INTRODUCTION TO GPU ARCHITECTURE & PROGRAMMING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

APRIL 2024

Australian National University

Logistics

➢ Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.

GPU SM Architecture & Execution Model

A Real GPU Architecture: NVIDIA TESLA H100

➢ The NVIDIA "Hopper" H100 The NVIDIA GH100 GPU is composed of multiple GPU Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), L2 cache, and HBM3 memory controllers.

➢ The full implementation of the GH100 GPU includes the following units:
  ➢ 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, 144 SMs per full GPU
  ➢ 128 FP32 CUDA Cores per SM, 18432 FP32 CUDA Cores per full GPU
  ➢ 4 Fourth-Generation Tensor Cores per SM, 576 per full GPU
  ➢ 6 HBM3 or HBM2e stacks, 12 512-bit Memory Controllers
  ➢ 60 MB L2 Cache
  ➢ Fourth-Generation NVLink and PCIe Gen 5

https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

A Real GPU Architecture: NVIDIA TESLA H100

- The GPU hardware parallelism is achieved through the replication of SMs.
- Each SM has the following key components
  - CUDA cores (e.g. FP32, FP64, Tensor cores)
  - Shared Memory & L1 Cache
  - Register File Load(LD)/Store(DT) Units, Special Function Units (SFU) Warp Scheduler
- When a grid is launched its thread blocks are distributed among available SMs by the GigaThread engine (see previous slide)
- All threads in a block are executed by the same SM
- Multiple thread blocks may be assigned to the same SM at once
- Instructions within a single thread are pipelined to leverage ILP

https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

---

A Real GPU Architecture: NVIDIA TESLA H100

Table 1.   NVIDIA H100 Tensor Core GPU Performance Specs

	NVIDIA H100 SXM5	NVIDIA H100 PCIe
Peak FP64	33.5 TFLOPS	25.6 TFLOPS
Peak FP64 Tensor Core	66.9 TFLOPS	51.2 TFLOPS
Peak FP32	66.9 TFLOPS	51.2 TFLOPS
Peak FP16	133.8 TFLOPS	102.4 TFLOPS
Peak BF16	133.8 TFLOPS	102.4 TFLOPS
Peak TF32 Tensor Core	494.7 TFLOPS	989.4 TFLOPS[1]	378 TFLOPS	756 TFLOPS[1]
Peak FP16 Tensor Core	989.4 TFLOPS	1978.9 TFLOPS[1]	756 TFLOPS	1513 TFLOPS[1]
Peak BF16 Tensor Core	989.4 TFLOPS	1978.9 TFLOPS[1]	756 TFLOPS	1513 TFLOPS[2]
Peak FP8 Tensor Core	1978.9 TFLOPS	3957.8 TFLOPS[1]	1513 TFLOPS	3026 TFLOPS[1]
Peak INT8 Tensor Core	1978.9 TOPS	3957.8 TOPS[1]	1513 TOPS	3026 TOPS[1]

1.   **Effective TFLOPS / TOPS using the Sparsity feature**

What is a TFLOP?

A TFLOP or Teraflop represents the ability to process **one trillion floating point operations per second.**

What does a TFLOP look like?

For square, n x n matrices using the standard matrix multiplication algorithm the total number of operations is $O(n^3)$. A matrix multiply with $n=10^4$ rows will require $O(n^3) = 10^{12}$ operations, about 1 TFLOP.

