Introduction to GPU Computing and CUDA

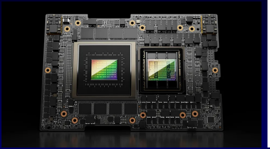
Chik Him (Ricky) Wong

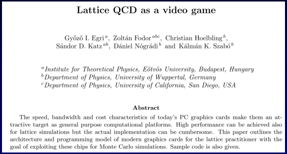
University of Wuppertal

Aug 15, 2024

Outline

What is a GPU?





Lattice QCD as a video game

Győző I. Egri[a], Zoltán Fodor[abc], Christian Hoelbling[b], Sándor D. Katz[ab], Dániel Nógrádi[b] and Kálmán K. Szabó[b]

[a] *Institute for Theoretical Physics, Eötvös University, Budapest, Hungary*
[b] *Department of Physics, University of Wuppertal, Germany*
[c] *Department of Physics, University of California, San Diego, USA*

Abstract

The speed, bandwidth and cost characteristics of today's PC graphics cards make them an attractive target as general purpose computational platforms. High performance can be achieved also for lattice simulations but the actual implementation can be cumbersome. This paper outlines the architecture and programming model of modern graphics cards for the lattice practitioner with the goal of exploiting these chips for Monte Carlo simulations. Sample code is also given.

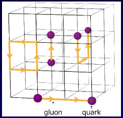
- GPU: Graphics Processing Unit
- Originally designed for rendering graphics
- Now used for general-purpose computing (GPGPU)
- Highly efficient for parallel computations

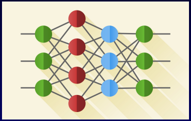
Applications of GPU Computing

- GPU has been widely applied in Research, way before the hype of Neural Networks
- All researches that utilize supercomputing involve:
    - Solving Differential Equations numerically on a field
      $\Rightarrow$ Discretization of differentiations on fields translates into Matrix operations on high-dimensional domains
    - Statistical analysis of Huge datasets
- Both of these tasks are highly parallelizable
- GPU is highly optimized for massively parallelized compuations on huge datasets
  $\Rightarrow$ If parallelizable in CPUs, GPUs can do better
- In the era of Machine learning and Neural Networks
    - Both simulations and data analysis accelerate due to algorithmic advancement
    - GPUs, being the most efficient hardware for Neural Network implementation, become very popular

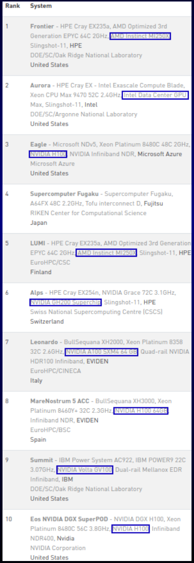
Applications of GPU Computing

https://www.top500.org/

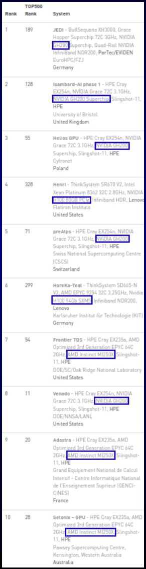




- Popular applications include:
  - Computer graphics and visualization
  - Deep learning and machine learning
  - **Scientific Computing (simulations, data analysis)**
- Applied in 9 of the top 10 fastest facilities
- Applied in all of the top 10 most energy efficient facilities



Rank	System
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** DOE/SC/Oak Ridge National Laboratory **United States**
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, **Intel** DOE/SC/Argonne National Laboratory **United States**
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, **Microsoft Azure** Microsoft Azure **United States**
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, **Fujitsu** RIKEN Center for Computational Science **Japan**
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** EuroHPC/CSC **Finland**
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** Swiss National Supercomputing Centre (CSCS) **Switzerland**
7	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, **EVIDEN** EuroHPC/CINECA **Italy**
8	**MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, **EVIDEN** EuroHPC/BSC **Spain**
9	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, **IBM** DOE/SC/Oak Ridge National Laboratory **United States**
10	**Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, **Nvidia** NVIDIA Corporation **United States**



Rank	TOP500 Rank	System
1	189	**JEDI** - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, **ParTec/EVIDEN** EuroHPC/FZJ **Germany**
2	128	**Isambard-AI phase 1** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** University of Bristol **United Kingdom**
3	55	**Helios GPU** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** Cyfronet **Poland**
4	328	**Henri** - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, **Lenovo** Flatiron Institute **United States**
5	71	**preAlps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** Swiss National Supercomputing Centre (CSCS) **Switzerland**
6	299	**HoreKa-Teal** - ThinkSystem SD665-N V3, AMD EPYC 9354 32C 3.25GHz, Nvidia H100 94Gb SXM5, Infiniband NDR200, **Lenovo** Karlsruher Institut für Technologie (KIT) **Germany**
7	54	**Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** DOE/SC/Oak Ridge National Laboratory **United States**
8	11	**Venado** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, **HPE** DOE/NNSA/LANL **United States**
9	20	**Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) **France**
10	28	**Setonix – GPU** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, **HPE** Pawsey Supercomputing Centre, Kensington, Western Australia **Australia**

