COMP4300 - Course Update

- **Final Exam**
  - Wednesday 11/06/2025 at 2:00pm at Copland G31 (Building 24)
  - The exam will cover all materials presented in the course e.g. in labs, lectures and assignments etc
  - Course/lecture notes permitted.

- **Assignment 2**
  - Released on 24 April
  - Due 26/05/2025, 11:55PM
  - Start early e.g. now

OpenMP with GPUs

OpenMP is traditionally used for parallel programming on CPUs, but recent versions have extended support to **GPUs** and other accelerators.

Target Directives

- #pragma omp target: Offloads a block of code to a GPU.
- #pragma omp target data: Manages data movement between host and device.
- #pragma omp target teams distribute parallel for: Enables fine-grained parallelism on the GPU.

Memory Management

- OpenMP handles memory transfers between host (CPU) and device (GPU) using map clauses.
- Example: map(to: a[0:N]) map(from: b[0:N])

Device Selection

- You can specify which device to use with device(n) clause.
- Useful in systems with multiple GPUs.

Unified Shared Memory (USM)

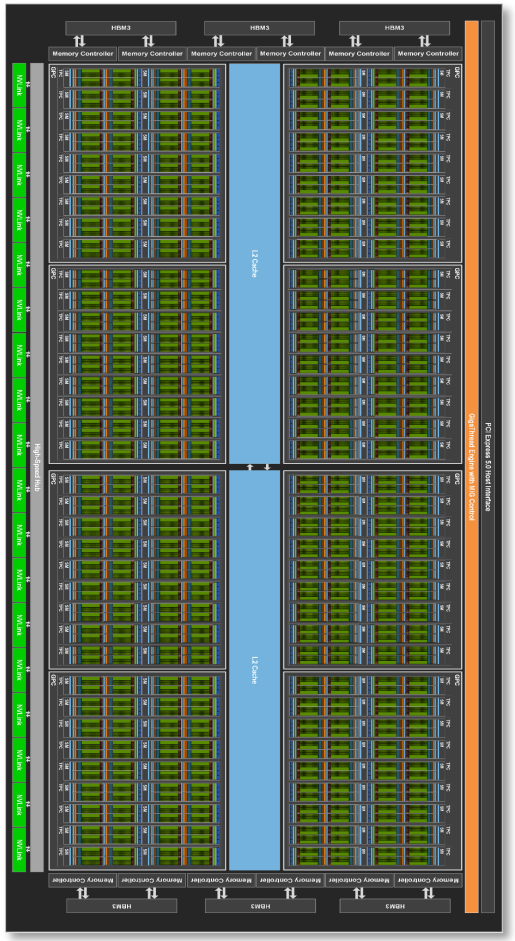
- Some implementations support shared memory between CPU and GPU, reducing the need for explicit data transfers.

OpenMP with GPUs

```
include <stdio.h>
#include <omp.h>
#define N 1000
int main() {
    float a[N], b[N], c[N];
    // Initialize arrays
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0f;
        b[i] = i * 2.0f;
    }
    // Offload computation to GPU
    #pragma omp target map(to: a[0:N], b[0:N]) map(from: c[0:N])
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
```

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**

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

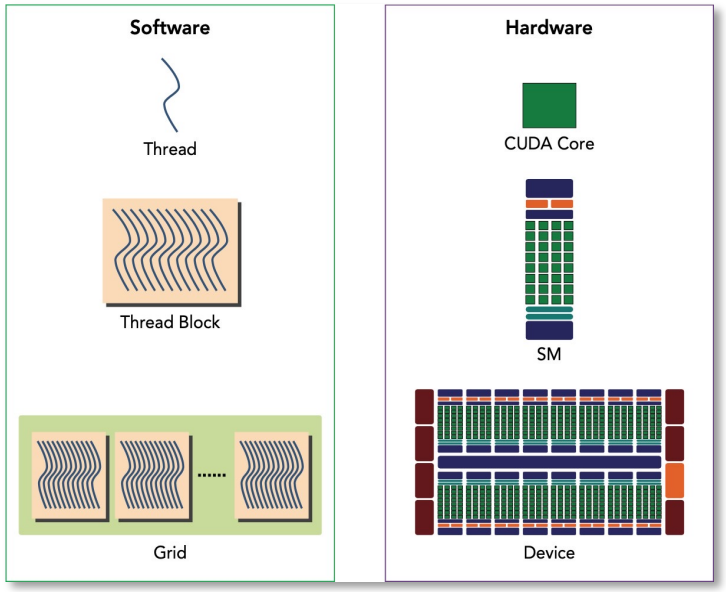
What is a TFLOP?

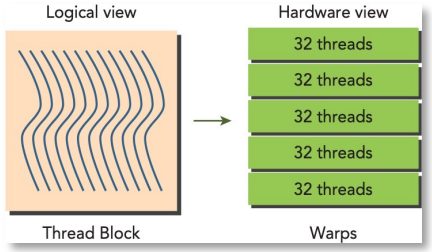
A TFLOP or Teraflop represents the ability to process **one trillion floating point operations per second.**

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.