https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

---

The Single Instruction Multiple Thread (SIMT) Model
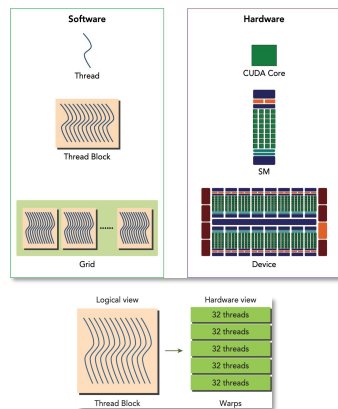
- CUDA uses a Single Instruction Multiple Thread (SIMT) architecture to manage and execute threads in groups of 32 called **warps**.
- Each SM partitions the thread blocks into warps that it then schedules for execution on available hardware resources.
- Threads in a warp execute the same instruction at the same time.
- The SIMT model includes three key features that SIMD does not:
  - Each thread has its own instruction address counter.
  - Each thread has its own register state.
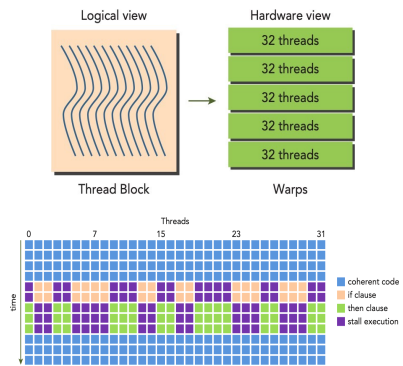 - Each thread can have an independent execution path.

Thread Block Clusters and Grids with Clusters
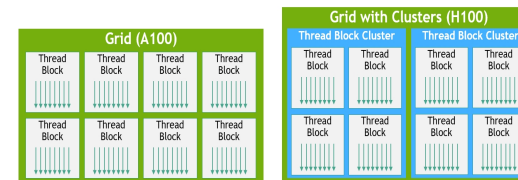
A Grid is composed of Thread Blocks in the legacy CUDA programming model as in A100, shown in the left half of the above diagram. The Hopper architecture adds an optional Cluster hierarchy, shown in the right half of the diagram.

- **With Clusters, it is possible for all the threads to directly access other SM's shared memory with load, store, and atomic operations.**
- This feature is called Distributed Shared Memory (DSMEM) because the shared memory's virtual address space is logically distributed across all the Blocks in the Cluster.
- DSMEM enables more efficient data exchange between SMs, where data no longer needs to be written to and read from global memory to pass the data.
- The dedicated SM-to-SM network for Clusters ensures fast, low latency access to remote DSMEM.
- Compared to using global memory, DSMEM accelerates data exchange between Thread Blocks by about 7x.
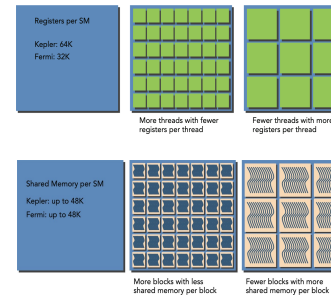
Warp Execution

Logical view → Hardware view

Thread Block → Warps

32 threads (×5)

Threads
0 7 15 23 31

time

Legend:
- coherent code
- if clause
- then clause
- stall execution

- A thread block is mapped to an SM and executed in warps
- The number of warps for a thread block can be determined as *ThreadsPerBlock*/32
- If thread block size is not an even multiple of warp size, some threads in the last warp are left inactive
- GPUs have very simple branch prediction mechanisms → conditionals are problematic as they cause *warp divergence*, *i.e.* threads in the same warp executing different instructions
- If threads of a warp diverge, the warp serially executes each branch path, disabling threads that do not take that path
- Warp divergence can cause significantly degraded performance (up to 1/32)
- Branch divergence occurs only within a warp. Different conditional values in different warps do not cause warp divergence.

