Parallel & Distributed Computing

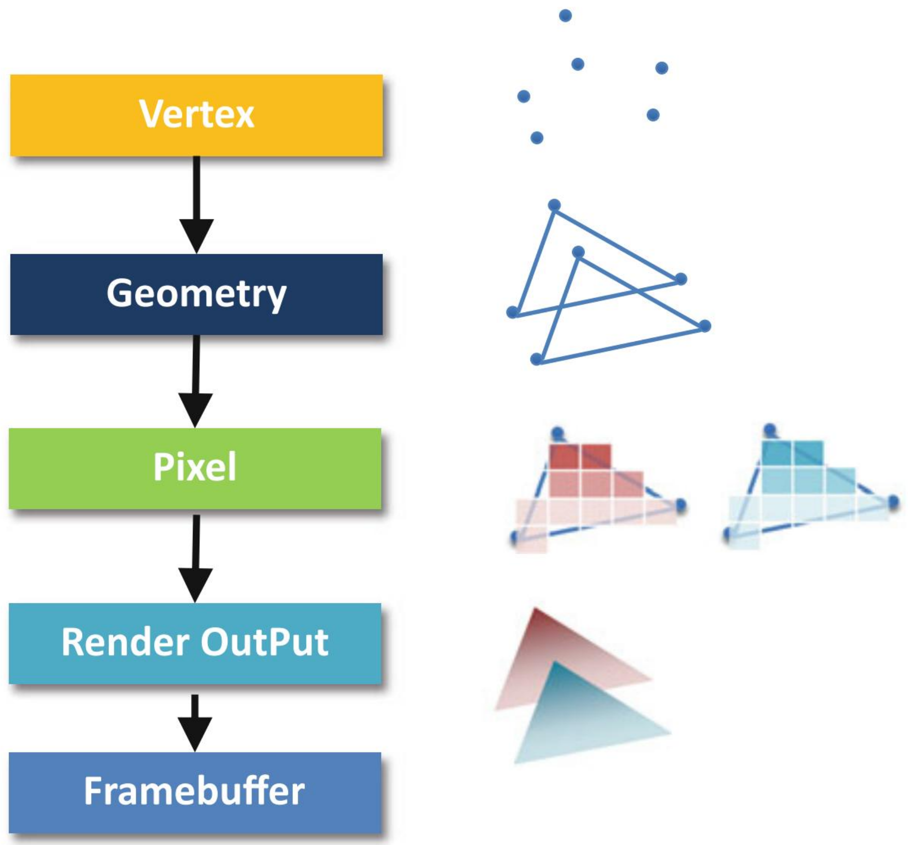
Lecture 4: GPGPU Programming

Spring 2025

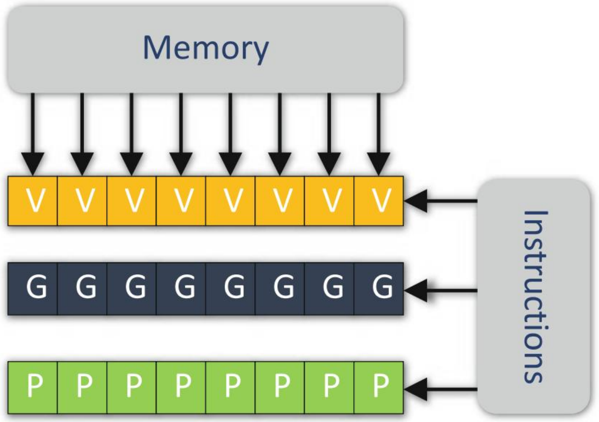
Instructor: 罗国杰

gluo@pku.edu.cn

From GPU to General-Purpose GPU (1/2)



Common pipeline of graphics processing

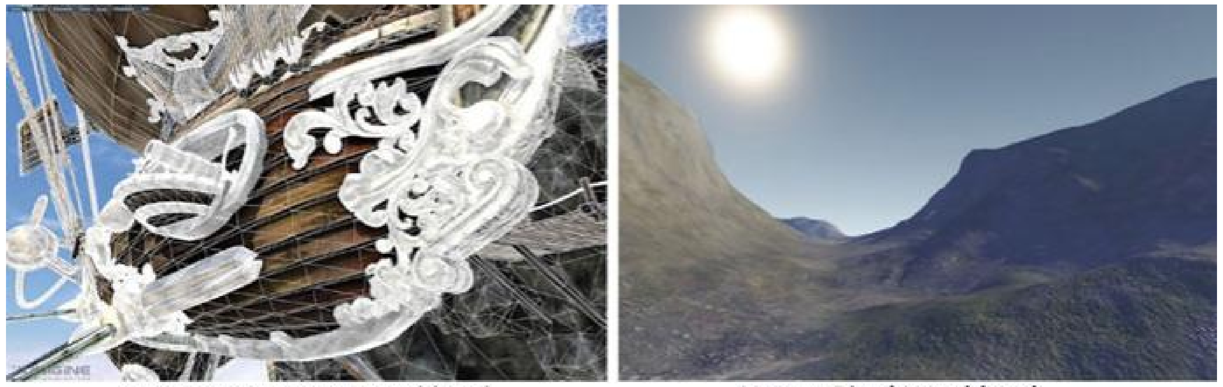


Simplified traditional GPU architecture

← vertex cores

← geometry cores

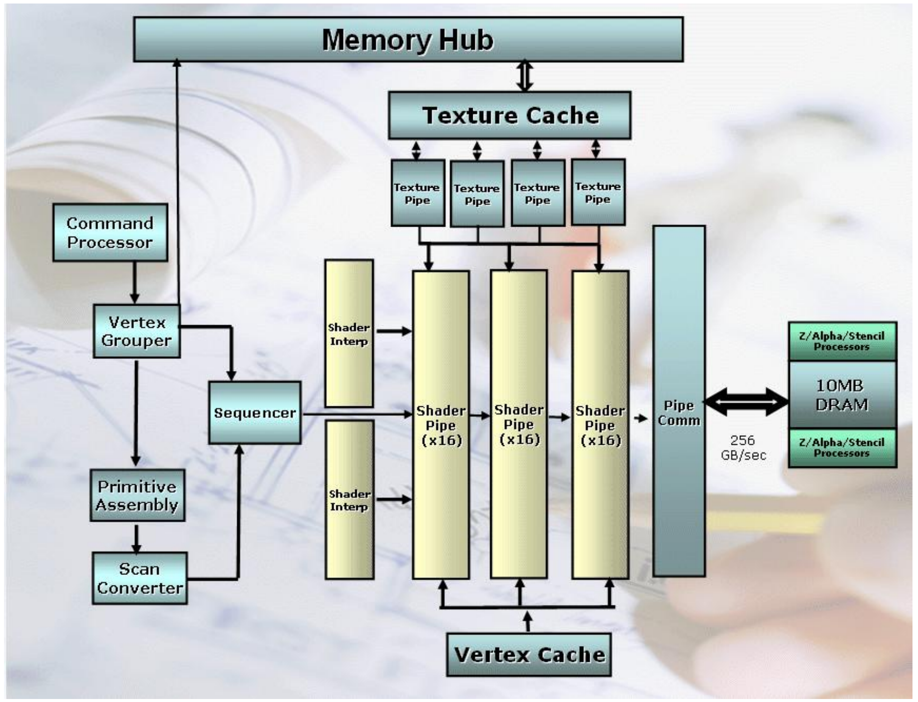
← pixel cores



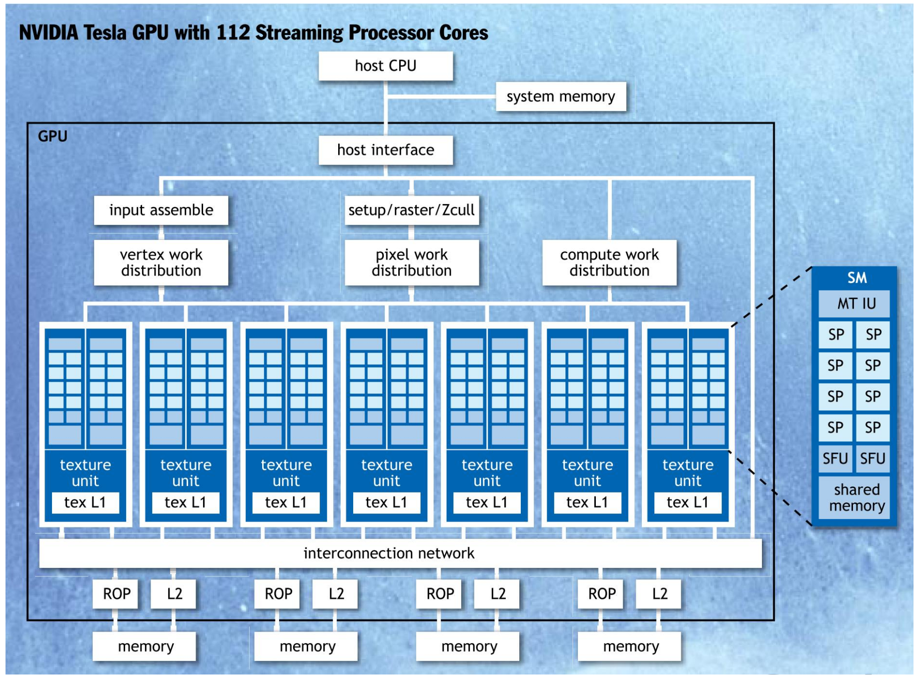
Different image inputs that cause **underutilization**