Applications of GPU Computing

- Large-scale simulation entirely on GPUs is too expensive ⇒ Hybrid setup with GPUs as Accelerators :
  - **GPU : Critical, Expensive, Parallelizable computations**
  - CPU : Simple, Cheap, Non-parallelizable computations

	CPU	GPU
Number of cores	4-64	$O(10^3)$
Control logics	Complex	Simple
Cache size	Large	Small
Optimal processings	Serial	Parallel
Languages / API	Fortran, C, C++, OpenMP, MPI, Python, Matlab $\cdots$	**CUDA (NVIDIA)**, OpenCL (Cross-platform), DirectCompute (Microsoft), OpenACC (Directive-based), ROCm (AMD) $\cdots$

- CPU parallelizations (e.g. OpenMP or MPI) are combined with GPU parallelization
- CUDA: Compute Unified Device Architecture, a parallel computing platform and API model created by NVIDIA
- Terminologies used in this talk are based on CUDA

Types of Parallelism

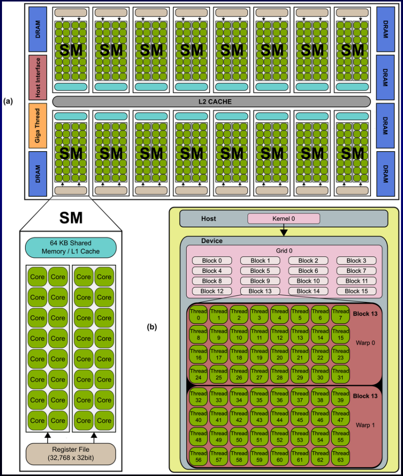
- Data Parallelism: (Most common)
  - Single Instruction Multiple Data (SIMD)
  - Example: matrix multiplication on a vector field (Lattice QCD)
  - GPU: Single Instruction Multiple Thread (SIMT), each thread handles different data
- Task Parallelism:
  - Multiple Instruction Multiple Data (MIMD)
  - Example: Evolution of particle-plasma systems
  - GPU architecture is not optimal for this, although possible via CUDA streams
- Hybrid Parallelism:
  - Parallelization in data and task
  - Example: Evolution of Domain-decomposed systems
  - Can leverage both CPU and GPU effectively:
    GPUs handle critical tasks and CPUs handle cheap tasks concurrently

GPU Architecture: Basic Components

- Architecture:
  - Streaming Multiprocessors (SMs)
  - Stream Processor (SP) Cores
  - Warp Schedulers
  - Memory Hierarchy: Global Memory, Shared Memory, Memory, Registers, L1 and L2 Cache etc
- API:
  - Host: CPU , Device: GPU
  - Kernel: Functions executed on GPU
  - Multiple Kernel calls possible, simultaneously or not
  - Thread < Warp < Block < Grid

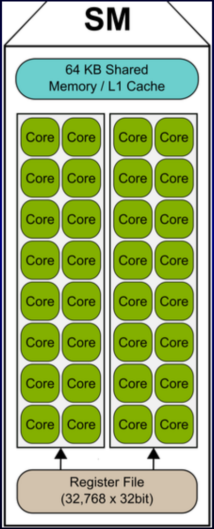
Moises Hernandez et al, 2013

https://doi.org/10.1371/journal.pone.0061892.g001



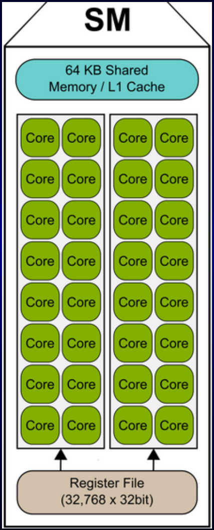
Streaming Multiprocessors (SMs)

- Building blocks of GPU
- Contains multiple CUDA cores
- Manages thread execution
- Includes warp schedulers, register file, shared memory
- Number of SMs varies by GPU model


SM
64 KB Shared Memory / L1 Cache
Core
Register File (32,768 x 32bit)

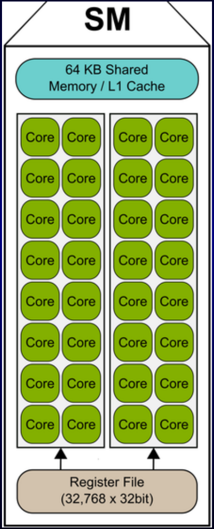
CUDA Cores

- Basic computational units in GPU
- Executes floating-point and integer operations
- Thousands of CUDA cores in modern GPUs
- Organized into groups within SMs
- Operate in SIMT (Single Instruction, Multiple Thread) fashion