Warp Scheduling: Resource Limitations

Registers per SM
Kepler: 64K
Fermi: 32K

More threads with fewer registers per thread | Fewer threads with more registers per thread

Shared Memory per SM
Kepler: up to 48K
Fermi: up to 48K

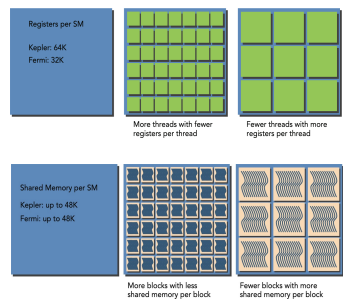More blocks with less shared memory per block | Fewer blocks with more shared memory per block

- The number of warps allocated to an SM depends on the resources it requires and affects performance significantly
- The local execution context of a warp mainly consists of program counters, registers and shared memory
- The execution context of each warp maintained on-SM during its lifetime of the warp → warp context switch has no cost.
- Each SM has a fixed number of of 32-bit registers (256KB on H100) and of shared memory (up to 228KB on H100) to be shared among threads
- The number of thread blocks and warps allocated to an SM depends on how many registers and shared memory each thread and thread block requires
- These memory requirement change based the kernel code

Warp Scheduling: Resource Limitations

Registers per SM
Kepler: 64K
Fermi: 32K

More threads with fewer registers per thread | Fewer threads with more registers per thread

Shared Memory per SM
Kepler: up to 48K
Fermi: up to 48K

More blocks with less shared memory per block | Fewer blocks with more shared memory per block
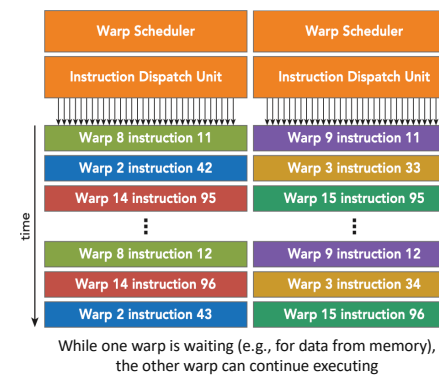
- If a thread consumes more registers, fewer warps can be placed on an SM (more registers per warp)
- If thread block consumes more shared memory, fewer thread blocks can be processed simultaneously by an SM
- If there are insufficient registers or shared memory on each SM to process at least one block, the kernel launch will fail

Latency Hiding Through Warp Scheduling

Warp Scheduler	Warp Scheduler
Instruction Dispatch Unit	Instruction Dispatch Unit
Warp 8 instruction 11	Warp 9 instruction 11
Warp 2 instruction 42	Warp 3 instruction 33
Warp 14 instruction 95	Warp 15 instruction 95
⋮	⋮
Warp 8 instruction 12	Warp 9 instruction 12
Warp 14 instruction 96	Warp 3 instruction 34
Warp 2 instruction 43	Warp 15 instruction 96

time

While one warp is waiting (e.g., for data from memory), the other warp can continue executing
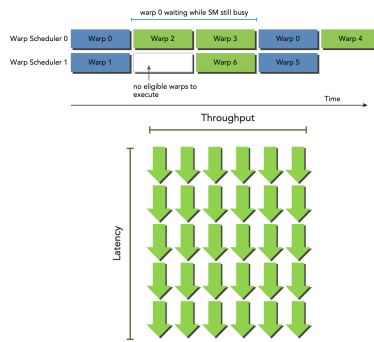
- An SM relies on thread-level parallelism to maximize utilization of its functional units
- This works essentially as hyperthreading, but where the equivalent of a thread is a warp
- Full compute resource utilization is achieved when all warp schedulers have an eligible warp at every clock cycle.
- This ensures that the latency of each instruction can be hidden by issuing other instructions in other resident warps.
- **Latency hiding is particularly important in GPU programming**: GPU instruction latency is hidden by computation from other warps (as opposed to CPUs which are designed for minimizing it!)