From GPU to General-Purpose GPU (2/2)

- Solution to the underutilization issue: **Unified Shader Cores**



ATI Xenos for the XBox 360 in 2004
https://www.beyond3d.com/content/articles/4/7



NVIDIA GeForce 8800GTX in 2006
[Nickolls et al., ACM Queue 2008]

GPU for G.-P. Programming Before CUDA (1/3)

- E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in Proceedings of the ACM/IEEE conference on Supercomputing (SC), Nov. 2001, pp. 55–55.
  - We present a technique for large matrix-matrix multiplies using low cost graphics hardware. The result is computed by literally visualizing the computations of a simple parallel processing algorithm. Current graphics hardware technology has limited precision and thus limits immediate applicability of our algorithm. We include results demonstrating proof of concept, correctness, speedup, and a simple application. This is therefore forward looking research: a technique ready for technology on the horizon.

GPU for G.-P. Programming Before CUDA (2/3)

- C. J. Thompson, Sahngyun Hahn, and M. Oskin, “Using modern graphics architectures for general-purpose computing: a framework and analysis,” in 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), 2002, pp. 306–317.
  - Recently, graphics hardware architectures have begun to emphasize versatility, offering rich new ways to programmatically reconfigure the graphics pipeline. In this paper we explore whether current graphics architectures can be applied to problems where general-purpose vector processors might traditionally be used. We develop a programming framework and apply it to a variety of problems, including matrix multiplication and 3-SAT. Comparing the speed of our graphics card implementations to standard CPU implementations, we demonstrate startling performance improvements in many cases, as well as room for improvement in others. We analyze the bottlenecks and propose minor extensions to current graphics architectures which would improve their effectiveness for solving general-purpose problems. Based on our results and current trends in microarchitecture, we believe that efficient use of graphics hardware will become increasingly important to high-performance computing on commodity hardware.

GPU for G.-P. Programming Before CUDA (3/3)

- Section 6 “Analysis” in the MICRO’02 paper: “Hence, we feel that these observations could have a real impact on future graphics architectures.”
  - A faster memory interface
  - Hardware to perform DMA from the graphics card to main memory
  - Operating system management of video memory
  - Better control over arithmetic precision
  - Logical Boolean operations
  - Ability to preserve state across vertex program invocations
  - A compiler

Compute Unified Device Architecture (CUDA)

- J. Nickolls, “GPU parallel computing architecture and CUDA programming model,” in IEEE Hot Chips 19 Symposium (HCS), Aug. 2007, vol. 19, pp. 1–12.
- J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming with CUDA,” Queue, vol. 6, no. 2, pp. 40–53, Mar. 2008.
- E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” IEEE Micro, vol. 28, no. 2, pp. 39–55, Mar. 2008.

*

Questions on CUDA Programming

- How to compile CUDA code?
- How do CPU and GPU interact?
- How to express parallelism?
- How to express synchronization?
- How to manage and access data?
- How is CUDA code executed on GPU?
- Misc. topics
  - debugging tools
  - useful libraries

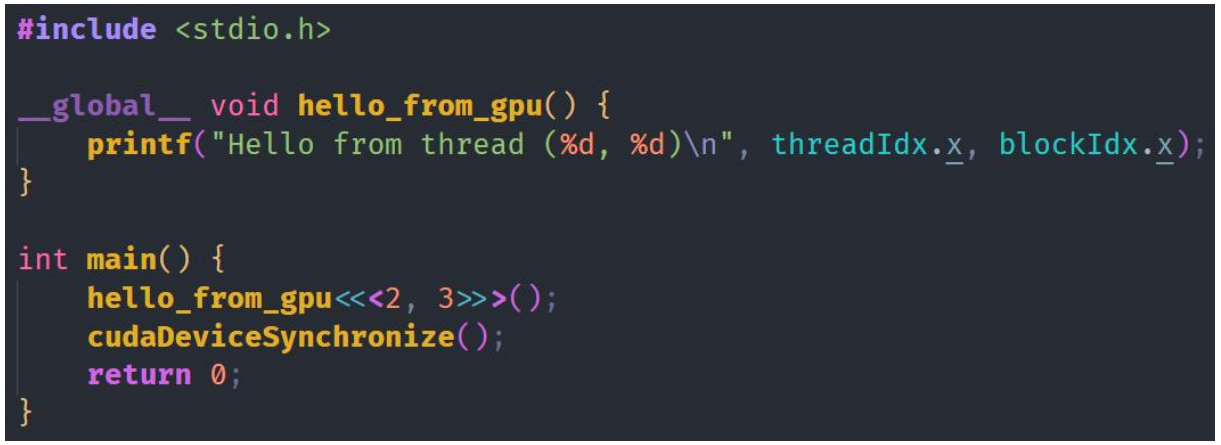
(incomplete list of) Concepts on CUDA Programming

How to compile CUDA code?	nvcc
How do CPU and GPU interact?	host, device, kernels
How to express parallelism?	threads, blocks, grid
How to express synchronization?	__syncthreads()
How to manage and access data?	cudaMalloc(), cudaMemcpy(), global/shared/constant
How is CUDA code executed on GPU?	warp scheduling

CUDA Programming: How to Compile CUDA Code?

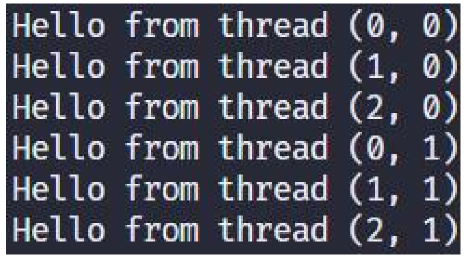
- Example: hello world from threads

```
nvcc -o hello hello.cu
```



```
#include <stdio.h>

__global__ void hello_from_gpu() {
    printf("Hello from thread (%d, %d)\n", threadIdx.x, blockIdx.x);
}

int main() {
    hello_from_gpu<<<2, 3>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

hello.cu



```
Hello from thread (0, 0)
Hello from thread (1, 0)
Hello from thread (2, 0)
Hello from thread (0, 1)
Hello from thread (1, 1)
Hello from thread (2, 1)
```

possible output

How do CPU and GPU interact?

- A Tale of Host and Device
  - host ~ CPU; device ~ GPU
  - in OpenCL/ROCm, CPUs can also serve as devices
- CPU-GPU interface
  - x86-NV5090 @ PCIe 5.0: 64 GB/s
  - Grace-Hopper NVLink C2C: 900 GB/s
  - Grace-Blackwell NVLink C2C: 1.8 TB/s
- kernel ≈ a device function interface for the host
- cudaMalloc(), cudaMemcpy()

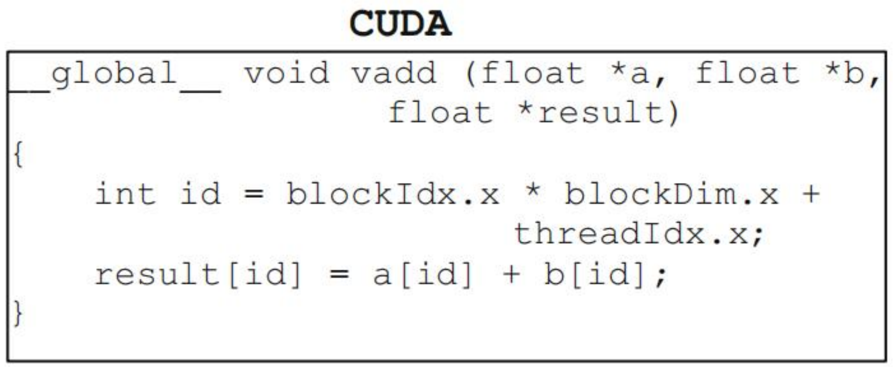
How do CPU and GPU interact?

- Host program in CUDA
  - Running as part of normal C/C++ application on CPU
  - Launch a grid of CUDA thread blocks
  - Call returns when all threads have terminated