SM
64 KB Shared Memory / L1 Cache
Core
Register File (32,768 x 32bit)

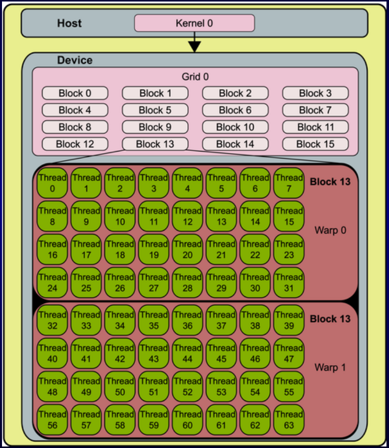
Memory Hierarchy in GPUs

- Global Memory: Large, high latency, accessible by all threads
- Shared Memory: Fast, limited size, shared within a thread block
- Registers: Fastest, very limited, per-thread storage
- L1 and L2 Cache: Automatic caching for global memory accesses
- Constant Memory: Read-only, cached, for constant data
- Texture Memory: Optimized for 2D spatial locality


SM
64 KB Shared Memory / L1 Cache
Core
Register File (32,768 x 32bit)

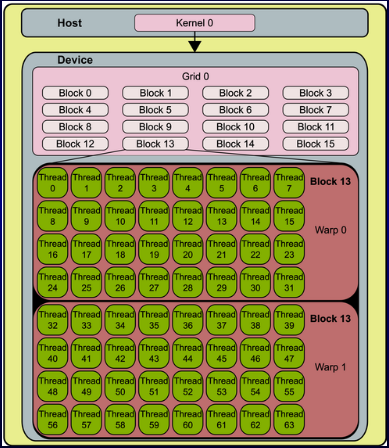
Thread Hierarchy

- Thread:
  Smallest unit of execution
- Warp:
  - Group of 32 threads, executed simultaneously
  - SIMT (Single Instruction, Multiple Thread) model
  - Divergence within a warp can impact performance
  - Warp scheduling is hardware-managed
- Thread Block:
  Group of threads that can cooperate
- Grid:
  Array of thread blocks



Kernel Functions

- Functions executed on GPU, either called by Host or Device
- The amount of shared memory, number of Grids, Blocks and Threads used are specified
- Memory is not shared between CPU and GPU
  $\Rightarrow$ Frequently used data stored on GPU if possible
- Expensive CPU-GPU and GPU-GPU communications
  $\Rightarrow$ Synchronization with CPU and other Kernels are crucial in optimization
- Kernels can be launched into seperate streams as a simple optimization mechanism

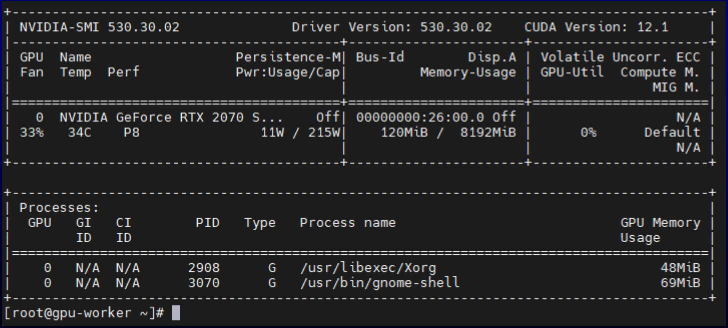


Programming in CUDA

- CUDA: Compute Unified Device Architecture
- Parallel computing platform and API model created by NVIDIA
- Allows software developers to use NVIDIA GPUs for general purpose processing
- Extends C, C++, and Fortran
- CUDA Installation and Setup:
  - Download CUDA Toolkit from NVIDIA website
  - Includes compiler, libraries, and development tools
  - Supports Windows, Linux, and macOS (limited)
- CUDA Toolkit Components:
  - NVIDIA CUDA Compiler (NVCC)
  - CUDA Runtime and Driver APIs
  - CUDA Libraries (cuBLAS, cuFFT, etc.)
  - CUDA Profiler and Debugger
  - GPU-Accelerated Libraries

Installation and Compilation

- Configure IDE for CUDA development
- Set up environment variables (PATH, CUDA_PATH)
- Install GPU drivers
- Check GPU information with **`nvidia-smi`**
- Compile CUDA programs with **`nvcc`**



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 530.30.02              Driver Version: 530.30.02    CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 2070 S...    Off| 00000000:26:00.0 Off |                  N/A |
| 33%   34C    P8              11W / 215W|    120MiB /  8192MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A      2908      G   /usr/libexec/Xorg                            48MiB |
|    0   N/A  N/A      3070      G   /usr/bin/gnome-shell                         69MiB |
+---------------------------------------------------------------------------------------+
[root@gpu-worker ~]#
```

CUDA Initialization