Latency Hiding Through Control of Warp Scheduling

Warp Scheduler 0 | Warp 0 | Warp 2 | Warp 3 | Warp 0 | Warp 4
Warp Scheduler 1 | Warp 1 | | Warp 6 | Warp 5
warp 0 waiting while SM still busy
no eligible warps to execute
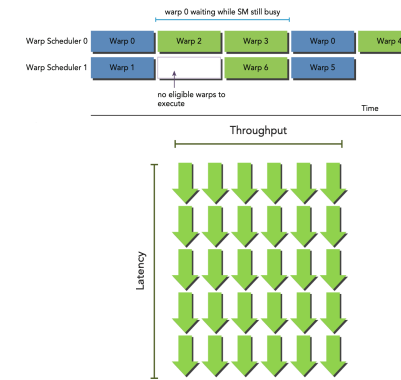Time
Throughput
Latency

- The instruction latency can derive from either arithmetic or memory instructions
- Arithmetic instruction latency varies typically between 4 an 20 cycles
- Global memory instruction latency ~500 cycles for global memory accesses (uncached transactions)
- The number of active warps required to hide latency can be estimated with Little's Law

  $\#RequiredWarps = Latency \times Throughput$

- Arithmetic operations: On H100 most single-precision ops have a latency of 4 cycles, while double-precision ones of 8 cycles
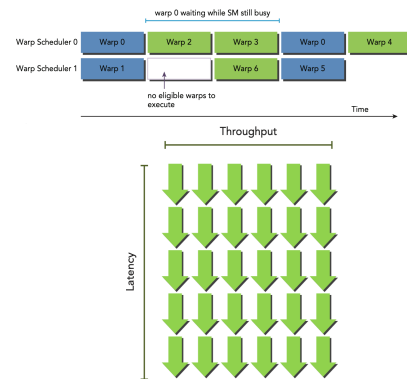- Global memory operations: Latency ~500 cycles

---

Latency Hiding Through Control of Warp Scheduling

Warp Scheduler 0 | Warp 0 | Warp 2 | Warp 3 | Warp 0 | Warp 4
Warp Scheduler 1 | Warp 1 | | Warp 6 | Warp 5
warp 0 waiting while SM still busy
no eligible warps to execute
Time
Throughput
Latency

- *Example 1:* Hide latency of single-precision FMA to maintain full arithmetic utilization
- V100 architecture considerations:
  - Each SM can have 4 Selected Warps/cycle, and maximum 64 Active Warps
  - Latency of single-precision FMA is 4 cycles

Latency Hiding Through Control of Warp Scheduling

Warp Scheduler 0 | Warp 0 | Warp 2 | Warp 3 | Warp 0 | Warp 4
Warp Scheduler 1 | Warp 1 | | Warp 6 | Warp 5
warp 0 waiting while SM still busy
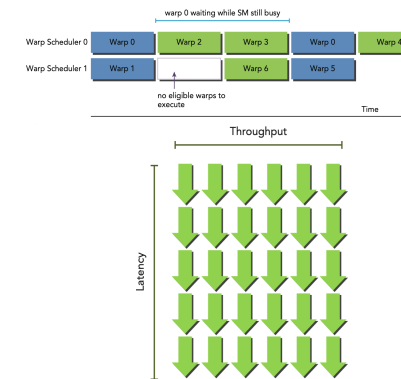no eligible warps to execute
Time
Throughput
Latency

- *Example 1:* Hide 4-cycle latency of single-precision (FP32) FMA to maintain full arithmetic utilization
- V100 architecture considerations:
  - Each SM can have 4 Selected Warps/cycle, and maximum 64 Active Warps
  - Latency of single-precision FMA is 4 cycles
- Throughput goal: 4 Selected Warps → 32 × 4 = 128 FMA/cycle (per SM)
- Latency is 4 cycles, Parallelism required is 128 ∗4 = 512 FP32 ops per cycle
- Number of Required Active Warps =

$$\frac{\#OpsPerCycle}{\#OpsPerWarp} = \frac{512}{32} = 16$$

Latency Hiding Through Control of Warp Scheduling

Warp Scheduler 0 | Warp 0 | Warp 2 | Warp 3 | Warp 0 | Warp 4
Warp Scheduler 1 | Warp 1 | | Warp 6 | Warp 5
warp 0 waiting while SM still busy
no eligible warps to execute
Time
Throughput
Latency

