**rchitecture.

- It is an extension of C/C++ programming
- Developed by Nvidia and is used to develop applications executed on NVidia GPUs

To execute any CUDA program, there are three main steps:

- Copy the input data from CPU or host memory to the device memory
- Execute the CUDA program
- Copy the results from device memory to host memory

1. Copy data for host to device

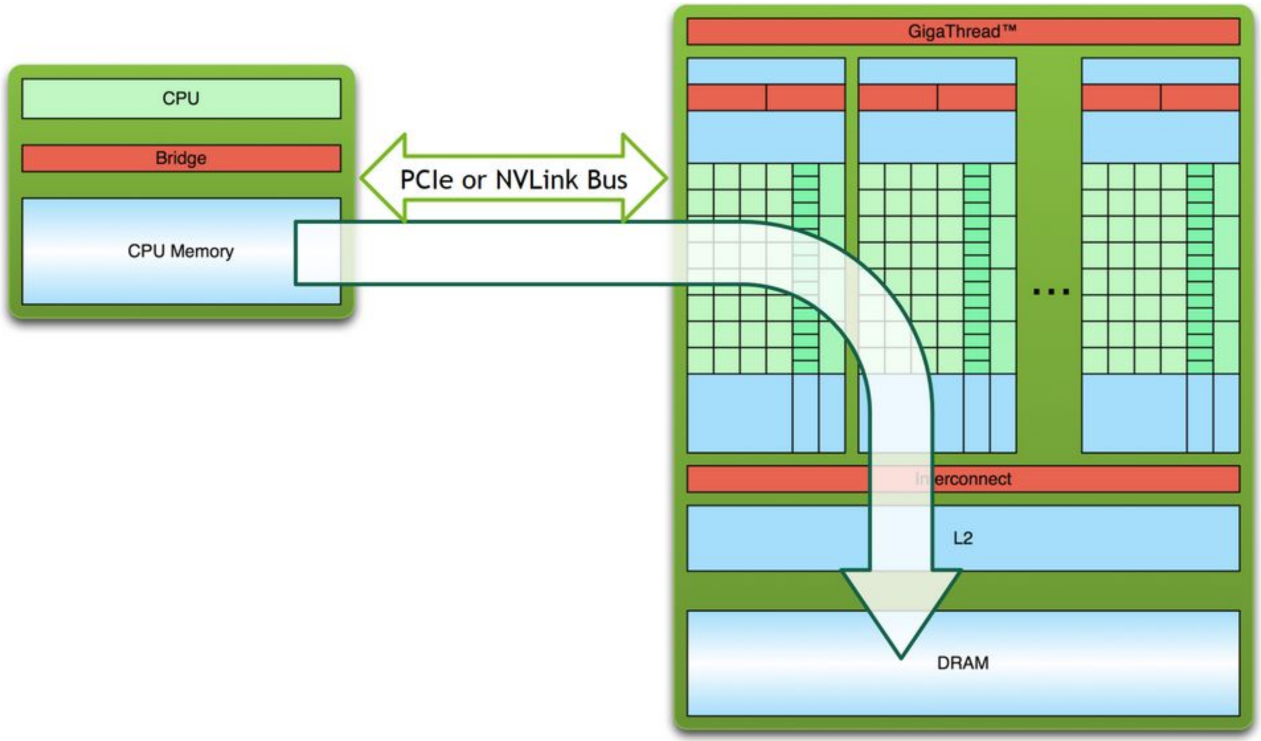


Image source [1]