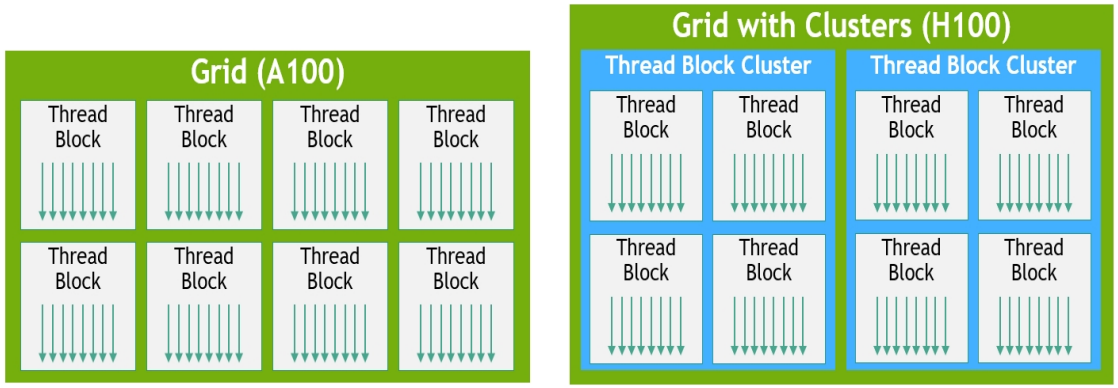
The Single Instruction Multiple Thread (SIMT) Model





- 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.

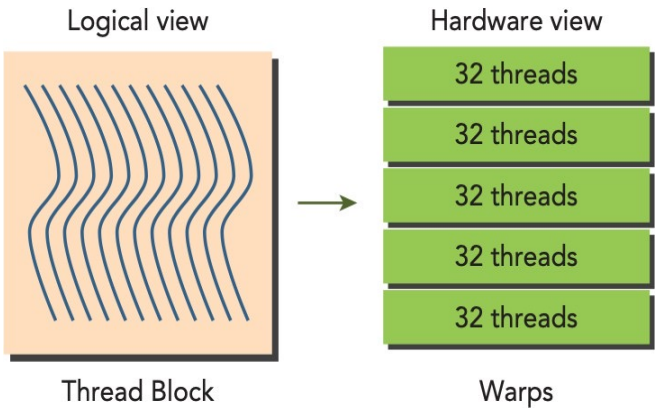
Thread Block Clusters and Grids with Clusters

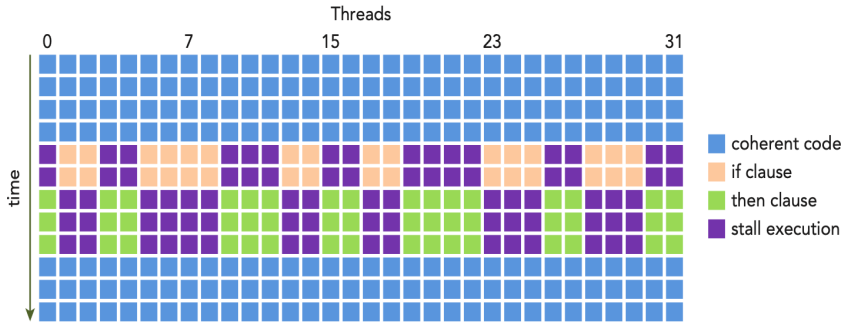


A Grid is composed of Thread Blocks in the legacy CUDA programming model as in A100, shown in the left half of the above diagram. The Hopper architecture adds an optional Cluster hierarchy, shown in the right half of the diagram.

- **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.**

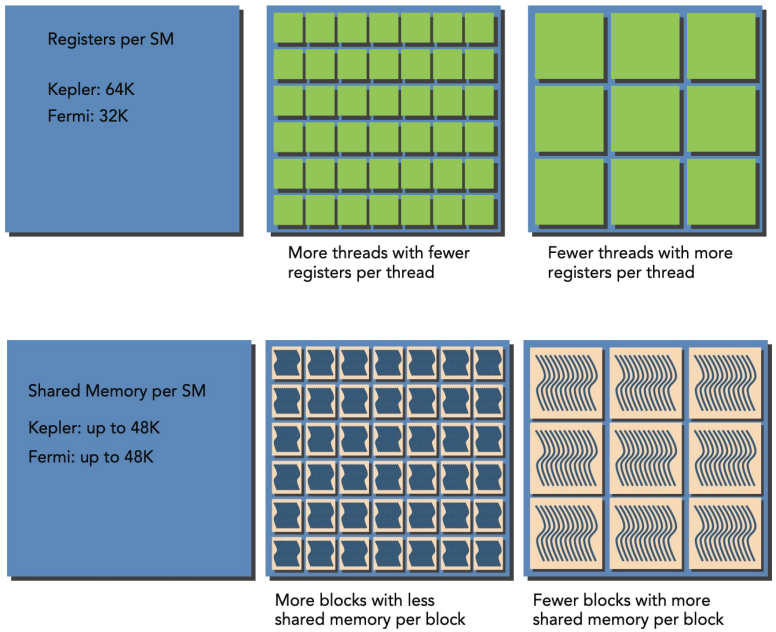
Warp Execution





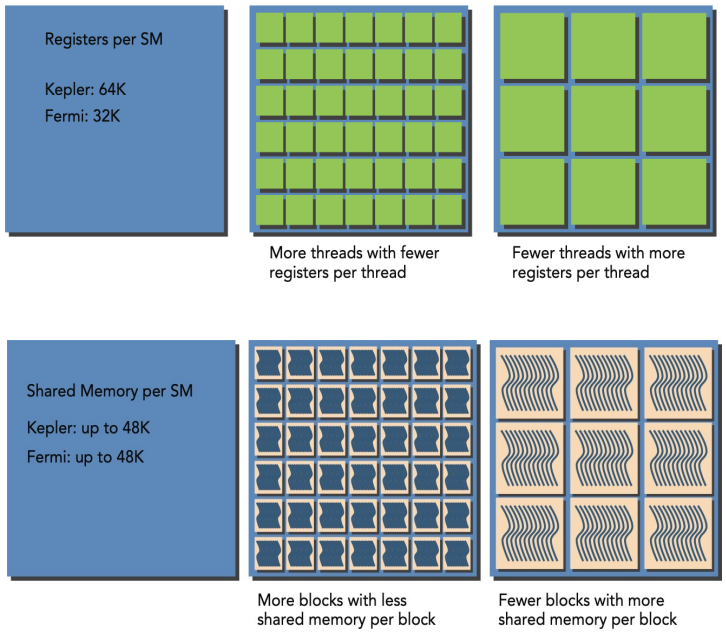
- A thread block is mapped to an SM and executed in warps
- The number of warps for a thread block can be determined as *ThreadsPerBlock*/32
- If thread block size is not an even multiple of warp size, some threads in the last warp are left inactive
- GPUs have very simple branch prediction mechanisms → 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