- *Example 2:* Hide global memory transaction latency to maintain peak bandwidth utilization
- V100 architecture considerations:
  - Global memory bandwidth ~800GB/s Latency of global memory transactions
  - ~500 cycles
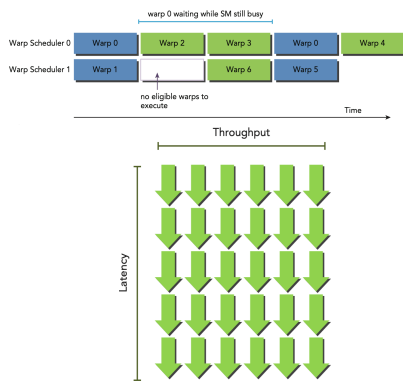 - HBM2 clock rate is 867 MHz

Latency Hiding Through Control of Warp Scheduling

- ➢ *Example 2:* Hide global memory transaction latency to maintain peak bandwidth utilization
- ➢ V100 architecture considerations:
  - ➢ Global memory bandwidth $\sim 800 GB/s$ Latency of global memory transactions
  - ➢ ~ 500 cycles
  - ➢ HBM2 clock rate is 867 MHz
  - ➢ Bandwidth per cycle =

$$\frac{800\ GB/s}{867\ Mhz} = 923\ B/cycle\ \text{(full GPU)}$$

- ➢ Required memory transaction volume = Bandwidth per cycle × Latency = 456.5 KB
- ➢ Transferring one `float` (4 Bytes) per GPU thread → 114,125 threads -> $\frac{114,125\ threads}{32\ threads/warp} \approx 3566\ warps$
- ➢ $\frac{3566\ warp}{84\ SM} = 43\ warp/SM$ → at least 43 active Warps to hide the latency
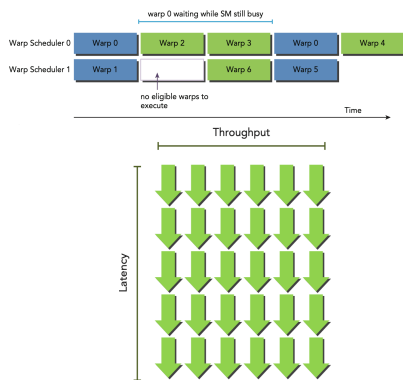

Can One Hide Latency? Beware Resource Limitations!
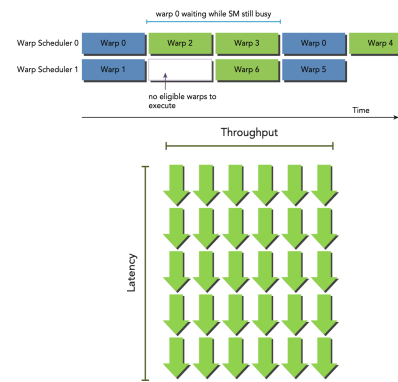
- ➢ Example 1, requires 16 Active Warps to hide the arithmetic latency.

- ➢ What are the register and shared memory limits per SM for this to be achievable?
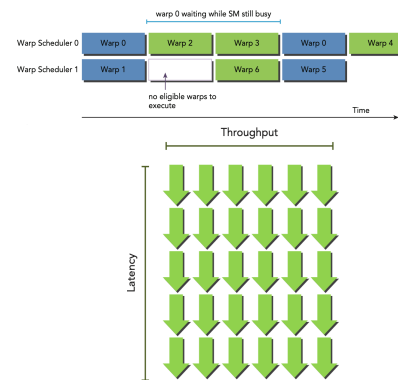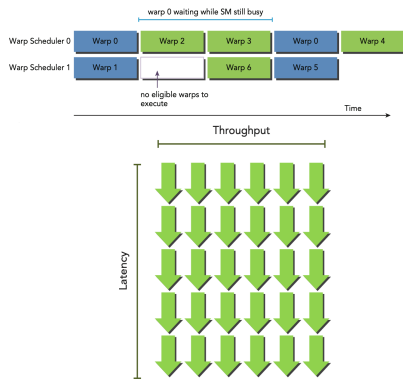

Can One Hide Latency? Beware Resource Limitations!