2. Execute the CUDA program

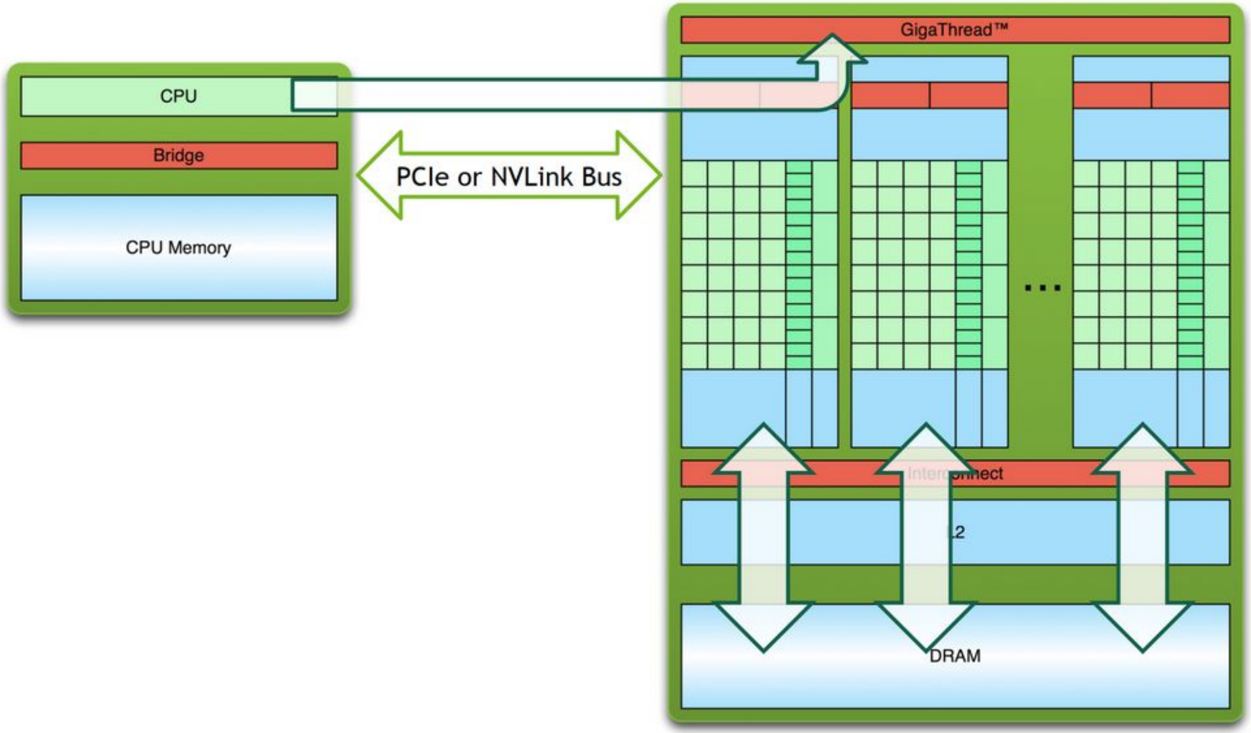


Image source [1]

3. Copy data from device back to host

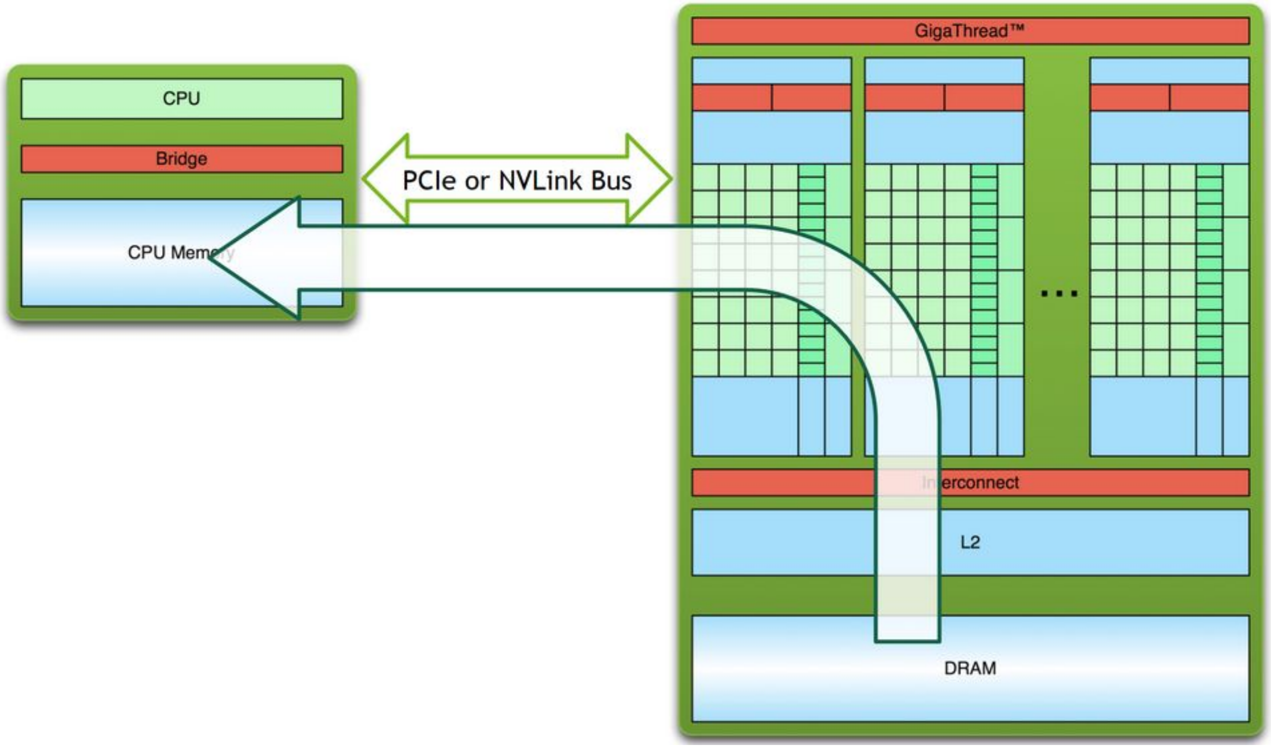
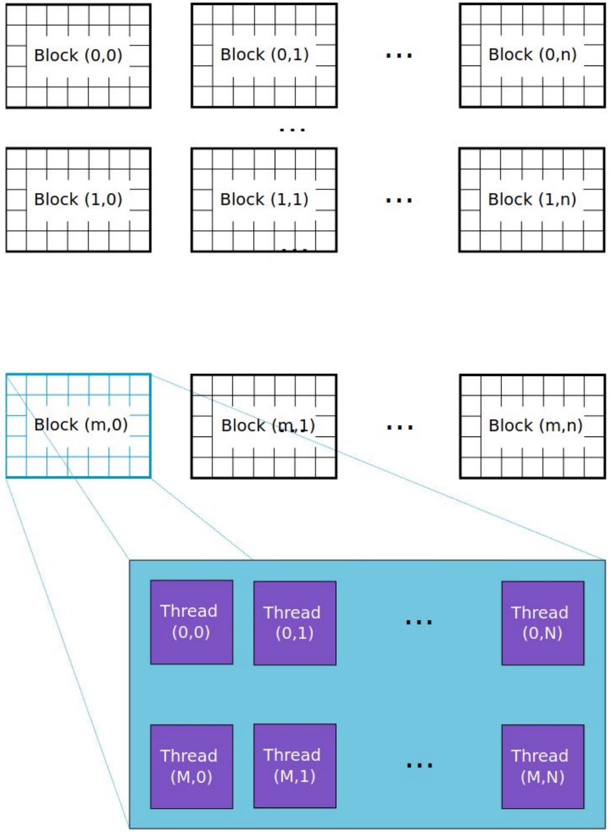