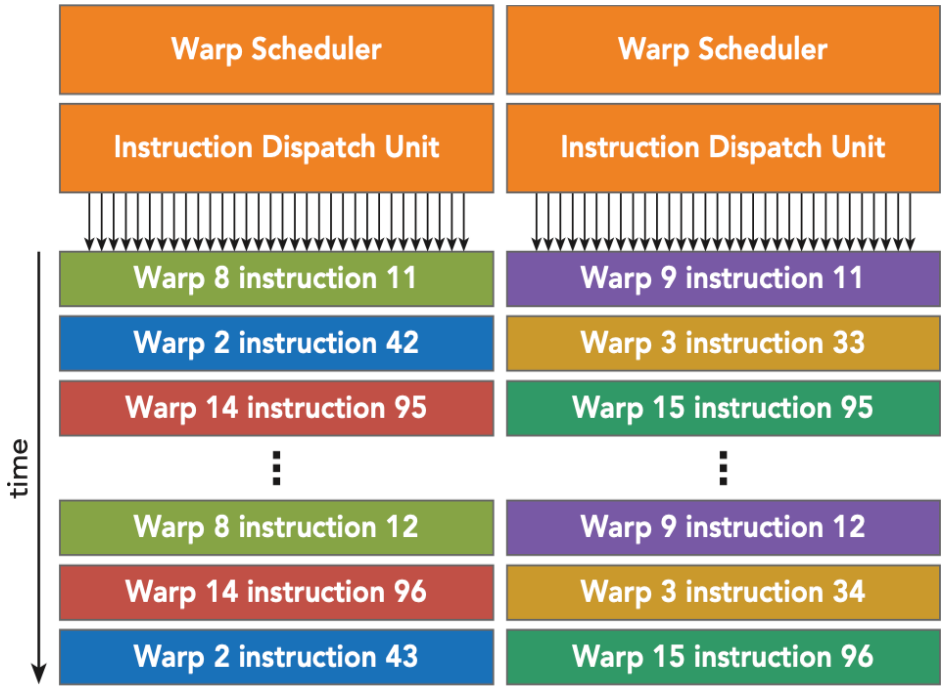
- The number of warps allocated to an SM depends on the resources it requires and affects performance significantly
- The local execution context of a warp mainly consists of program counters, registers and shared memory
- The execution context of each warp maintained on-SM during its lifetime of the 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 requirements change based on the kernel code