- ➢ In Example 1, one requires 16 Active Warps to hide arithmetic latency.
- ➢ What are the register and shared memory limits per SM in order for this to be achievable?
- ➢ V100 architecture considerations:
  - ➢ Register File Size/SM = 256 KB Shared Memory Size/SM = 64 KB (96 KB configurable)

Can One Hide Latency? Beware Resource Limitations!

- ➢ In Example 1, one requires 16 Active Warps to hide arithmetic latency.
- ➢ What are the register and shared memory limits per SM in order for this to be achievable?
- ➢ V100 architecture considerations:
  - ➢ Register File Size/SM = 256 KB of 32-bit registers
  - ➢ Shared Memory Size/SM = 64 KB
  - ➢ (96 KB configurable)
- ➢ $16\ warps \times 32\ thread = 512\ threads$
  $\frac{256\ KB/SM}{32\ bit/register} = 64,000\ registers/SM$
  $\frac{64,000\ registers/SM}{512\ threads} = 125\ registers/thread$
- ➢ If your kernel requires >125 registers/thread it cannot hide the arithmetic latency!

# Can One Hide Latency? Beware Resource Limitations!

- Example 1 requires 16 Active Warps to hide arithmetic latency.
- What are the register and shared memory limits per SM in order for this to be achievable?
- V100 architecture considerations:
  - Register File Size/SM = 256 KB of 32-bit registers
  - Shared Memory Size / SM = 64 KB
  - (96 KB configurable)
- What about Shared Memory?

Can One Hide Latency? Beware Resource Limitations!
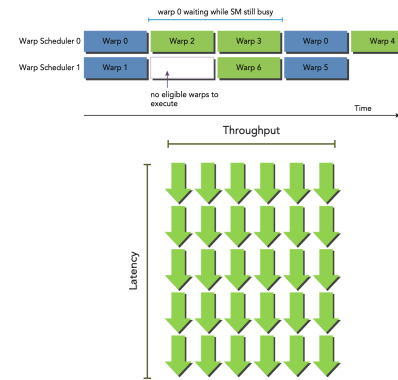
- What about Shared Memory?
- This depends on the *thread block size* and the amount of shared memory requested per thread block
- For example, thread block size = 128 and 64 KB of shared memory per block, can we hide the arithmetic latency?

Occupancy
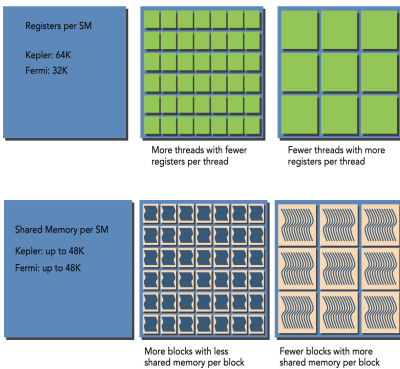
- You want to have sufficient warps to hide instruction latencies
- *Occupancy* = *ActiveWarps / MaximumWarps*
- It is a number $0 < Occupancy \leq 1$
- The higher the achieved occupancy the higher the chance your code will hide instruction latency
- This is just a rule of thumb. Can you achieve high performance with *Occupancy < 1*?

Guidelines for Grid and Block Sizes

- **Small thread blocks**: Too few threads per block leads to hardware limitations on the number of warps per SM to be reached before all resources are fully utilized.