Image source [1]

Threads & blocks

- In CUDA, built-in variables are available in order to express threads and blocks :
  - `threadIdx & blockIdx`
- The variables have 3-dimensional indexing & provide a natural way to express elements in vectors and matrices :
  - `threadIdx.x , threadIdx.y threadIdx.z`
- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).
- The dimension of the thread block is accessible within the kernel through the built-in **`blockDim`** variable.



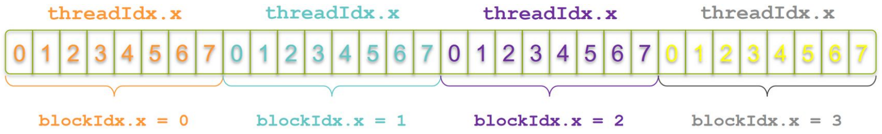
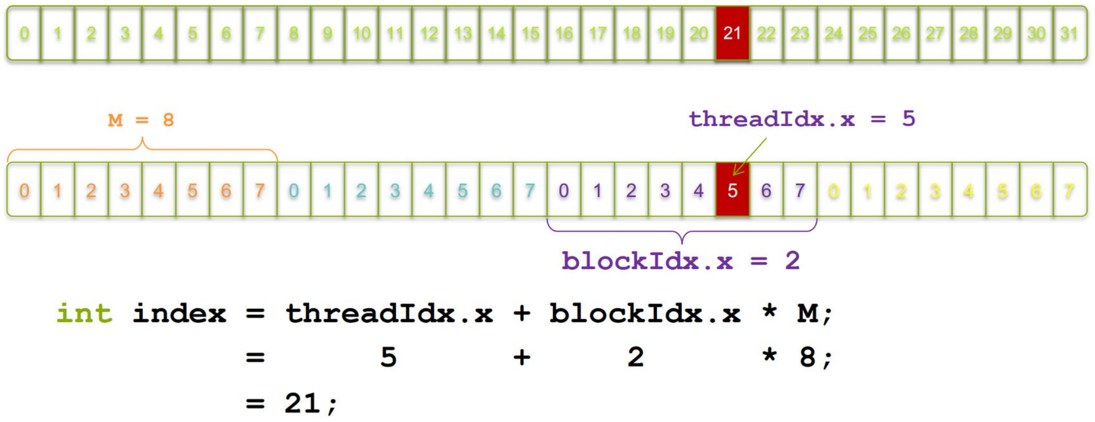


Image source [1],[2]

Indexing using blockIdx and threadIdx

- The `threadIdx & blockIdx` variables can be used to express the unique index of an element in an array/matrix etc.
- Assuming that each block consists of a number of M threads :
  - `index = threadIdx.x + blockIdx.x * M;`