Warp Scheduling: Resource Limitations



- 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

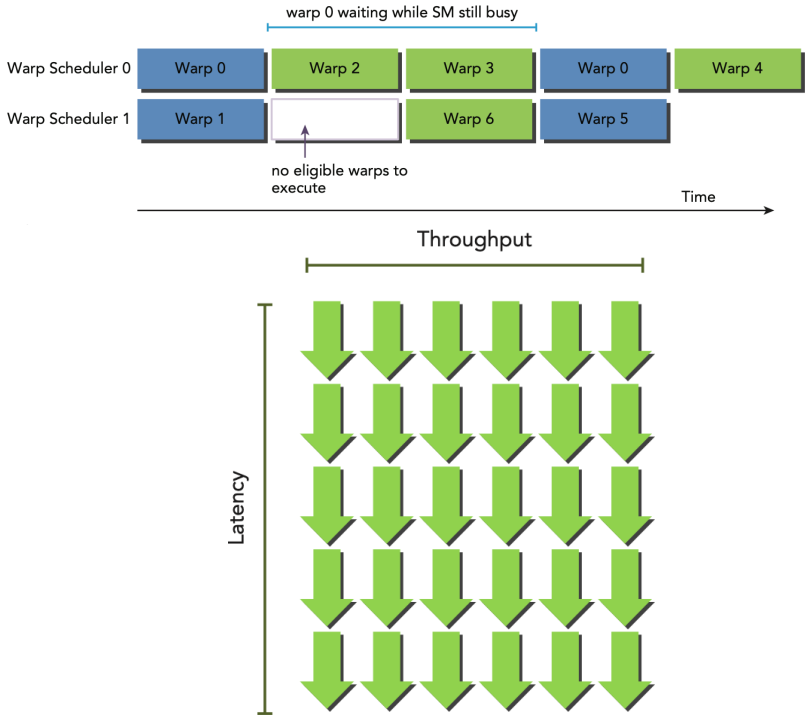
Latency Hiding Through Warp Scheduling



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

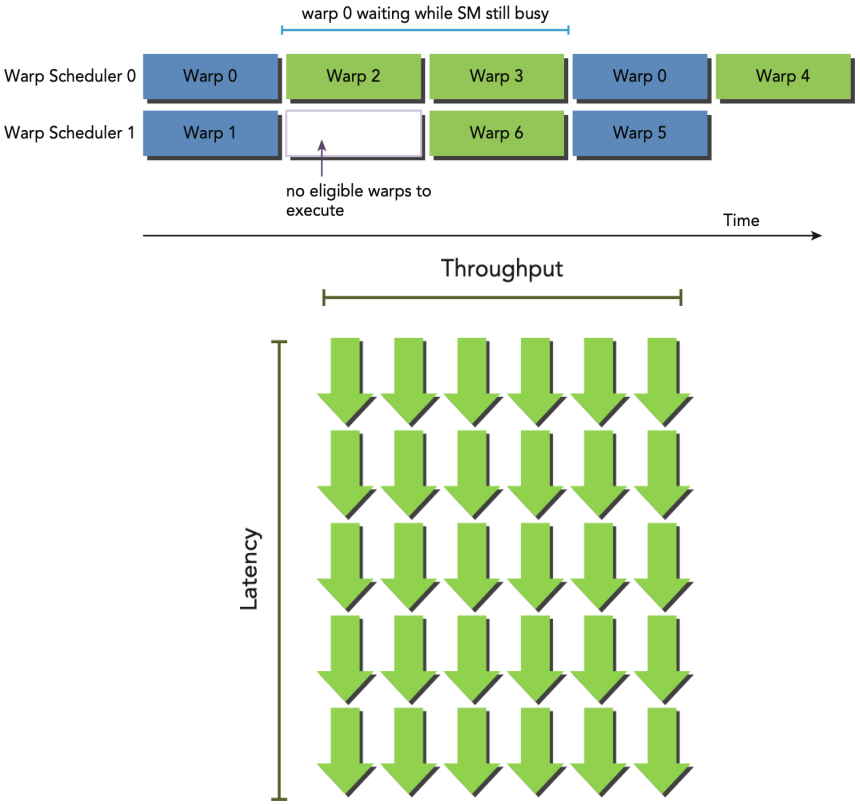


- The instruction latency can derive from either arithmetic or memory instructions
- Arithmetic instruction latency varies typically between 4 and 20 cycles
- Global memory instruction latency ~500 cycles for global memory accesses (uncached transactions)
- The number of active warps required to hide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

  $$\#RequiredWarps = Latency \times Throughput$$

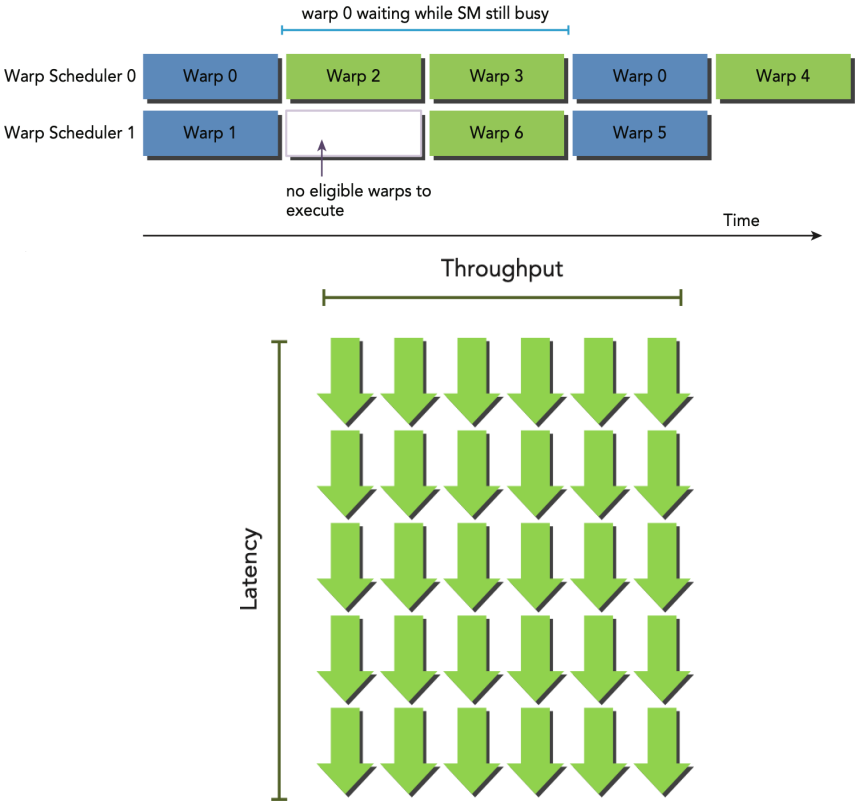
- Arithmetic operations: On H100 most single-precision ops have a latency of 4 cycles, while double-precision ones of 8 cycles
- Global memory operations: Latency ~500 cycles