```
#include <cuda_runtime.h>
#include <stdio.h>

int main() {
    int deviceCount = 0;
    cudaError_t error_id = cudaGetDeviceCount(&deviceCount);

    if (error_id != cudaSuccess) {
        cout << "cudaGetDeviceCount returned  " << (int)error_id
             <<  "-> " <<  cudaGetErrorString(error_id) << endl;
        cout << "Result = FAIL" << endl; return 1;
    }
    if (deviceCount == 0) {
        cout << "There are no available device(s)
                 that support CUDA" << endl;
    } else {
        cout << "Detected " << deviceCount
             <<  " CUDA Capable device(s)" << endl;
    }

    int dev = 0; cudaSetDevice(dev);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    cout << "Device " << dev <<  ":" << deviceProp.name << endl;
    cout << "CUDA Capability Major/Minor version number:  "
    << deviceProp.major <<  "." << deviceProp.minor << endl;
    ...
    return 0;
}
```

CUDA code structure

```
__global__
void daxpy(double a, double* x, double* y, int N){
    int i=blockDim.x*blockIdx.x+threadIdx.x;
    if (i<N) y[i] = a*x[i]+y[i];
}
/* within Host code */
/* a, x_host, y_host and N are defined on Host */
...

cudaMalloc(&x, sizeof(double)*N);
cudaMalloc(&y, sizeof(double)*N);
cudaMemcpy(x, x_host,sizeof(double)*N, cudaMemcpyHostToDevice);

/* GridShape, BlockShape and SharedMem are user-defined on Host */

daxpy<<<GridShape, BlockShape, SharedMem>>>(a, x, y, N);
cudaDeviceSynchronize();

cudaMemcpy(y_host, y,sizeof(double)*N, cudaMemcpyDeviceToHost);
...
cudaFree(x);
cudaFree(y);
```

Host code, Kernel functions,
Memory allocation, deallocation and transfer

Kernel Functions

```
__global__
void daxpy(double a, double* x, double* y, int N){
    int i=blockDim.x*blockIdx.x+threadIdx.x;
    if (i<N) y[i]= a*x[i]+y[i];
}
```

- Keywords:
  `__global__` :
  Defined both on Host and Device (called from Host code)
  `__device__` :
  Defined only on Device (Host code cannot call)
- Executed on the GPU
- Predefined variables:
  - Data Type : **`dim3`**, CUDA-defined 3-D array of unsigned integers
  - $0, 1, 2$-th Elements are accessed as **`.x`**, **`.y`**, **`.z`**
  - **`gridDim`**, **`blockDim`**:
    Total numbers of blocks and threads per block
  - **`blockIdx`**, **`threadIdx`**:
    Indices of executing block and thread
- All threads execute the same instructions (SIMT)

Kernel Functions

```
/* GridShape, BlockShape and SharedMem are user-defined on Host */

daxpy<<<GridShape, BlockShape, SharedMem>>>(a, x, y, N);
cudaDeviceSynchronize();
```

- `func«< GridShape, BlockShape, SharedMem »>(args)`
- **`GridShape`**: Total number of blocks requested (Defines **`gridDim`** within kernel)
- **`BlockShape`**: Total number of threads per block requested (Defines **`blockDim`** within kernel)
- **`SharedMem`**: Shared memory per block in bytes requested
- The shapes are also of type **`dim3`**. The effects of sizes is usually much more than the shapes. The latter should be determined by optimizing memory access pattern.
- **`cudaDeviceSynchronize();`**:
  Synchronize the Device and Host to ensure that all data on Device is up-to-date before next usage.
- Kernels do NOT run consecutively in predefined order
  $\Rightarrow$ It is critical to synchronize between kernels if order is needed

Memory Allocation and Deallocation in CUDA

```
cudaMalloc(&x, sizeof(double)*N); cudaMalloc(&y, sizeof(double)*N);
cudaMemcpy(x, x_host,sizeof(double)*N, cudaMemcpyHostToDevice);
cudaMemcpy(y_host, y,sizeof(double)*N, cudaMemcpyDeviceToHost);
...
cudaFree(x); cudaFree(y);
```

- Host memory: `malloc()`, `free()`
- Pinned Host memory: `cudaMallocHost()`, `cudaFreeHost()`
- Device memory: `cudaMalloc()`, `cudaFree()`
- Unified memory: `cudaMallocManaged()`,`cudaFree()` Data migrations between CPU and GPU are handled internally by CUDA instead of programmer.
    - Convenient for quick deployments
    - Lose control of fine-tuning ⇒ Inconvenient for optimization
- Data is transferred using `cudaMemcpy()`
- Transfer direction is specified: `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`

Asynchronous Data Transfer