- **Large thread blocks**: Too many threads per block leads to fewer per-SM hardware resources available to each thread

- In general, you should conduct experiments to discover the best execution configuration and resource usage. Some rules of thumb:

  - Keep the number of threads per block a multiple of warp size (32)
  - Avoid small block sizes: Start with at least 128 or 256 threads per block.
  - Adjust block size up or down according to kernel resource requirements.
  - Keep the number of blocks much greater than the number of SMs to expose sufficient parallelism to your device
  - Ask the compiler to print the number of registers using the flag `--ptxas-options=-v`. In case occupancy is register-limited try to optimize the number of registers per thread through the `nvcc` flag `-maxrregcount=NUM`.

Synchronization

In CUDA, synchronization can be performed at two levels:

- **System-level**: Wait for all work on both the host and the device to complete.
- **Block-level**: Wait for all threads in a thread block to reach the same point in execution on the device.
- `cudaError_t cudaDeviceSynchronize(void)` can be used to block the host application until all CUDA operations (copies, kernels, and so on) have completed
- `__device__ void __syncthreads(void)` can be used to **synchronize all theads within a block**:
  - Each thread in the same thread block must wait until all other threads in that thread block have reached this synchronization point
  - All global and shared memory accesses made by all threads prior to this barrier will
  - be visible to all other threads in the thread block after the barrier
- There is no thread synchronization among different blocks.
- GPUs can execute blocks in any order. This enables CUDA programs to be scalable across massively parallel GPUs.

## 2D Matrix Addition: Elapsed Time
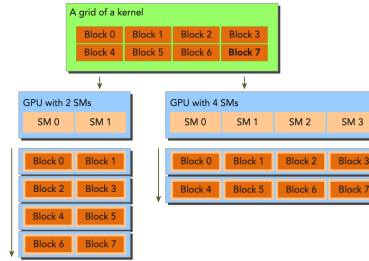
Matrix dimensions $nx = ny$ = 16,384

```
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) /
block.x, (ny + block.y - 1) /
block.y);
```

- Performance on an NVIDIA Tesla M2070 (Fermi):
  - Grid(xGDim, yGDim), Block(xBDim, yBDim) → elapsed time
  1. (512,512), (32,32) → 60 ms
  2. (512,1024), (32,16) → 38 ms
  3. (1024,512), (16,32) → 51 ms
  4. (1024,1024),(16,16) → 46 ms
- You can measure achieved warp occupancy by running
  - ncu –metrics achieved_ occupancy <application>

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```
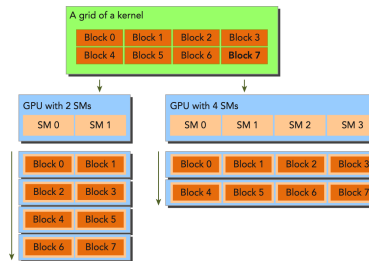

2D Matrix Addition: Achieved Occupancy
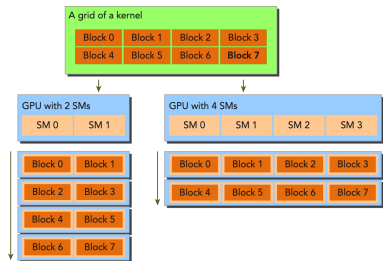
- Grid(xGDim, yGDim), Block(xBDim, yBDim) → elapsed time
  1. (512,512), (32,32) → 60 ms
  2. (512,1024), (32,16) → 38 ms
  3. (1024,512), (16,32) → 51 ms
  4. (1024,1024),(16,16) → 46 ms
- Achieved Occupancy
  1. (512,512), (32,32) → 0.50
  2. (512,1024), (32,16) → 0.74
  3. (1024,512), (16,32) → 0.77
  4. (1024,1024),(16,16) → 0.81

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```

2D Matrix Addition: Timings Versus Occupancy

- Elapsed time
  1. (512,512), (32,32) → 60 ms
  2. (512,1024), (32,16) → 38 ms
  3. (1024,512), (16,32) → 51 ms
  4. (1024,1024),(16,16) → 46 ms
- Achieved Occupancy
  1. (512,512), (32,32) → 0.50
  2. (512,1024), (32,16) → 0.74
  3. (1024,512), (16,32) → 0.77
  4. (1024,1024),(16,16) → 0.81

- Configuration "2" has more blocks than "1", this exposes more active warps to the device. This is likely why "2" has higher achieved occupancy and better performance than "1".
- Configuration "4" has the highest achieved occupancy, but it is not the fastest!
- Higher occupancy $\neq$ higher performance. There must be other factors that restrict performance.