Latency Hiding Through Control of Warp Scheduling



- ***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

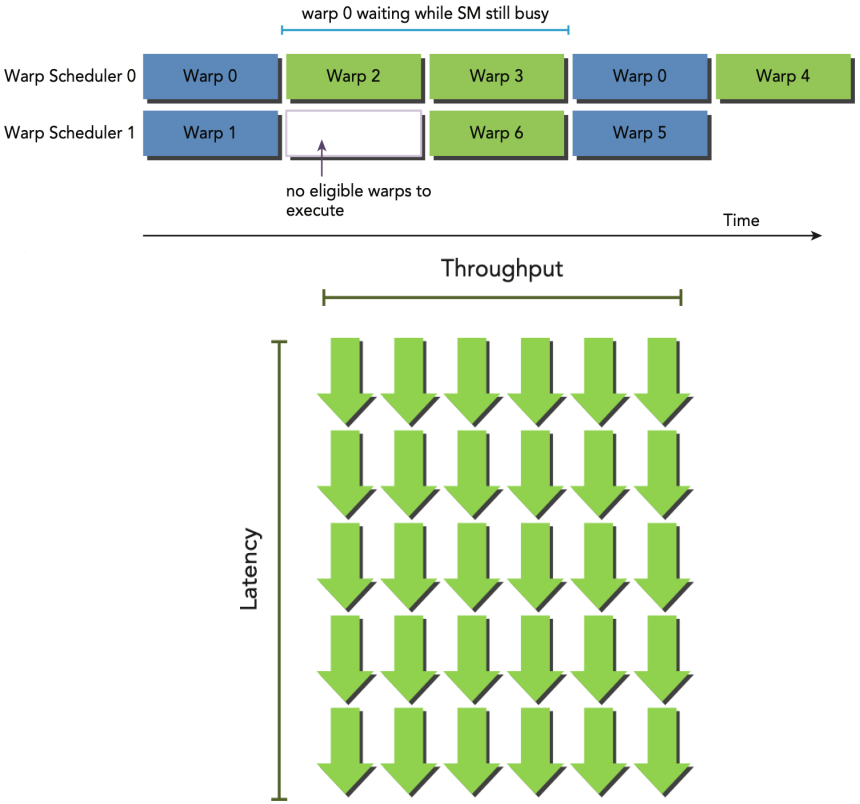
Latency Hiding Through Control of Warp Scheduling



- ***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 =

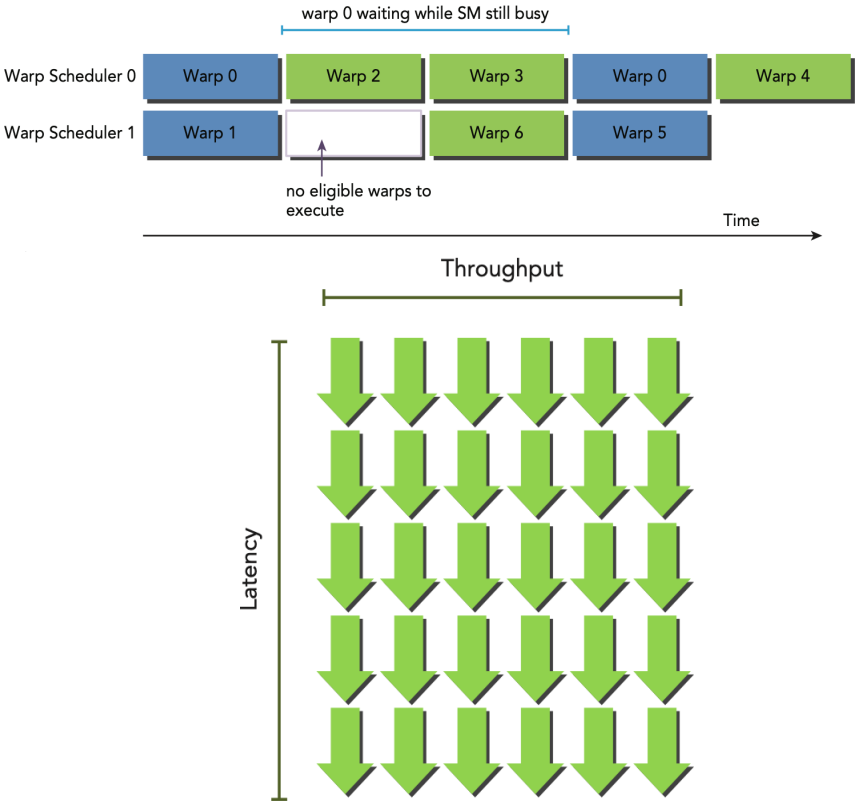
$$\frac{\#OpsPerCycle}{\#OpsPerWarp} = \frac{512}{32} = 16$$

Latency Hiding Through Control of Warp Scheduling



- ***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
  - $\sim$500 cycles
  - HBM2 clock rate is 867 MHz

Latency Hiding Through Control of Warp Scheduling

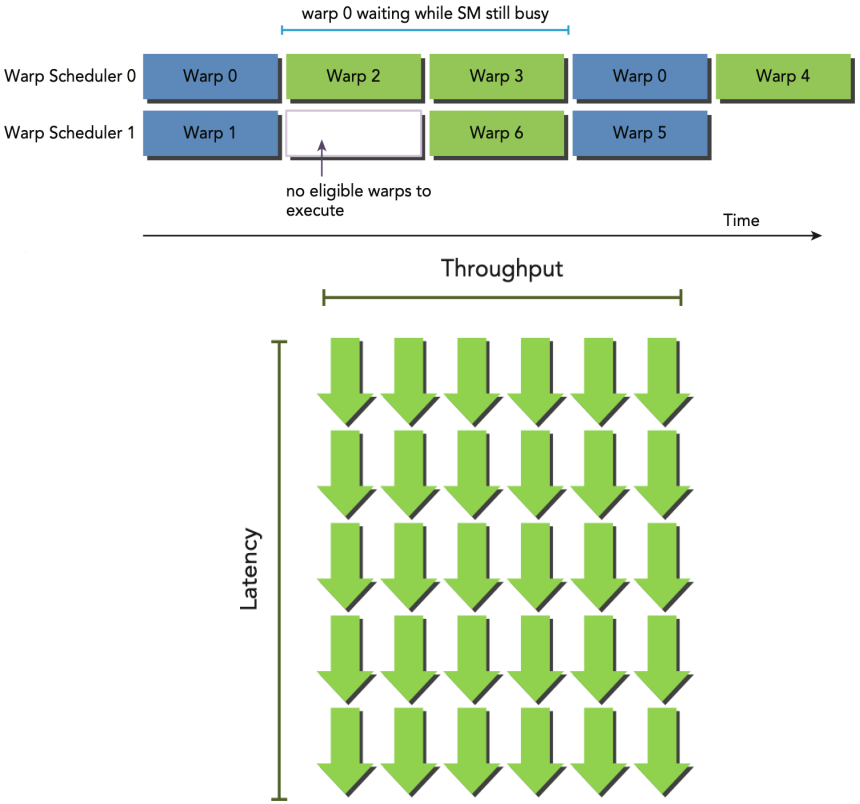


- ***Example 2:*** Hide global memory transaction latency to maintain peak bandwidth utilization
- V100 architecture considerations:
  - Global memory bandwidth $\sim 800 GB/s$ Latency of global memory transactions
  - $\sim$500 cycles
  - HBM2 clock rate is 867 MHz
  - Bandwidth per cycle =