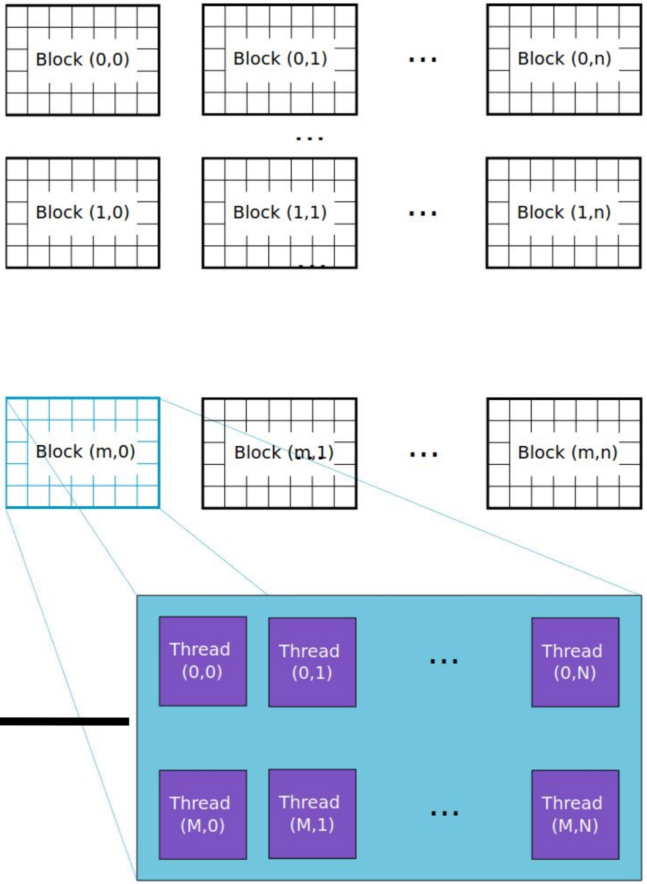


```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2      * 8;
          = 21;
```

Image source [1]

Warps

- Within a thread block, threads are executed in groups → **Warps**
- A warp is an entity of 32 threads on Nvidia GPUs
- If the block size is not divisible by 32, some of the threads in the last warp will remain idle :
  - block size should be chosen to be a multiple of the warp size
- Threads in the same warp are processed simultaneously



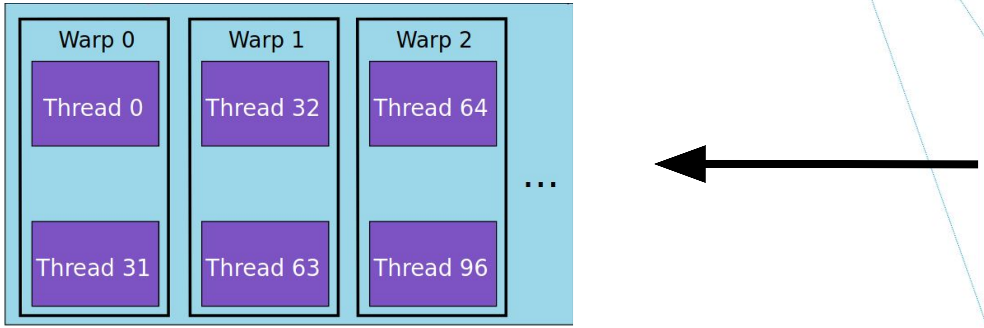


Image source [2]

CUDA kernel

- CUDA kernel is a function that gets executed on the GPU
- The kernel expresses the portion of the application that is parallelizable
    - It will be executed multiple times in parallel by different CUDA threads

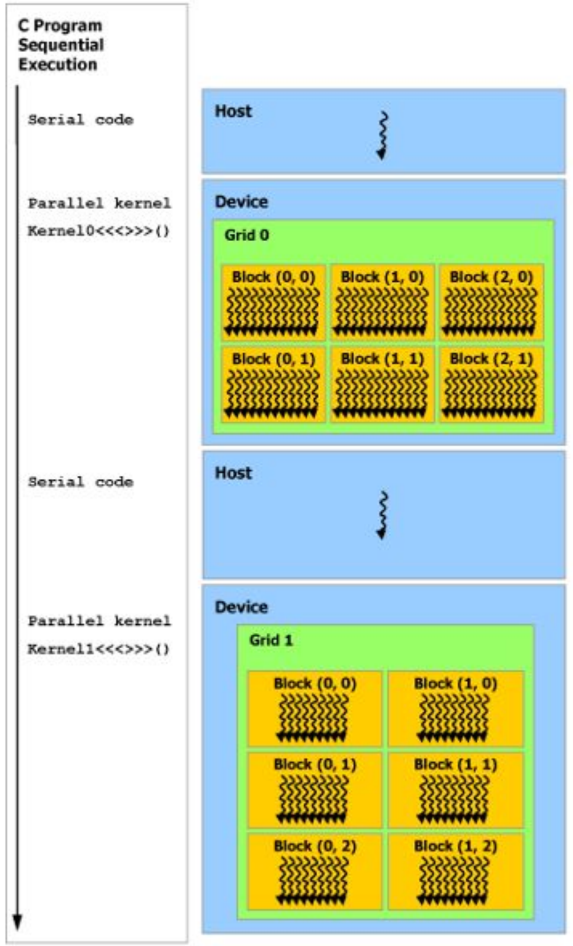


Image source [4]

CUDA function declarations

Declaration	Callable from:	Executed on:
_ _global_ _	host	device
_ _device_ _	device	device
_ _host_ _	host	host

- **_ _global_ _** keyword defines a kernel function:
  - Is launched by host and executed on the device
  - Must return void
- **_ _device_ _** and **_ _host_ _** can be used together
- **_ _host_ _** declaration, if used alone, can be omitted

Launching a CUDA kernel

- Let’s assume we have the following kernel :

```
__global__ void mykernel() {
     …Do something…
}
```

- How do we launch it?