```
int main()
{   // executed on CPU
   int A[32] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, … 31};
   int B[32] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0, … 2};
   int C[32] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, … 0};

   int *g_A, *g_B, *g_C;
   cudaMalloc((void**)&g_A, 32 * sizeof(int));
   cudaMalloc((void**)&g_B, 32 * sizeof(int));
   cudaMalloc((void**)&g_C, 32 * sizeof(int));
   cudaMemcpy(g_A, A, 32 * sizeof(int),cudaMemcpyHostToDevice);
   cudaMemcpy(g_B, B, 32 * sizeof(int),cudaMemcpyHostToDevice);

   int numBlocks = 1;
   int numThreads = 32;
   vadd <<<numBlock,numThreads>>>(g_A,g_B,g_C);
   cudaMemcpy(C, g_C, 32 * sizeof(int),cudaMemcpyDeviceToHost);
}
```

How do CPU and GPU interact?

- Kernel definition in CUDA
  - __global__ denotes a CUDA kernel function
  - Thread computes its thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)



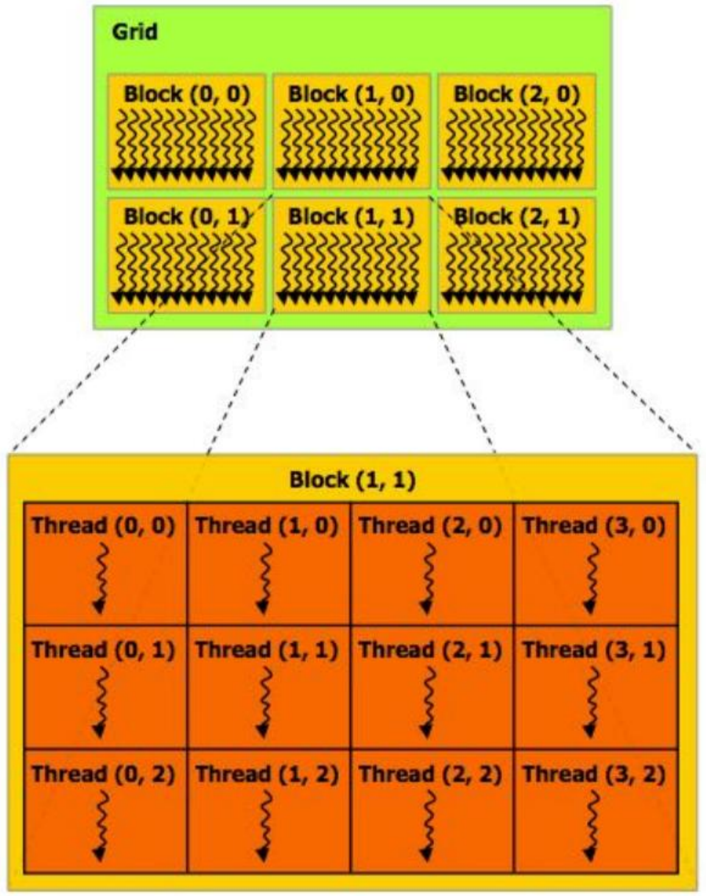
CUDA

```
__global__ void vadd (float *a, float *b,
                      float *result)
{
    int id = blockIdx.x * blockDim.x +
                          threadIdx.x;
    result[id] = a[id] + b[id];
}
```

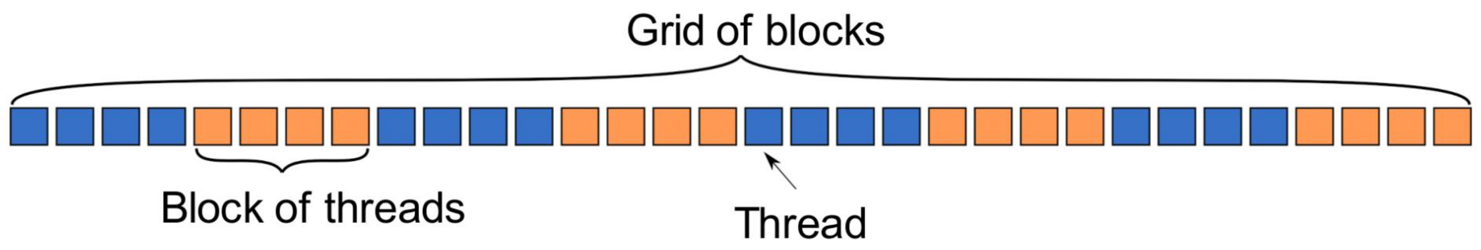
How to Express Parallelism?

- Thread Hierachy
  - Thread
    - The smallest unit of execution

    (not the smallest unit of scheduling)
  - Thread Block (Block)
    - A group of threads
    - Cooperate through shared memory and synchronization
    - Organized in up to 3-dims
  - Grid
    - A collection of thread blocks
    - Organized in up to 3-dims
    - Lauched by CPUs, executed on GPUs



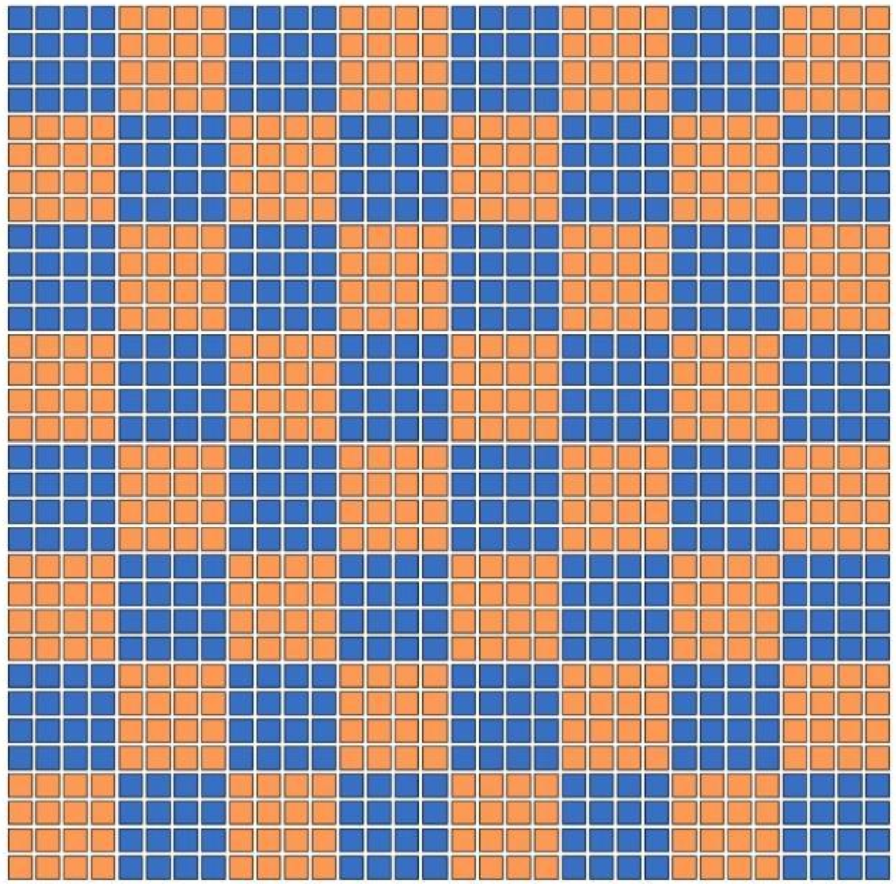
The Grid: Blocks of Threads in 1D



- Threads in grid have access to:
  - Their respective block: `blockIdx.x` (for the “Thread” above: == 4)
  - Their respective thread ID in a block: `threadIdx.x` (for the “Thread” above: == 0)
  - Their block's dimension: `blockDim.x` (for the example above: == 4)
  - The number of blocks in the grid: `gridDim.x` (for the example above: == 8)

The Grid: Blocks of Threads in 2D

- Each color is a block of threads
- Each small square is a thread
- The concept is the same in 1D and 2D
- In 2D each block and thread now has
- a two-dimensional index
- Threads in grid have access to:
  - Their respective block IDs: `blockIdx.x, blockIdx.y`
  - Their respective thread IDs in a block: `threadIdx.x, threadIdx.y`



How to Express Synchronizations?

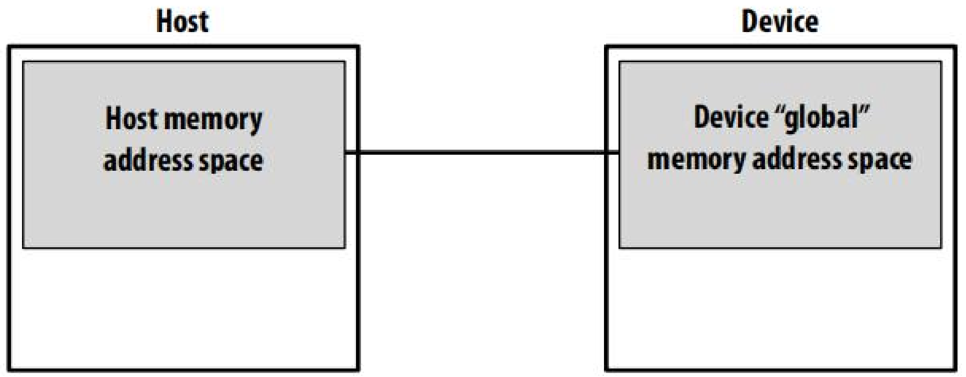
- __syncthreads()
- cudaDeviceSynchronize()
- events

(to be discussed in later lectures)

How to Manage and Access Data?

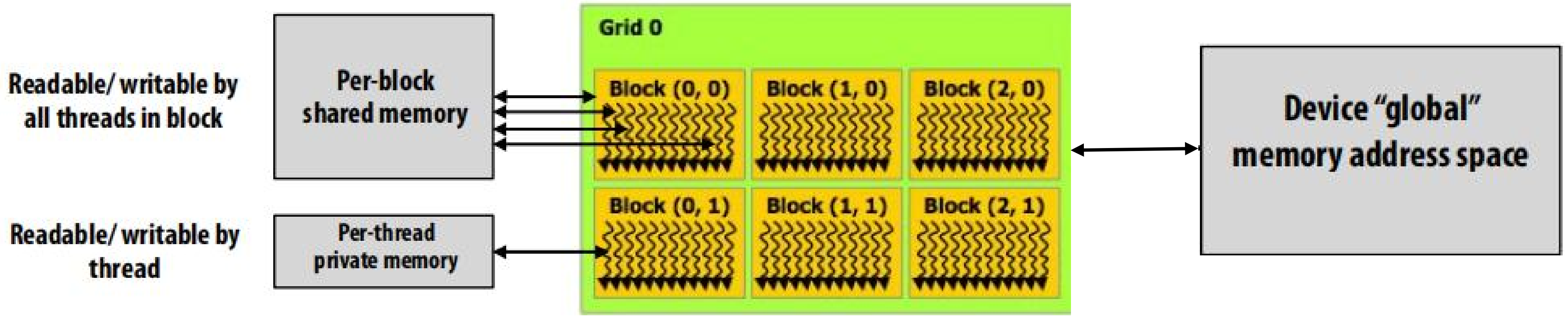
- CUDA memory model
  - Distinct host and device address spaces (when not using unified memory)
  - Memory allocation in GPU through `cudaMalloc()`
  - Move data between address spaces through `cudaMemcpy()`