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

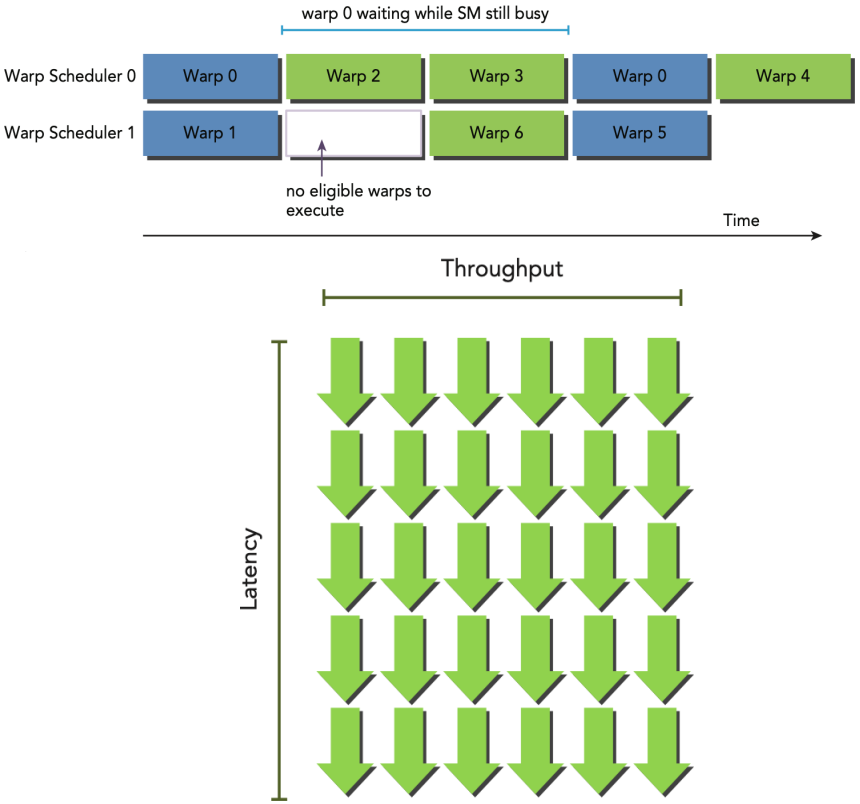
- Required memory transaction volume = Bandwidth per cycle × Latency = 456.5 KB
- Transferring one `float` (4 Bytes) per GPU thread → 114,125 threads -> $\frac{114,125 threads}{32\ threads/warp} \approx$ 3566 warps
- $\frac{3566\ warp}{84\ SM} = 43\ warp/SM \rightarrow$ at least 43 active Warps to hide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

Can One Hide Latency? Beware Resource Limitations!



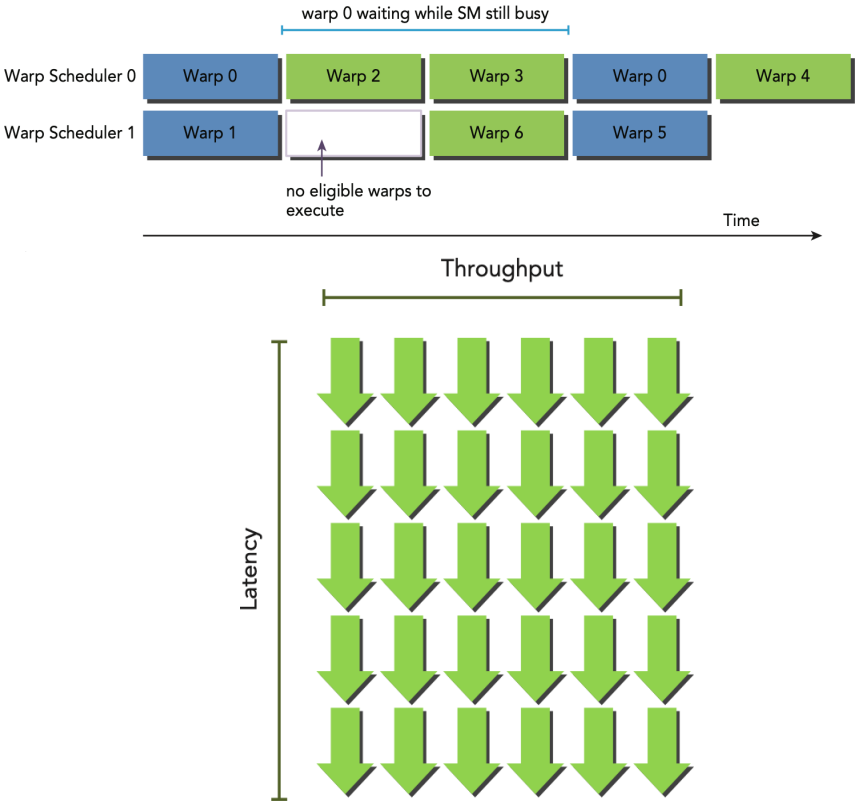
- Example 1, requires 16 Active Warps to hide the arithmetic latency.
- What are the register and shared memory limits per SM for this to be achievable?