This is the grid dimension i.e. the number of blocks that will be launched

This is the block dimension i.e. the number of threads within a block

```
myKernel<<<nBlocks,nThreads>>>();
```

- The above command will launch the kernel with `nBlocks`, each of which has `nThreads`.
- The kernel is executed multiple times concurrently by different threads
- The total number of invocations of the kernel body is now `nBlocks` * `nThreads`.

Memory management

- The host and device have their own separate memory:
  - Device pointers point to GPU memory
  - Host pointers point to CPU memory
- CUDA kernels operate out of device memory
- CUDA provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory :

```
cudaMalloc(&ptr, size_in_bytes_to_allocate)
```

```
cudaFree(ptr)
```

```
cudaMemcpy(destination_ptr,source_ptr, size_in_bytes, direction)
```

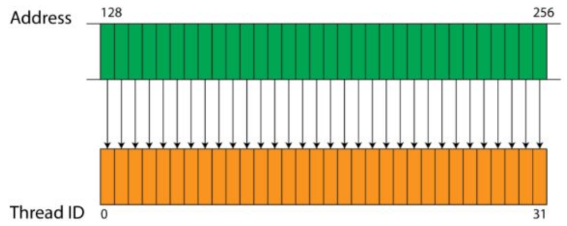
where direction can be :

- For copying data from CPU to GPU
- For copying data from GPU to CPU

```
int* a;
int* d_a;
// Host copy of variable a
a = (int*) malloc(sizeof(int));
// Device copy of variable a
cudaMalloc(&d_a, sizeof(int));
// Set the host value of a
*a = 1;
// Copy the value of a to the device
cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
// Launch the kernel to set the value
do_something<<<1,1>>>(d_a);
cudaDeviceSynchronize();
// Copy the value of a back to the host
cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
// Free the allocated memory
free(a);
cudaFree(d_a);
```

Coalesced global memory access

- Global memory loads and stores data in as few as possible transactions → coalesced memory access
- Important performance consideration as it can affect the time needed to access data
- Every successive 128 bytes (DRAM burst) can be accessed by a warp
- If the data accessed by the threads in a warp are not in the same burst section, the data access will take twice as long



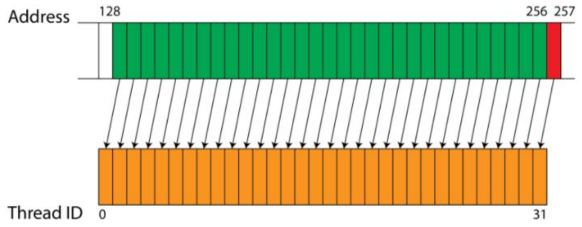


Image source [i]

Putting together a CUDA program

The main components of a CUDA program are :

- Declarations of functions :
  - These can be __host__ / __global__ / __device__ functions
- Copying data to/from host :
  - Use cudaMalloc / cudaMemcpy / cudaFree
- Kernel launch <<<grid size, block size >>>(<arguments>)
- Concurrency management
  - Use __syncthreads() or CudaDeviceSynchronize()

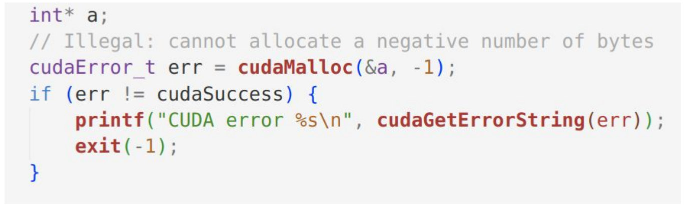
```
__global__ void do_something (int* a) {
    *a = 2;
}

int main() {

    int* a;
    int* d_a;
    // Host copy of variable a
    a = (int*) malloc(sizeof(int));
    // Device copy of variable a
    cudaMalloc(&d_a, sizeof(int));
    // Set the host value of a
    *a = 1;
    // Copy the value of a to the device
    cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
    // Launch the kernel to set the value
    do_something<<<1,1>>>(d_a);
    cudaDeviceSynchronize();
    // Copy the value of a back to the host
    cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
    // Free the allocated memory
    free(a);
    cudaFree(d_a);

}
```

Error handling

- Error codes can be converted to a human-readable error messages with the following CUDA run- time function:

```
char* cudaGetErrorString(cudaError_t error)
```

- A common practice is to wrap CUDA calls in utility functions that manage the error returned :



```
int* a;
// Illegal: cannot allocate a negative number of bytes
cudaError_t err = cudaMalloc(&a, -1);
if (err != cudaSuccess) {
    printf("CUDA error %s\n", cudaGetErrorString(err));
    exit(-1);
}
```

- To detect errors in a kernel launch, we can use the API call **cudaGetLastError()** which returns the error code for whatever the last CUDA API call was.

```
cudaError_t err = cudaGetLastError();
```

- For errors that occurs asynchronously during the kernel launch, **cudaDeviceSynchronize()** has to be invoked after the kernel in order to return any errors associated with the kernel launch.