```
cudaHostAlloc(&x_host, sizeof(double)*N);
cudaHostAlloc(&y_host, sizeof(double)*N);
...
cudaMalloc(&x, sizeof(double)*N); cudaMalloc(&y, sizeof(double)*N);
cudaMemcpyAsync(x, x_host,sizeof(double)*N, cudaMemcpyHostToDevice);

/* do something useful on host */
...

cudaDeviceSynchronize();
...
cudaMemcpyAsync(y_host, y,sizeof(double)*N, cudaMemcpyDeviceToHost);

/* do something useful on host */
...

cudaDeviceSynchronize();
...
cudaFree(x); cudaFree(y);
...
cudaFreeHost(x_host); cudaFreeHost(y_host);
```

- One can overlap Host computation with data transfer by Asynchronous Data Transfer : `cudaMemcpyAsync()`
- Requires pinned host memory allocated and deallocated with: `cudaHostAlloc()`, `cudaFreeHost()`

CUDA Streams

```
cudaStream_t Stream0, Stream1;
cudaStreamCreate(&Stream0);
cudaStreamCreate(&Stream1);
...
func0<<<GridShape0, BlockShape0, SharedMem0, Stream0>>>(args);
func1<<<GridShape1, BlockShape1, SharedMem1, Stream1>>>(args);
...
cudaStreamSynchronize(Stream0);
cudaStreamSynchronize(Stream1);
...
cudaStreamDestroy(Stream0);
cudaStreamDestroy(Stream1);
```

- A stream is a sequence of Kernels running consecutively
- Multiple streams can run concurrently
- Create and destroy with `cudaStreamCreate()`, `cudaStreamDestroy()`

CUDA Event

```
cudaStream_t Stream0, Stream1;
cudaStreamCreate(&Stream0);
cudaStreamCreate(&Stream1);
cudaEvent_t event;
...
cudaEventCreate(&event);
func0<<<GridShape0, BlockShape0, SharedMem0, Stream0>>>(args);
cudaEventRecord(event, stream0);
cudaStreamWaitEvent(stream1, event, 0);
func1<<<GridShape1, BlockShape1, SharedMem1, Stream1>>>(args);
...
cudaStreamSynchronize(Stream0);
cudaStreamSynchronize(Stream1);
...
cudaStreamDestroy(Stream0);
cudaStreamDestroy(Stream1);
cudaEventDestroy(event);
```

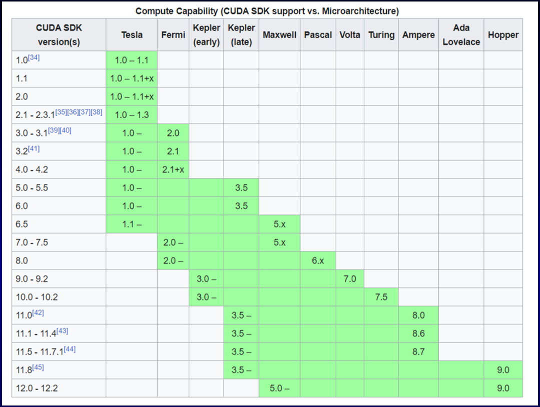
- CPU-GPU or stream-stream synchronization can be achieved by Event mechanism
- Typically used for timing: **`cudaEventElapsedTime()`** returns time difference between two Events recorded at the start and the end
- Create and destroy with **`cudaEventCreate()`**, **`cudaEventDestroy()`**

Error Handling in CUDA