```
int main()
{   // executed on CPU
    int A[32] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, … 31};
    int B[32] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0, … 2};
    int C[32] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, … 0};

    int *g_A, *g_B, *g_C;
    cudaMalloc((void**)&g_A, 32 * sizeof(int));
    cudaMalloc((void**)&g_B, 32 * sizeof(int));
    cudaMalloc((void**)&g_C, 32 * sizeof(int));
    cudaMemcpy(g_A, A, 32 * sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(g_B, B, 32 * sizeof(int),cudaMemcpyHostToDevice);

    int numBlocks = 1;
    int numThreads = 32;
    vadd <<<numBlock,numThreads>>>(g_A,g_B,g_C);
    cudaMemcpy(C, g_C, 32 * sizeof(int),cudaMemcpyDeviceToHost);
}
```


Host
Host memory address space
Device
Device "global" memory address space

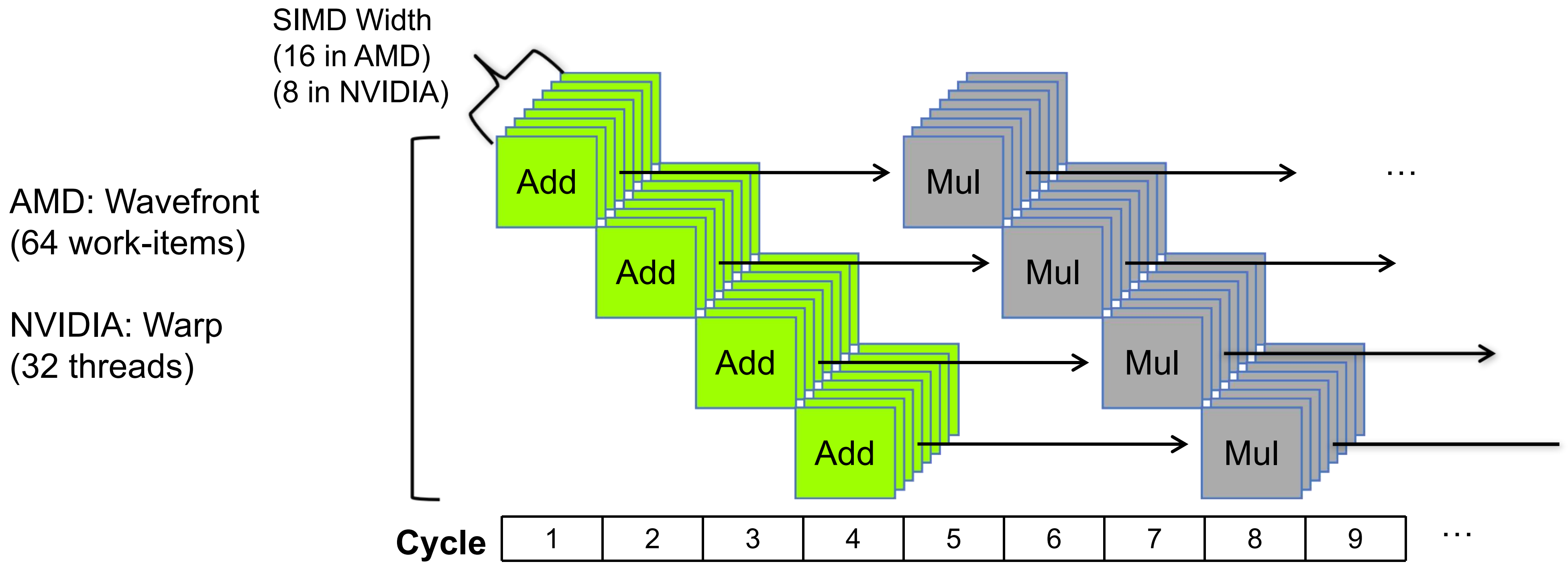
How to Manage and Access Data?

- CUDA memory model
  - Three distinct types of address spaces visible to kernels
  - `__global__`: accessible by all threads
  - `__device__`: distinct from CPU memory
  - `__shared__`: shared by threads in a block


Readable/ writable by all threads in block
Per-block shared memory
Readable/ writable by thread
Per-thread private memory
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Device "global" memory address space

How is CUDA Code Executed on GPU?

- SIMD execution can be combined with pipelining
- ALUs all execute the same instruction
- Pipelining is used to break instruction into phases
- When first instruction completes (4 cycles here), the next instruction is ready to execute



SIMD, SPMD, and SIMT (1/3)

- SIMD is an architecture
  - A single sequential instruction stream of SIMD instructions
  - For example, a matrix-vector multiplcation using AVX-512:

```
static float *outx16;
static float *avx512Multiply() {
  for (uint64_t i = 0; i < row; i++) {
    __m512 sumx16 = _mm512_set1_ps(0.0);
    for (uint64_t j = 0; j < col; j += 16) {
      __m512 a = _mm512_loadu_ps(&(t1[i*col + j]));
      __m512 b = _mm512_loadu_ps(&(t2[j]));
      sumx16 = _mm512_fmadd_ps(a, b, sumx16);
    }
    outx16[i] = _mm512_reduce_add_ps(sumx16);
  }
  return outx16;
}
```

https://www.intel.com/content/www/us/en/developer/articles/technical/accelerating-compute-intensive-workloads-with-intel-avx-512-using-microsoft-visual-studio.html

SIMD, SPMD, and SIMT (2/3)

- SPMD is a programming paradigm
  - “programmers normally write a single program that runs on all processors of an MIMD computer, relying on conditional statements when different processors should execute different sections of code.”
  - see Section 6.3 of Computer Organization and Design, Fifth Edition: The Hardware/Software interface by David A. Patterson, John L. Hennessy

SIMD, SPMD, and SIMT (3/3)

- SIMT is both a programming model and an execution model
  - Multiple instruction streams of scalar instructions
  - Can treat each thread separately
    - i.e., can execute each thread independently (on any type of scalar pipeline)
    - MIMD processing
  - Can group threads into warps flexibly
    - i.e., can group threads that are supposed to truly execute the same instruction
    - dynamically obtain and maximize benefits of SIMD processing
    - For NVIDIA GPUs, **warp** size = 32; For AMD GPUs, **wavefront** size = 64

SIMT: How to Express the Work of a Thread?

(the **Single Instruction** Stream in **SI**MT)

A simple embarrassingly parallel loop:

```
for (int i= 0; i<N; i++)
  h_a[i] *= 2.0;
```

Can be translated into a GPU kernel:

```
__global__ void myKernel(int N, double *d_a) {
  int i = threadldx.x + blockldx.x * blockDim.x;
  if (i<N) d_a[i] *= 2.0;
}
```

- A device function that will be launched from the host program is called a kernel & is declared with the __global__ attribute
- Kernels should be declared void
- All pointers passed to kernels must point to memory on the device
- All threads execute the kernel's body "simultaneously"
- Each thread uses its unique thread and block IDs to compute a global ID
- There could be more than N threads in the grid

SIMT: How to Express the Thread Structure?

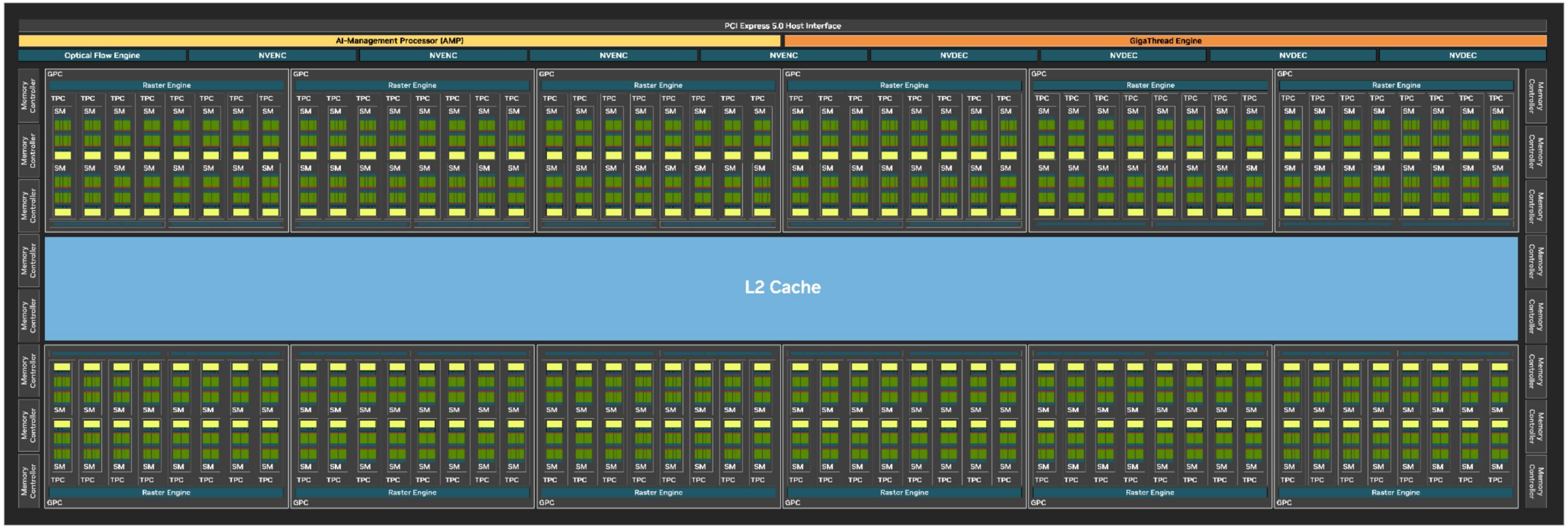
(The Grid: **Multi-threading** in **SI**MT)

Kernels are launched from the host:

```
dim3 threads(256,1,1);             // 3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1);   // 3D dimensions the grid of blocks
myKernel<<<blocks,threads,0,0>>>(N,a);
```

GPU Architecture: a Blackwell Example (1/2)

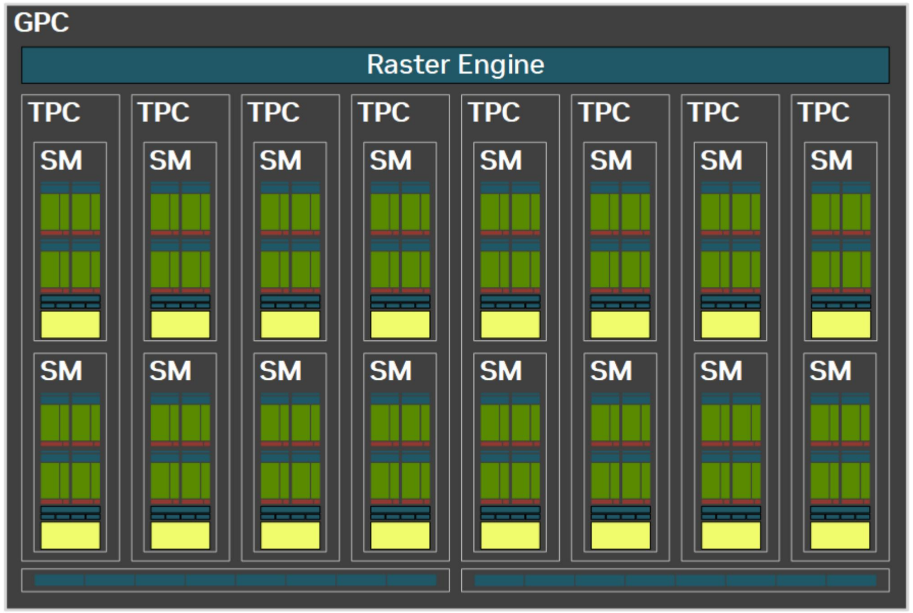
- The full GB202 GPU includes 12 Graphics Processing Clusters (GPCs), 12×8=96 Texture Processing Clusters (TPCs), 96×2=192 Streaming Multiprocessors (SMs), and a 512-bit memory interface with sixteen 32-bit memory controllers.

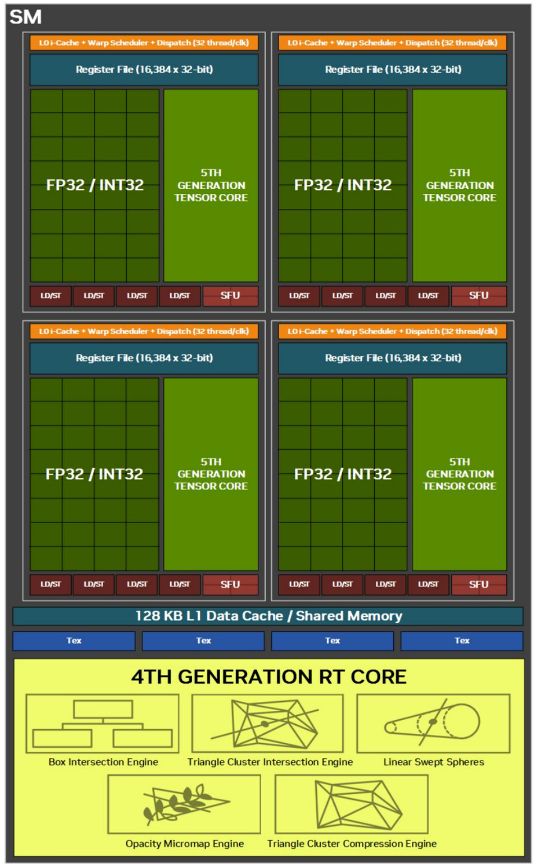


https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf

GPU Architecture: a Blackwell Example (1/2)

- each SM includes 128 CUDA Cores, one Blackwell 4th-gen RT Core, four Blackwell 5th-gen Tensor Cores, 4 Texture Units, a 256 KB Register File, and 128 KB of L1/Shared Memory
  - also includes 2 FP64 Cores per SM which are not depicted in the above diagram. The FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations.

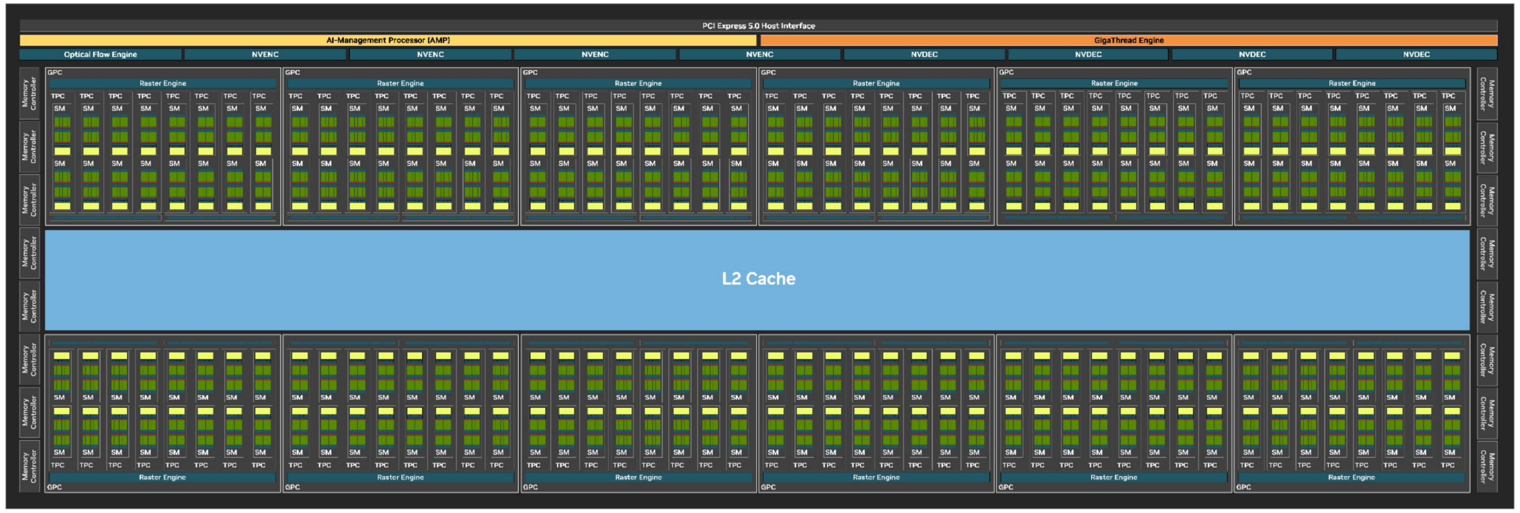


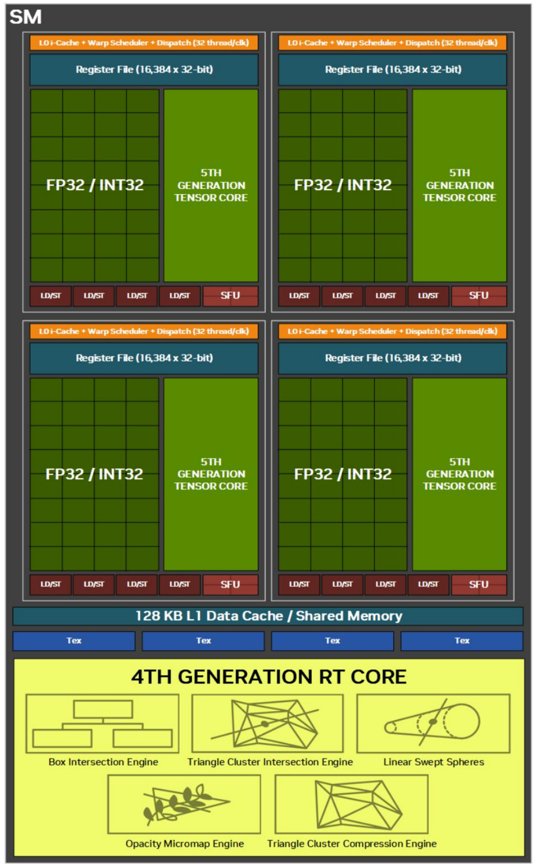

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
5TH GENERATION TENSOR CORE
LD/ST
SFU
128 KB L1 Data Cache / Shared Memory
Tex
4TH GENERATION RT CORE
Box Intersection Engine
Triangle Cluster Intersection Engine
Linear Swept Spheres
Opacity Micromap Engine
Triangle Cluster Compression Engine

https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf

Why the Grid-Block-Thread Hierarchy?

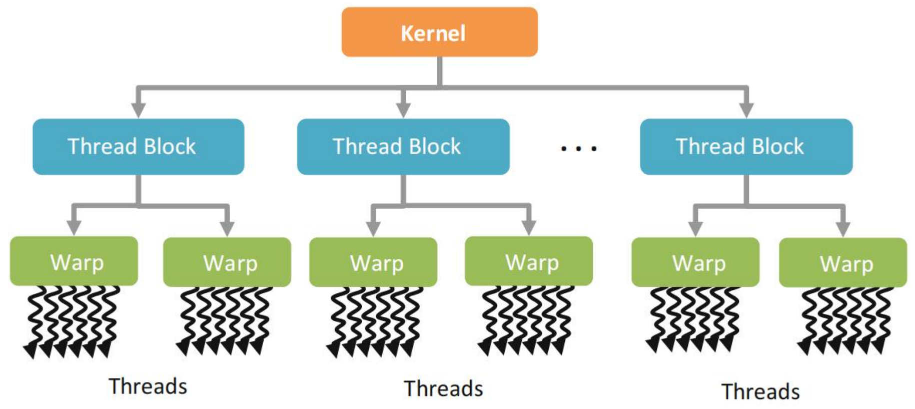
- For flexiblility and scalability
- shared memory
- __syncthreads()

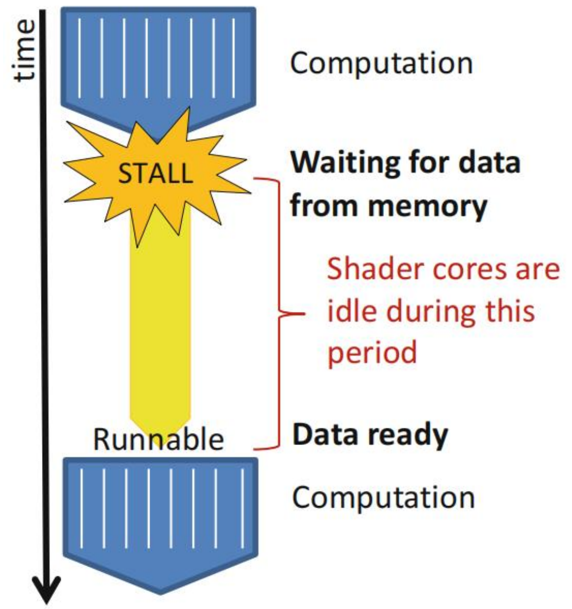


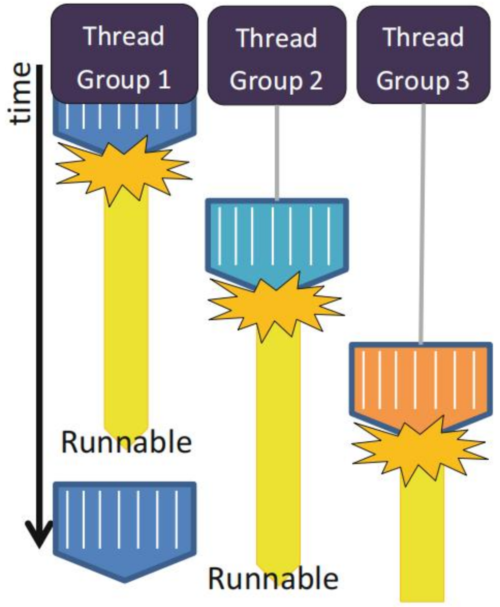

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
5TH GENERATION TENSOR CORE
LD/ST
SFU
128 KB L1 Data Cache / Shared Memory
Tex
4TH GENERATION RT CORE
Box Intersection Engine
Triangle Cluster Intersection Engine
Linear Swept Spheres
Opacity Micromap Engine
Triangle Cluster Compression Engine

GPU Architecture

- Execution model
  - GPU execution without/with batched thread groups
  - Single-instruction multiple threads (SIMT)
  - Hierachical thread execution model

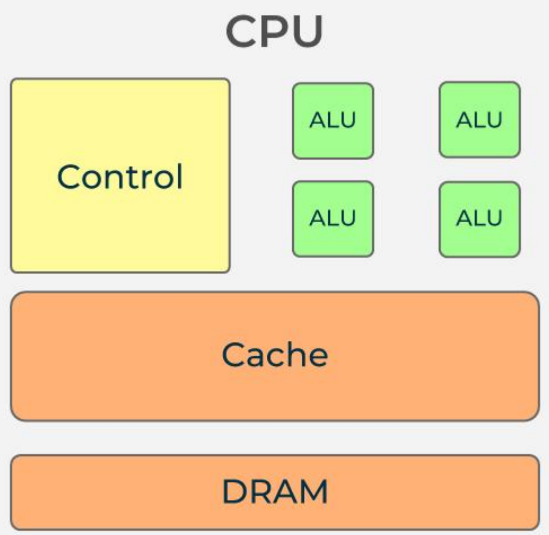



time
Computation
STALL
Waiting for data from memory
Shader cores are idle during this period
Runnable
Data ready
Computation



GPU vs CPU

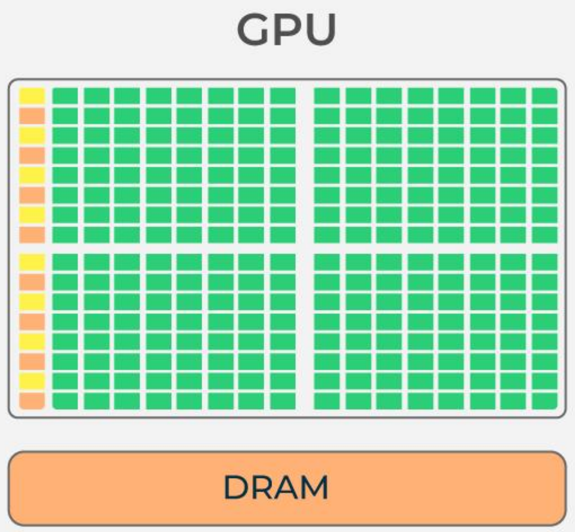
- Thread-centric



- **CPU (Central Processing Unit)**

- Executes general-purpose tasks (arithmetic, logic).
- Handles complex instructions and task switching.
- CPU has a lower core count (4-8 cores) than others, focusing on delivering higher performance for tasks that rely on a single core.

- Throughput-centric

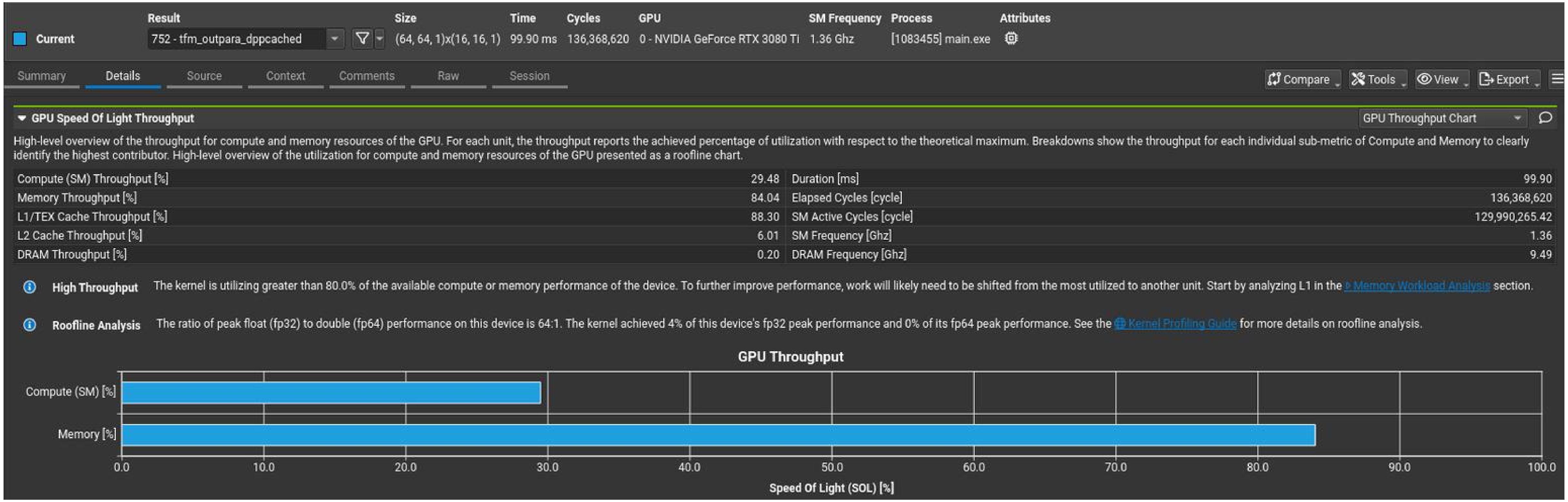


- **GPU (Graphics Processing Unit)**

- Used for parallel processing tasks (graphics, AI).
- Has many cores (100s or 1000s of cores) for handling multiple operations simultaneously.