Compilation

- Compiling a CUDA program is similar to compiling a C/C++ program.
- Cuda code should be typically stored in a file with extension .cu
- NVIDIA provides a CUDA compiler called **nvcc** :
  - nvcc is called for CUDA parts
  - gcc is called for c++ parts
  - nvcc converts .cu files into C++ for the host system and CUDA assembly or binary instructions for the device
- Usage :

```
nvcc myCudaProgram.cu -o myCudaProgram
```

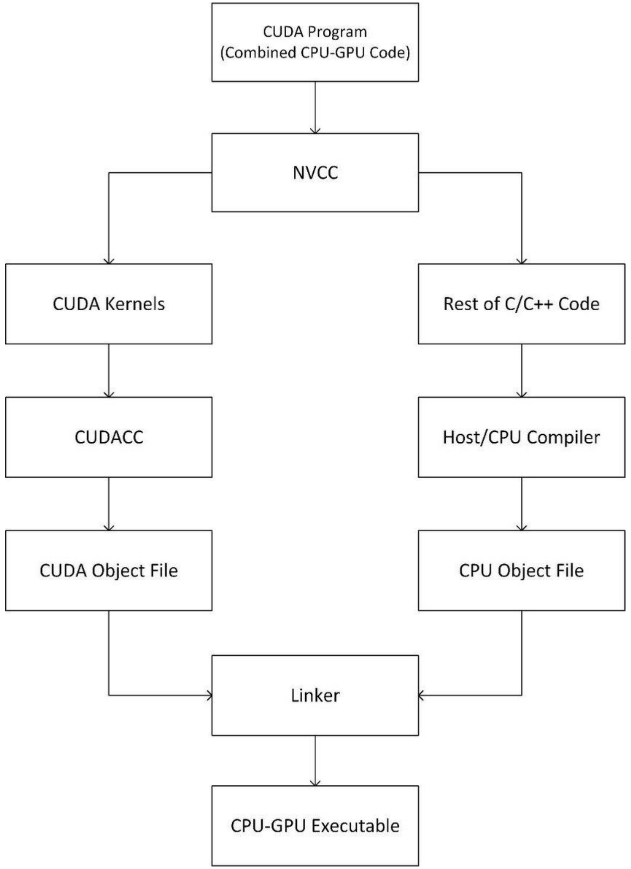


Image source [i]

Profiling (1)

- Similarly to CPU code, GPU code can be profiled
- Goal of profiling is to identify and optimise performance limiters.
- Common reasons for limited performance include :
    - Portions of the code that run serially on the CPU
    - Memory copies for host to device
    - Latency of launching GPU kernels
    - Uncoalesced memory accesses, lack of cache reuse, not using shared memory, register spilling etc.
    - Low arithmetic intensity (operations computed per byte accessed from memory)

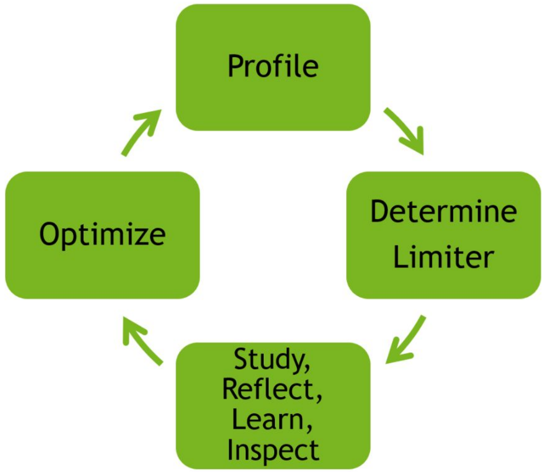
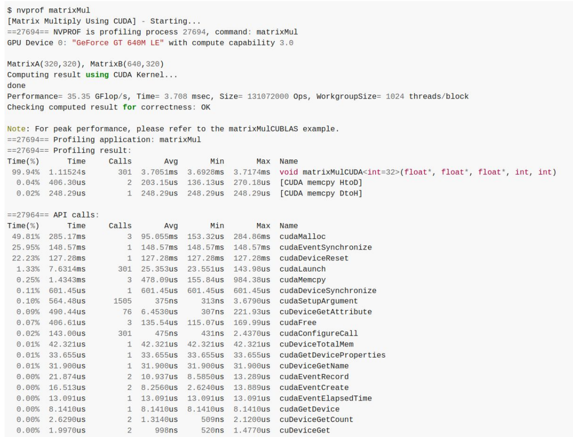