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```

2D Matrix Addition: Memory Operations

- The kernel performs two memory loads, one memory store, one FLOP per thread.
- You can measure the global load throughput using
  - `ncu --metrics gld_throughput <application>`
  1. (512,512), (32,32) → 35.908GB/s
  2. (512,1024), (32,16) → 56.478GB/s
  3. (1024,512), (16,32) → 85.195GB/s
  4. (1024,1024),(16,16) → 94.708GB/s
- higher load throughput *does not guarantee* higher performance!

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```

2D Matrix Addition: Memory Operations

- Global load throughput
  1. (512,512), (32,32) → 35.908GB/s
  2. (512,1024), (32,16) → 56.478GB/s
  3. (1024,512), (16,32) → 85.195GB/s
  4. (1024,1024),(16,16) → 94.708GB/s
- The global load efficiency is the ratio of requested global load throughput to required global load throughput.
- You can measure it using
  `ncu --metrics gld_efficienct <application>`
  1. (512,512), (32,32) → 100 %
  2. (512,1024), (32,16) → 100 %
  3. (1024,512), (16,32) → 49.96 %
  4. (1024,1024),(16,16) → 49.80 %
- This explains why the higher load throughput and achieved occupancy of the last two cases did not yield improved performance.

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```

2D Matrix Addition: Memory Operations

- Global load throughput
  1. (512,512), (32,32) → 35.908GB/s
  2. (512,1024), (32,16) → 56.478GB/s
  3. (1024,512), (16,32) → 85.195GB/s
  4. (1024,1024),(16,16) → 94.708GB/s
- Global load efficiency
  1. (512,512), (32,32) → 100 %
  2. (512,1024), (32,16) → 100 %
  3. (1024,512), (16,32) → 49.96 %
  4. (1024,1024),(16,16) → 49.80 %
- The common feature for the last two cases is that their block size in the innermost dimension is half of a warp
- For grid and block heuristics, the innermost dimension should always be a multiple of the warp size

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY) {
        C[idx] = A[idx] + B[idx];
    }
}
```
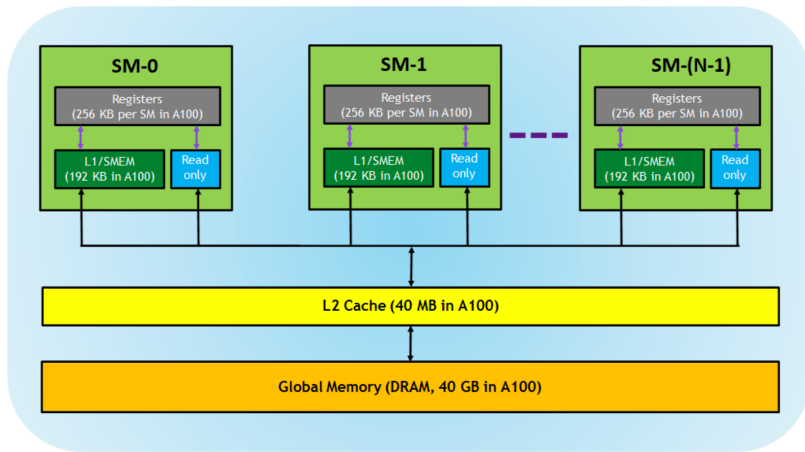
DEMO

2D Matrix Addition: Memory Operations

GPU (A100) Memory Hierarchy

CUDA Program