```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    cout << "CUDA error:" << cudaGetErrorString(err) << endl;
}
```

- Most CUDA bulit-in functions return `cudaError_t`
- Check errors with `cudaGetLastError()`
- Get error string with `cudaGetErrorString()`

GPU Compute Capability



Compute Capability (CUDA SDK support vs. Microarchitecture)

CUDA SDK version(s)	Tesla	Fermi	Kepler (early)	Kepler (late)	Maxwell	Pascal	Volta	Turing	Ampere	Ada Lovelace	Hopper
1.0[34]	1.0 – 1.1										
1.1	1.0 – 1.1+x										
2.0	1.0 – 1.1+x										
2.1 - 2.3.1[35][36][37][38]	1.0 – 1.3										
3.0 - 3.1[39][40]	1.0 –	2.0									
3.2[41]	1.0 –	2.1									
4.0 - 4.2	1.0 –	2.1+x									
5.0 - 5.5	1.0 –	🟩	🟩	3.5							
6.0	1.0 –	🟩	🟩	3.5							
6.5	1.1 –	🟩	🟩	🟩	5.x						
7.0 - 7.5		2.0 –	🟩	🟩	5.x						
8.0		2.0 –	🟩	🟩	🟩	6.x					
9.0 - 9.2			3.0 –	🟩	🟩	🟩	7.0				
10.0 - 10.2			3.0 –	🟩	🟩	🟩	🟩	7.5			
11.0[42]				3.5 –	🟩	🟩	🟩	🟩	8.0		
11.1 - 11.4[43]				3.5 –	🟩	🟩	🟩	🟩	8.6		
11.5 - 11.7.1[44]				3.5 –	🟩	🟩	🟩	🟩	8.7		
11.8[45]				3.5 –	🟩	🟩	🟩	🟩	🟩	🟩	9.0
12.0 - 12.2					5.0 –	🟩	🟩	🟩	🟩	🟩	9.0

Compute capability depends on:

- Maximum threads per block
- Shared memory size
- Number of registers per thread
- Support for advanced features (e.g., dynamic parallelism)

Compiling and Running in CUDA

- Compiling CUDA Programs
  - Use NVCC compiler : `nvcc -O3 myprogram.cu -o myprogram`
  - Specify compute capability: `-arch=sm_XX`
  - Example: `nvcc -O3 -arch=sm_70 myprogram.cu -o myprogram`
- Running CUDA Programs
  - Environment variables : `CUDA_VISIBLE_DEVICES`, `CUDA_PATH` etc
  - Execute like a normal program: `./myprogram`
  - Detect memory errors with `cuda-memcheck`
  - Profile with Nsight

Beyond one GPU

- In scientific applications, it is essential to run on more than one GPU in parallel
- Typically in super-computing facilities, these GPUs are arranged in different nodes
- GPU-GPU and GPU-CPU communication is crucial in performance, especially across nodes (often the bottleneck)
- Hardware solutions:
  - High-bandwidth PCIe:
    New standards (PCIe 4.0 or 5.0) improve CPU-GPU communications
  - NVLink:
    High-bandwidth, low-latency GPU-GPU interconnect by NVIDIA
- Software solutions:
  - Workload distribution optimization: Minimize communications
  - CUDA-aware MPI:
    extend MPI for CPU-initiated GPU-GPU communication
  - NVSHMEM:
    NVidia implementation of OpenSHMEM, allowing GPU-initiated GPU-GPU communications

Scalings

- Amdahl's Law (Strong scaling)
  - Describes how speedup from parallelization scales with requested resources at given problem size
  - $S(n) = \frac{1}{(1-p)+\frac{p}{n}}$, S: speedup, n: number of processors, p: parallelized portion
  - Bad scaling $\Rightarrow$ Parallellized portion becomes so fast that non-parallelizable portion dominates the cost (e.g. communication among processes too slow)
- Gustafson's Law (Weak Scaling)
  - Describes how speedup scales at given proportion of requested resources to problem size
  - $S(n) = n - \alpha(n-1)$, α: non-parallelizable portion
  - Bad Scaling $\Rightarrow$ Non-paralellizable portion overwhelmingly dominates (e.g. not enough parallelization) or actual available resources cannot catch up with the requested (e.g. memory requested exceeds available cache size)