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!



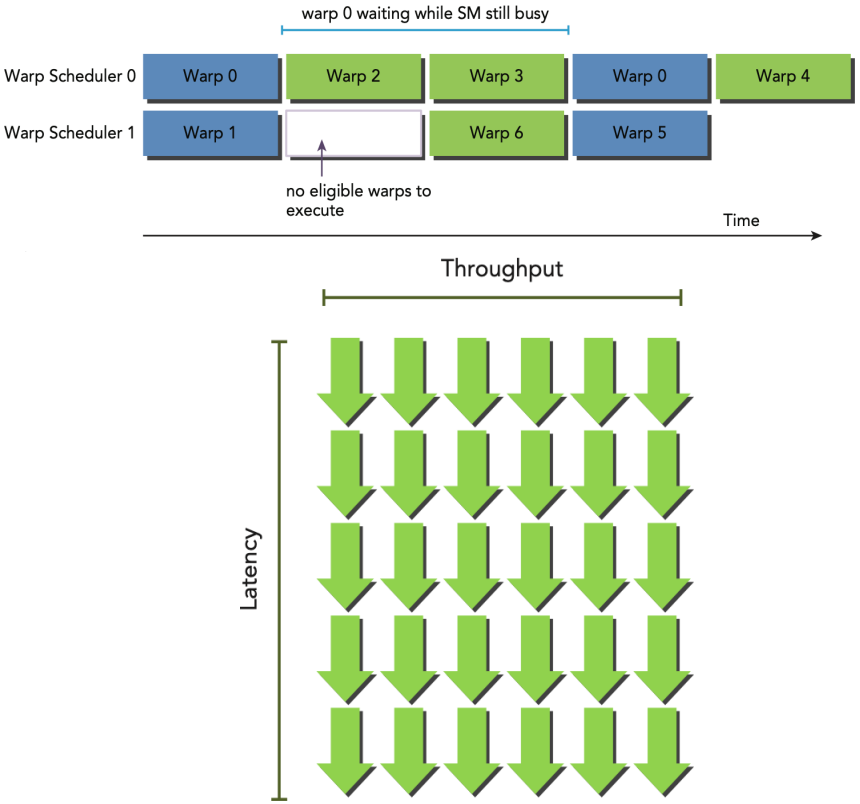
- In Example 1, one requires 16 Active Warps to hide arithmetic latency.
- What are the register and shared memory limits per SM in order for this to be achievable?
- V100 architecture considerations:
  - Register File Size/SM = 256 KB of 32-bit registers
  - Shared Memory Size/SM = 64 KB
  - (96 KB configurable)
- 16 *warps* × 32 *thread* = 512 *threads*

$$\frac{256\ KB/SM}{32\ bit/register} = 64{,}000\ registers/SM$$

$$\frac{64{,}000\ registers/SM}{512\ threads} = 125\ registers/thread$$

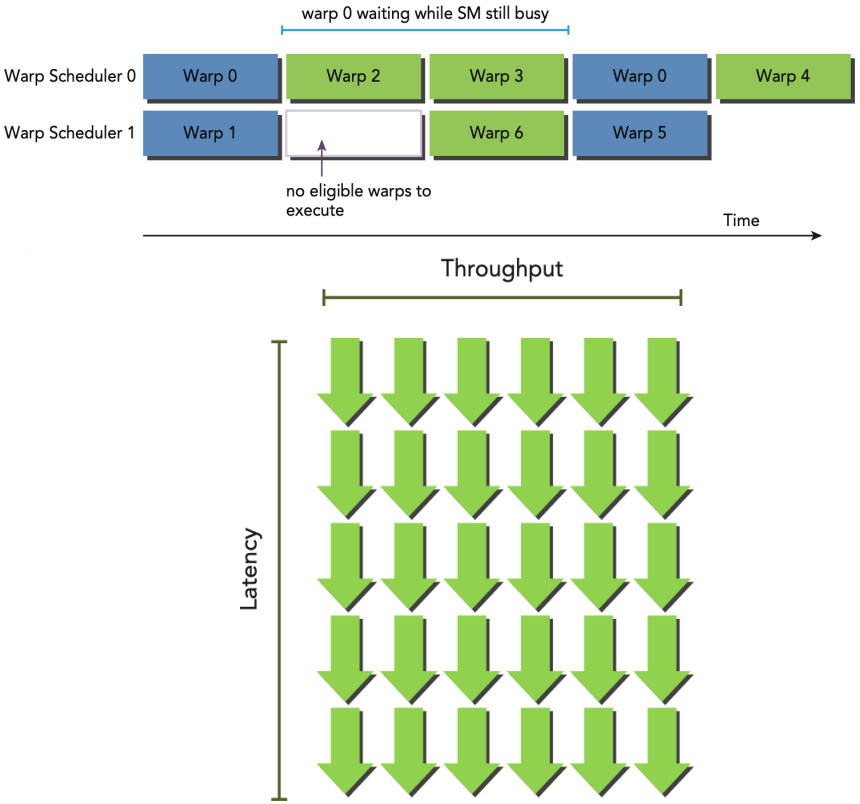
  - If your kernel requires >125 registers/thread it cannot hide the arithmetic latency!

Can One Hide Latency? Beware Resource Limitations!