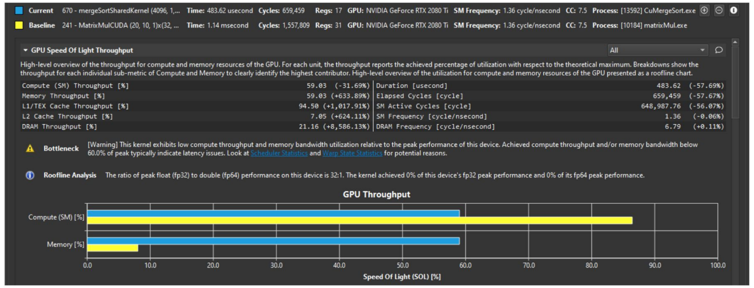
Image source [1]

Profiling (2)

- Many profiling tools exist. Some commonly used ones include
- **nsys** :
  - Command line profiler for CUDA applications
  - Results can be saved for later viewing by the Visual Profiler.
- **nvvp / ncu** :
  - Nvidia Visual Profiler, Nsight Compute
  - Interactive kernel profilers for CUDA applications.
  - Provide detailed performance metrics and API debugging via a user interface and command line tool.



```
$ nvprof matrixMul
[Matrix Multiply Using CUDA] - Starting...
==27694== NVPROF is profiling process 27694, command: matrixMul
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: OK

Note: For peak performance, please refer to the matrixMulCUBLAS example.
==27694== Profiling application: matrixMul
==27694== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.94%  1.11524s       301  3.7051ms  3.6928ms  3.7174ms  void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
  0.04%  406.30us         2  203.15us  136.13us  270.18us  [CUDA memcpy HtoD]
  0.02%  248.29us         1  248.29us  248.29us  248.29us  [CUDA memcpy DtoH]

==27964== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.81%  285.17ms         3  95.055ms  153.32us  284.86ms  cudaMalloc
 25.95%  148.57ms         1  148.57ms  148.57ms  148.57ms  cudaEventSynchronize
 22.23%  127.28ms         1  127.28ms  127.28ms  127.28ms  cudaDeviceReset
  1.33%  7.6314ms       301  25.353us  23.551us  143.98us  cudaLaunch
  0.25%  1.4343ms         3  478.09us  155.84us  984.38us  cudaMemcpy
  0.11%  601.45us         1  601.45us  601.45us  601.45us  cudaDeviceSynchronize
  0.10%  564.48us      1505     375ns     313ns  3.6790us  cudaSetupArgument
  0.09%  490.44us        76  6.4530us     307ns  221.93us  cuDeviceGetAttribute
  0.07%  406.61us         3  135.54us  115.07us  169.99us  cudaFree
  0.02%  143.00us       301     475ns     431ns  2.4370us  cudaConfigureCall
  0.01%  42.321us         1  42.321us  42.321us  42.321us  cuDeviceTotalMem
  0.01%  33.655us         1  33.655us  33.655us  33.655us  cudaGetDeviceProperties
  0.01%  31.900us         1  31.900us  31.900us  31.900us  cuDeviceGetName
  0.00%  21.874us         2  10.937us  8.5850us  13.289us  cudaEventRecord
  0.00%  16.513us         2  8.2560us  2.6240us  13.889us  cudaEventCreate
  0.00%  13.091us         1  13.091us  13.091us  13.091us  cudaEventElapsedTime
  0.00%  8.1410us         1  8.1410us  8.1410us  8.1410us  cudaGetDevice
  0.00%  2.6290us         2  1.3140us     509ns  2.1200us  cuDeviceGetCount
  0.00%  1.9970us         2     998ns     520ns  1.4770us  cuDeviceGet
```



🟦 **Current** 670 - mergeSortSharedKernel (4096, 1,... **Time:** 483.62 usecond **Cycles:** 659,459 **Regs:** 17 **GPU:** NVIDIA GeForce RTX 2080 Ti **SM Frequency:** 1.36 cycle/nsecond **CC:** 7.5 **Process:** [13592] CuMergeSort.exe

🟨 **Baseline** 241 - MatrixMulCUDA (20, 10, 1)x(32, ... **Time:** 1.14 msecond **Cycles:** 1,557,809 **Regs:** 31 **GPU:** NVIDIA GeForce RTX 2080 Ti **SM Frequency:** 1.36 cycle/nsecond **CC:** 7.5 **Process:** [10184] matrixMul.exe

GPU Speed Of Light Throughput All

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

```
Compute (SM) Throughput [%]        59.03  (-31.69%) | Duration [usecond]                 483.62  (-57.69%)
Memory Throughput [%]              59.03 (+633.89%) | Elapsed Cycles [cycle]             659,459  (-57.67%)
L1/TEX Cache Throughput [%]   94.50 (+1,017.91%) | SM Active Cycles [cycle]       648,987.76  (-56.07%)
L2 Cache Throughput [%]             7.05 (+624.11%) | SM Frequency [cycle/nsecond]          1.36   (-0.06%)
DRAM Throughput [%]          21.16 (+8,586.13%) | DRAM Frequency [cycle/nsecond]        6.79   (+0.11%)
```

Bottleneck [Warning] This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance.