- GPUs use a specialized type of DRAM known as VRAM (Video RAM), specifically designed to handle graphical data and textures.

https://www.geeksforgeeks.org/difference-between-cpu-and-gpu/

Nsight Compute Profiling Report Example (1/3)



	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	752 - tfm_outpara_dppcached	(64, 64, 1)x(16, 16, 1)	99.90 ms	136,368,620	0 - NVIDIA GeForce RTX 3080 Ti	1.36 Ghz	[1083455] main.exe	

GPU Speed Of Light Throughput

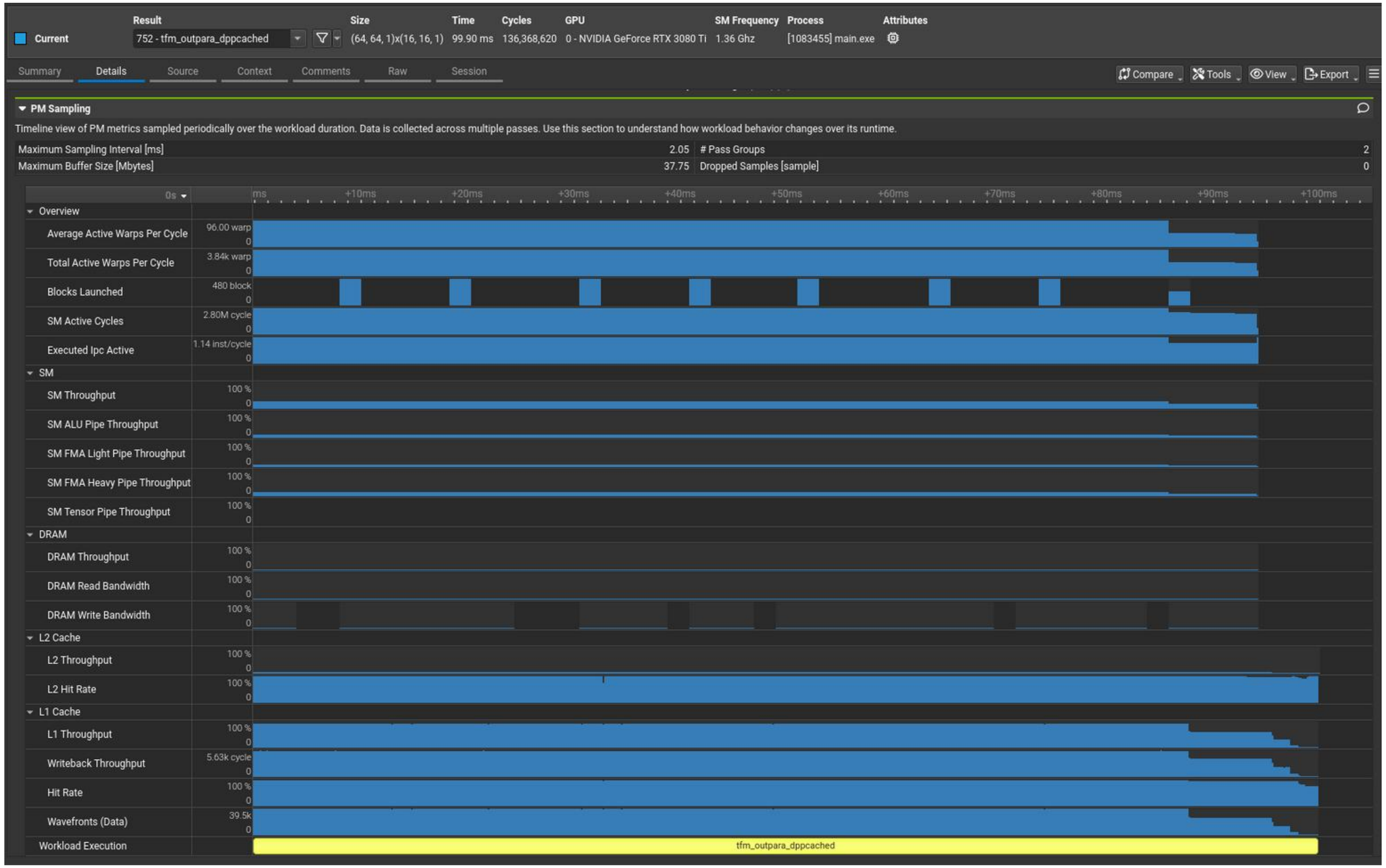
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify 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 [%]	29.48	Duration [ms]	99.90
Memory Throughput [%]	84.04	Elapsed Cycles [cycle]	136,368,620
L1/TEX Cache Throughput [%]	88.30	SM Active Cycles [cycle]	129,990,265.42
L2 Cache Throughput [%]	6.01	SM Frequency [Ghz]	1.36
DRAM Throughput [%]	0.20	DRAM Frequency [Ghz]	9.49

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing L1 in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 4% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Nsight Compute Profiling Report Example (2/3)



	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	752 - tfm_outpara_dppcached	(64, 64, 1)x(16, 16, 1)	99.90 ms	136,368,620	0 - NVIDIA GeForce RTX 3080 Ti	1.36 Ghz	[1083455] main.exe	

Summary Details Source Context Comments Raw Session

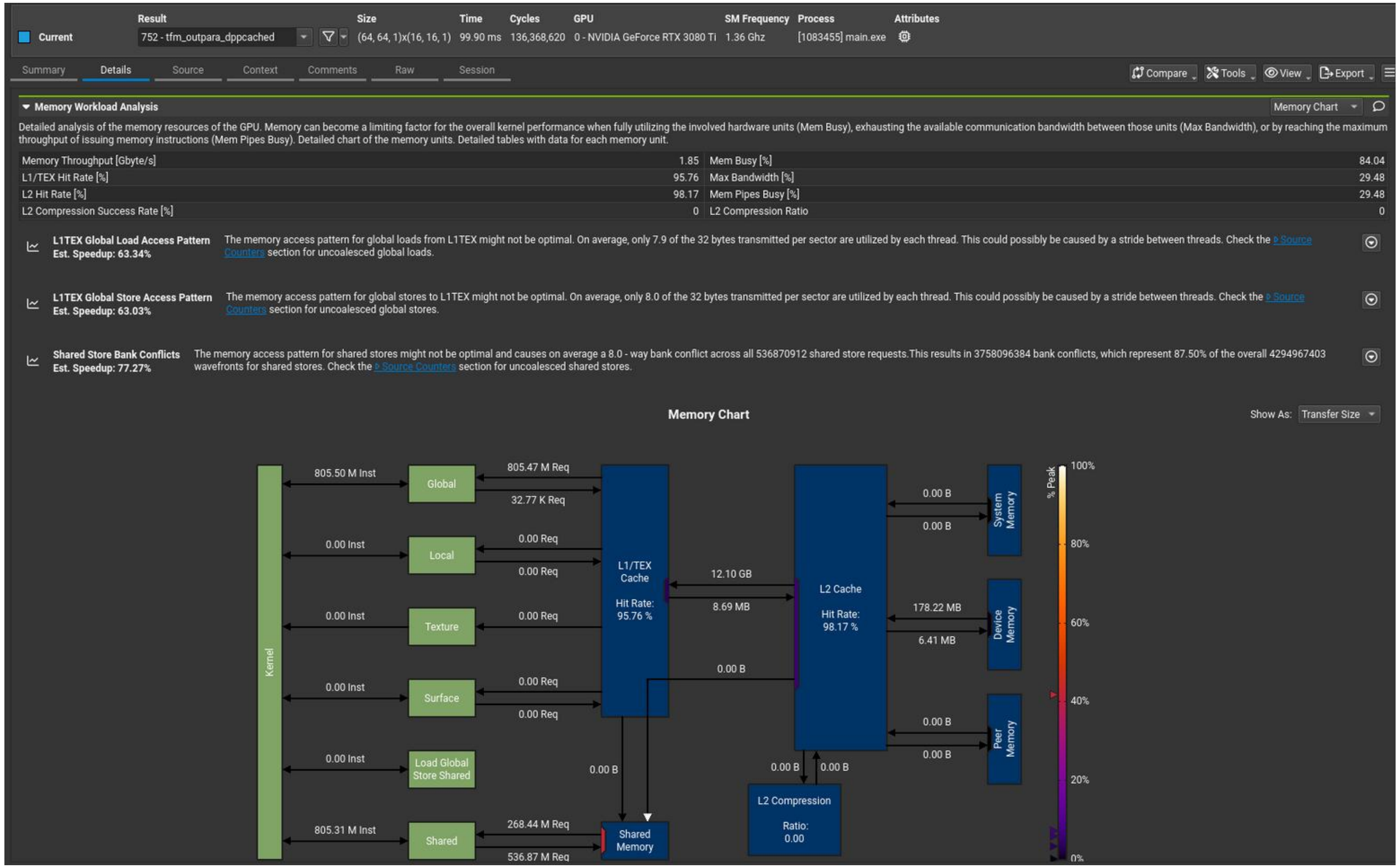
PM Sampling

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand how workload behavior changes over its runtime.

Maximum Sampling Interval [ms]	2.05	# Pass Groups	2
Maximum Buffer Size [Mbytes]	37.75	Dropped Samples [sample]	0

Compare
Tools
View
Export
0s
ms
+10ms
+20ms
+30ms
+40ms
+50ms
+60ms
+70ms
+80ms
+90ms
+100ms
Overview
Average Active Warps Per Cycle 96.00 warp 0
Total Active Warps Per Cycle 3.84k warp 0
Blocks Launched 480 block 0
SM Active Cycles 2.80M cycle 0
Executed Ipc Active 1.14 inst/cycle 0
SM
SM Throughput 100 % 0
SM ALU Pipe Throughput 100 % 0
SM FMA Light Pipe Throughput 100 % 0
SM FMA Heavy Pipe Throughput 100 % 0
SM Tensor Pipe Throughput 100 % 0
DRAM
DRAM Throughput 100 % 0
DRAM Read Bandwidth 100 % 0
DRAM Write Bandwidth 100 % 0
L2 Cache
L2 Throughput 100 % 0
L2 Hit Rate 100 % 0
L1 Cache
L1 Throughput 100 % 0
Writeback Throughput 5.63k cycle 0
Hit Rate 100 % 0
Wavefronts (Data) 39.5k 0
Workload Execution
tfm_outpara_dppcached

Nsight Compute Profiling Report Example (3/3)



	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	752 - tfm_outpara_dppcached	(64, 64, 1)x(16, 16, 1)	99.90 ms	136,368,620	0 - NVIDIA GeForce RTX 3080 Ti	1.36 Ghz	[1083455] main.exe	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

Memory Workload Analysis — Memory Chart

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	1.85	Mem Busy [%]	84.04
L1/TEX Hit Rate [%]	95.76	Max Bandwidth [%]	29.48
L2 Hit Rate [%]	98.17	Mem Pipes Busy [%]	29.48
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L1TEX Global Load Access Pattern
Est. Speedup: 63.34%