- A high-performing code should scale well weakly and strongly. Strong scaling is harder to achieve typically due to slow GPU-GPU and GPU-CPU communications

Optimization Techniques

- Use **`nvcc`** flags to control resource usage
- Optimal choice of kernel sizes:
  - Too large: unwanted overheads
  - Too small: underutilize resources
- Occupancy
  - Ratio of active warps to maximum possible active warps
  - Higher occupancy can hide latency better
  - Affected by register usage, shared memory, block size
- Combine kernels to minimize launching overheads
- Use CUDA streams for concurrent execution
- Optimize arithmetic operations (use intrinsics)

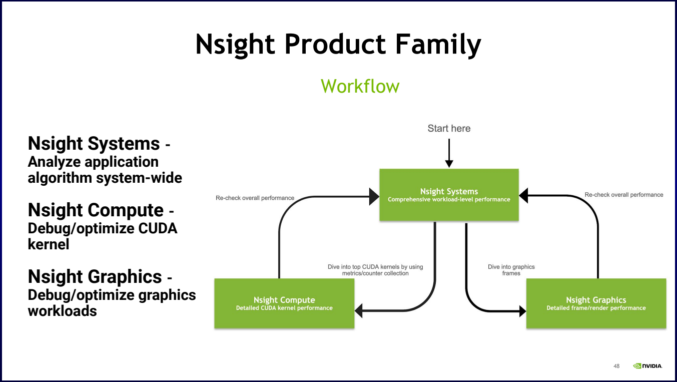
Optimization Techniques

- Thread Divergence
  - Occurs when threads in a warp take different execution paths
  - Caused by conditional statements (if, switch, etc.)
  - Can severely impact performance
  - Minimize by restructuring code or data
- Minimize data transfers and Overlap computation with data transfers
- Use asynchronous memory operations if possible
- Implement pipeline parallelism where applicable
- Maximize arithmetic density (computation per memory accesses)

Optimization Techniques

- Use appropriate data types (e.g., float vs. double)
- Balance use of registers, shared memory, and threads
- Avoid using Unified Memory that limits low-level optimizations
- Use Shared Memory within kernels for data reuse
- Use Pinned Memory (**`cudaMallocHost()`**) for faster transfers:
  - Pinned Memory is locked in a physical address in RAM, allowing direct access by devices like GPU without intermediate copies
  - It allows Direct Memory Access(DMA) transfers between host and GPU without CPU intervention
- Aim at contiguous and aligned memory accesses
- Ensure coalesced memory access :
  Threads within a warp should access nearby memory locations so that they can be combined into single transaction

CUDA Profiling Tools



Nsight Product Family

Workflow

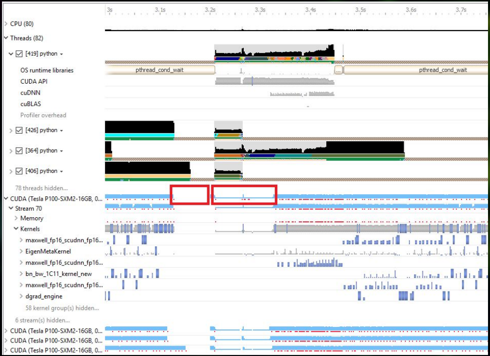
Nsight Systems - Analyze application algorithm system-wide

Nsight Compute - Debug/optimize CUDA kernel

Nsight Graphics - Debug/optimize graphics workloads

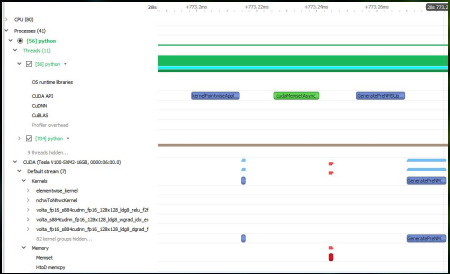
Start here
Nsight Systems
Comprehensive workload-level performance
Re-check overall performance
Re-check overall performance
Dive into top CUDA kernels by using metrics/counter collection
Dive into graphics frames
Nsight Compute
Detailed CUDA kernel performance
Nsight Graphics
Detailed frame/render performance
48
NVIDIA

Identifying Performance Bottlenecks



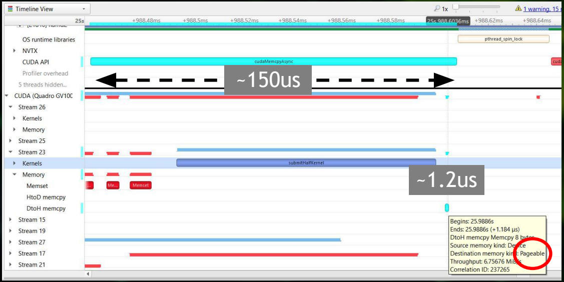
GPU idle and low utilization level of detail

Identifying Performance Bottlenecks



Fusion opportunities
CPU launch cost + small GPU work size ≈ GPU sparse idle
This can apply to DNN nodes/layers

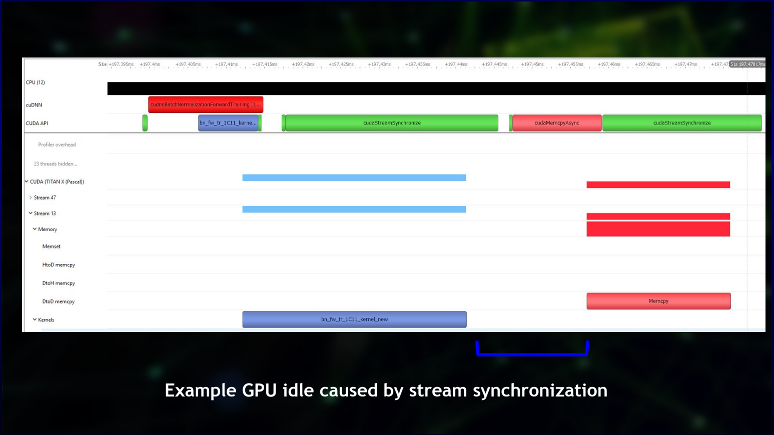
Identifying Performance Bottlenecks



Begins: 25.9886s
Ends: 25.9886s (+1.184 µs)
DtoH memcpy Memcpy 8 bytes
Source memory kind: Device
Destination memory kind: Pageable
Throughput: 6.75676 MiB/s
Correlation ID: 237265

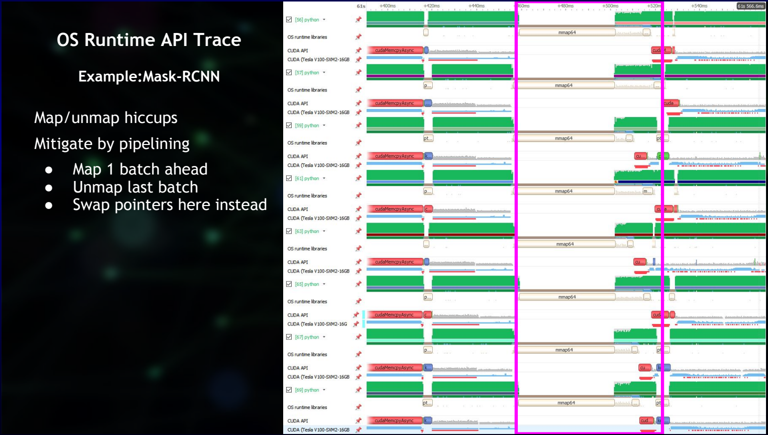