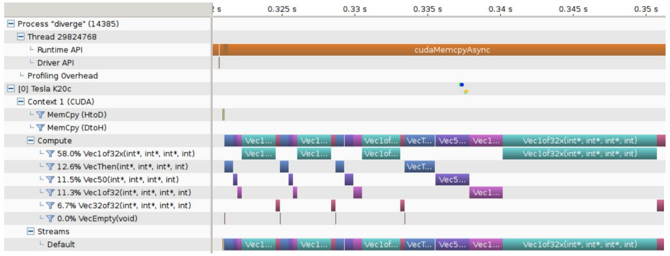

s
0.325 s
0.33 s
0.335 s
0.34 s
0.345 s
0.35 s
Process "diverge" (14385)
Thread 29824768
Runtime API
cudaMemcpyAsync
Driver API
Profiling Overhead
[0] Tesla K20c
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
58.0% Vec1of32x(int*, int*, int*, int)
12.6% VecThen(int*, int*, int*, int)
11.5% Vec50(int*, int*, int*, int)
11.3% Vec1of32(int*, int*, int*, int)
6.7% Vec32of32(int*, int*, int*, int)
0.0% VecEmpty(void)
Streams
Default
Vec1of32x(int*, int*, int*, int)

Image source [6]

Wrapping-up

Summary

- Hardware accelerators are a part of everyday life and are used in heterogeneous computing systems
- GPUs emphasize on high data throughput and massive parallel computing
- GPUs have made their way into HEP and are used for many applications
- The CUDA programming model :
    - Extension of C/C++ programming developed by Nvidia and used for applications executed on Nvidia GPUs
    - CPU and GPU system are referred to as host and device respectively.
        - The host and device have their own separate memory
    - Typically, we run serial workload on the CPU and offload parallel computation to the GPUs
        - CUDA threads are used to execute work in parallel
    - Basic CUDA syntax:
        - __global__ function declaration (kernel) is called from the host and executed on the device
        - Memory management can be performed using cudaMalloc(), cudaFree() & cudaMemcpy()
        - To launch a CUDA kernel with N blocks and M threads/block syntax is <<<N,M>>>()

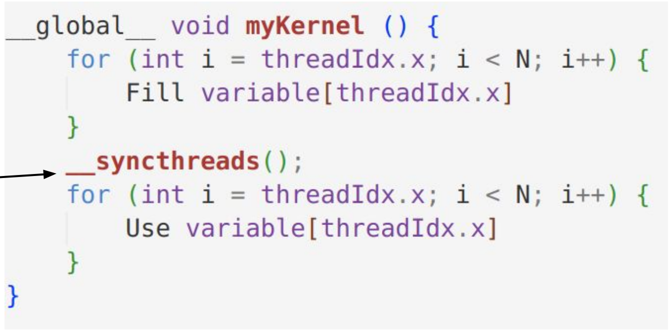
Back-up

Resources

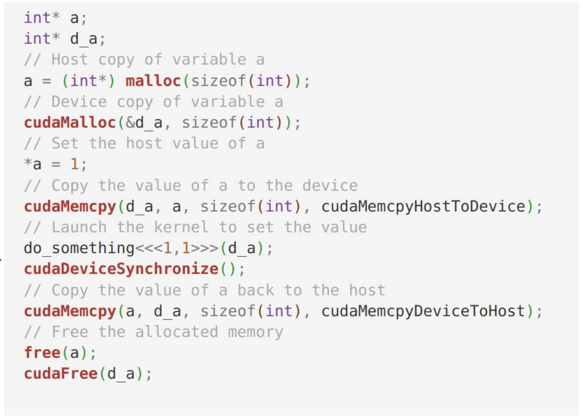
1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link

Thread synchronization

- A kernel call is asynchronous with respect to the host thread :
  - After a kernel is invoked, the program returns to the host side and continues execution.
- There are two levels of synchronization
  - **Block level**
  - **Grid level**
- To synchronize threads within one block :
  - Call `__syncthreads()` within the kernel code
- To synchronize threads at grid level
  - Call to `CudaDeviceSynchronize()` from host code.
  - Program waits until all work launched on the device has finished.



```
__global__ void myKernel () {
    for (int i = threadIdx.x; i < N; i++) {
        Fill variable[threadIdx.x]
    }
    __syncthreads();
    for (int i = threadIdx.x; i < N; i++) {
        Use variable[threadIdx.x]
    }
}
```



```
int* a;
int* d_a;
// Host copy of variable a
a = (int*) malloc(sizeof(int));
// Device copy of variable a
cudaMalloc(&d_a, sizeof(int));
// Set the host value of a
*a = 1;
// Copy the value of a to the device
cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
// Launch the kernel to set the value
do_something<<<1,1>>>(d_a);
cudaDeviceSynchronize();
// Copy the value of a back to the host
cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
// Free the allocated memory
free(a);
cudaFree(d_a);
```