The memory access pattern for global loads from L1TEX might not be optimal. On average, only 7.9 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the ▷ Source Counters section for uncoalesced global loads.

L1TEX Global Store Access Pattern
Est. Speedup: 63.03%
The memory access pattern for global stores to L1TEX might not be optimal. On average, only 8.0 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the ▷ Source Counters section for uncoalesced global stores.

Shared Store Bank Conflicts
Est. Speedup: 77.27%
The memory access pattern for shared stores might not be optimal and causes on average a 8.0 - way bank conflict across all 536870912 shared store requests.This results in 3758096384 bank conflicts, which represent 87.50% of the overall 4294967403 wavefronts for shared stores. Check the ▷ Source Counters section for uncoalesced shared stores.

Memory Chart

Show As: Transfer Size

*

References (HIP & CUDA)

- AMD ROCm documentation
  - https://rocm.docs.amd.com/en/latest/
- ROCm exmaples
  - https://github.com/ROCm/rocm-examples/
- HIP CPU Runtime
  - https://github.com/ROCm/HIP-CPU/
  - a header-only library that allows CPUs to execute unmodified HIP code
- Omniperf: a system performance profiling tool
  - https://rocm.github.io/omniperf/
- CUDA Toolkit Documentation
  - https://docs.nvidia.com/cuda/
- NVIDIA Technologies - Architectures
  - https://www.nvidia.com/en-us/technologies/

Incomplete "History" of CUDA

- since CUDA 4.0: GPUDirect v2.0
- since CUDA 5.0: Dynamic Parallelism
- since CUDA 5.5: Multi-Process Service (MPS)
- since CUDA 6.0: Unified Memory
- since CUDA 6.5: support for using __shfl intrinsics
- since CUDA 7.0: CUDA Streams
- since CUDA 9.0: Cooperative Groups
- since CUDA 10.0: CUDA Graphs
- since CUDA 10.2: Virtual Memory Management APIs
- since CUDA 11.0.1: data format BF16, compute type TF32, and DMMA instruction,
- since CUDA 11.7: NVIDIA Open GPU Kernel Modules
- since CUDA 12.0.0: tensor operations via public PTX

https://developer.nvidia.com/cuda-toolkit-archive

"History" of NVIDIA GPU Architectures

- Blackwell Architecture (March 2024)
  - Fueling accelerated computing and generative AI with unparalleled performance, efficiency, and scale.
- Hopper Architecture (March 2022)
  - Extraordinary performance, scalability, and security for every data center.
- Ada Lovelace Architecture (September 2022)
  - Performance and energy efficiency for endless possibilities.
- Ampere Architecture (2020)
- Turing Architecture (2018)
- Volta Architecture (2017)
- Pascal Architecture (2016)
- Maxwell Architecture (2014)
- Kepler Architecture (2012)
- Fermi Architecture (2010)
- Tesla Architecture (2006)
- Curie Architecture (2004)
- Rankine (2003)
- Kelvin (2001)
- Celsius (1999)

https://www.nvidia.com/en-us/technologies/

Summary

- Origins of GPGPU
- Very basic concepts in CUDA
- NVIDIA GPU architectures
- Preview of debugging tools
- Review of CUDA/arch history