cudaMemcpyAsync behaving synchronous
Device to host pageable memory
Mitigate with pinned memory

Identifying Performance Bottlenecks



Example GPU idle caused by stream synchronization

Identifying Performance Bottlenecks



OS Runtime API Trace

Example:Mask-RCNN

Map/unmap hiccups

Mitigate by pipelining

- Map 1 batch ahead
- Unmap last batch
- Swap pointers here instead

General practices

- If porting existing codes into GPU, make sure there are unit tests to check for correctness
- Hybrid model (CPU+GPU) usually performs better than pure GPU model
- Start with the most computationally intensive parts
- Use libraries for common operations, e.g. for CUDA:
  - cuBLAS: Basic Linear Algebra Subprograms
  - cuFFT: Fast Fourier Transforms
  - cuRAND: Random Number Generation
  - cuSPARSE: Sparse Matrix Operations
  - cuSOLVER: Dense and Sparse Direct Solvers
- Implement proper error checking and validation
- Profiling and optimization is essential to achieve good scalings
- Documentation of optimization strategies

Comparison of GPU Programming Models

Feature	CUDA	OpenCL	OpenACC	oneAPI	HIP
Primary Vendor	NVIDIA	Khronos Group	NVIDIA, PGI	Intel	AMD
Language Base	C/C++	C	C, C++, Fortran	C++ (DPC++)	C++
Portability	NVIDIA only	Cross-platform	NVIDIA, x86	Intel, some x86	AMD, NVIDIA
Programming Model	Explicit	Explicit	Directive-based	SYCL-based	Explicit
Ecosystem	Extensive	Broad	Moderate	Growing	Growing
Compiler Support	nvcc	GCC, Clang, Intel, AMD, IBM, PGI	PGI, GCC, Cray	Intel DPC++	HIP, nvcc
Memory Model	Unified	Separate	Unified	Unified	Unified
Kernel Launch	«< »>	API calls	Directives	SYCL syntax	hipLaunchKernel

Concept	CUDA	OpenCL	HIP
Device	int deviceId	cl_device	int deviceId
Queue	cudaStream_t	cl_command_queue	hipStream_t
Event	cudaEvent_t	cl_event	hipEvent_t
Memory	void *	cl_mem	void *
Grid of threads	grid	NDRange	grid
Subgroup of threads	block	work-group	block
Thread	thread	work-item	thread
Thread-index	threadIdx.x	get_local_id(0)	hipThreadIdx_x
Block-index	blockIdx.x	get_group_id(0)	hipBlockIdx_x
Block-dim	blockDim.x	get_local_size(0)	hipBlockDim_x
Grid-dim	gridDim.x	get_global_size(0)	hipGridDim_x
Device Kernel	__global__	__kernel	__global__
Device Function	__device__	N/A (Implied)	__device__
Host Function	__host__ (default)	N/A (Implied)	__host__ (default)
Host + Device Function	__host__ __device__	N/A	__host__ __device__
Kernel Launch	«< »>	clEnqueueNDRangeKernel	hipLaunchKernel
Global Memory	__global__	__global	__global__
Group Memory	__shared__	__local	__shared__
Constant Memory	__constant__	__constant	__constant__
Thread Synchronization	__syncthreads	barrier(CLK_LOCAL_MEMFENCE)	__syncthreads
Atomic Builtins	atomicAdd	atomic_add	atomicAdd
Precise Math	cos(f)	cos(f)	cos(f)
Fast Math	__cos(f)	native_cos(f)	__cos(f)

Summary

- GPU is a very powerful accelerator in high performance computing
- CUDA basics and parallelization stretagies are discussed
- For efficient utilization of the computing power of GPUs, Profiling and Optimization are crucial
- Part 2: GPU Applications
  - Neural Networks
  - Astrophysics :
    - N-body simulation
    - Adaptive Mesh Refinement framework
  - Nuclear physics :
    - Lattice QCD
    - Parton shower simulation