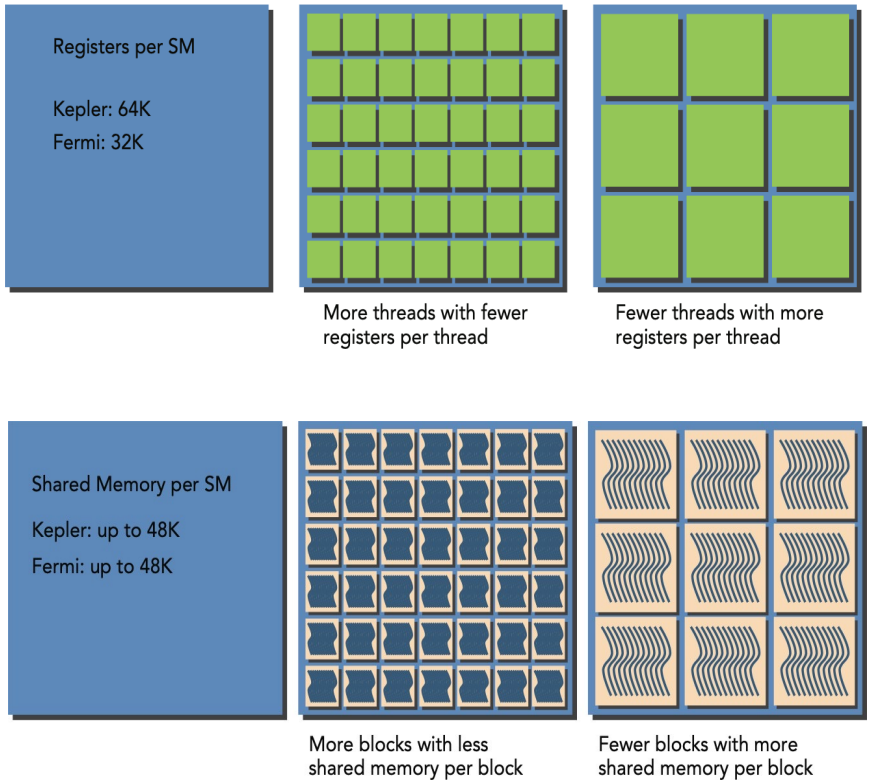
- *Example 1 requires 16 Active Warps to hide arithmetic latency.*
- *What are the register and shared memory limits per SM in order for this to be achievable?*
- V100 architecture considerations:
  - Register File Size/SM = 256 KB of 32-bit registers
  - Shared Memory Size / SM = 64 KB
  - (96 KB configurable)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

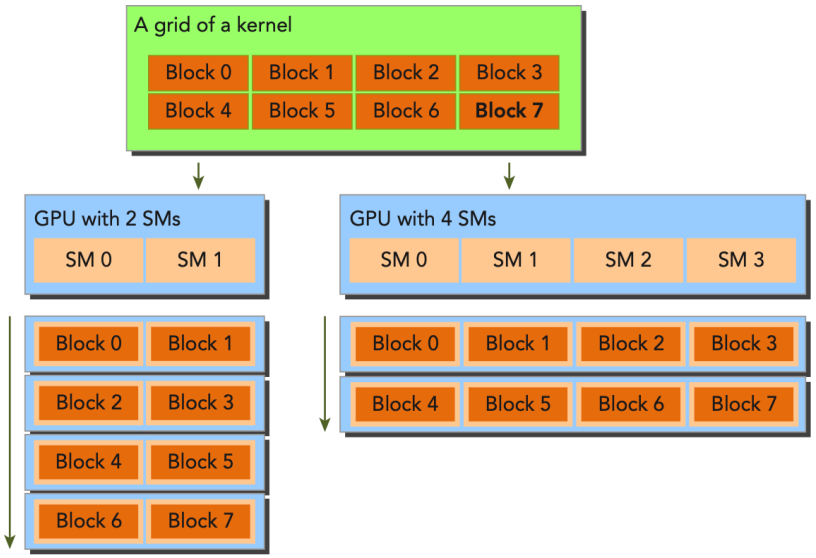
  - Keep the number of threads per block a multiple of warp size (32)
  - Avoid small block sizes: Start with at least 128 or 256 threads per block.
  - Adjust block size up or down according to kernel resource requirements.
  - Keep the number of blocks much greater than the number of SMs to expose sufficient 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



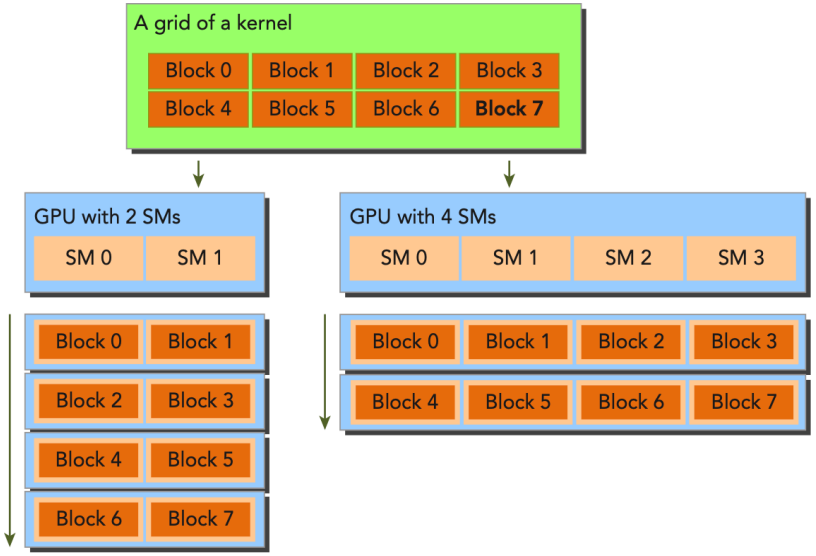

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

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

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>`

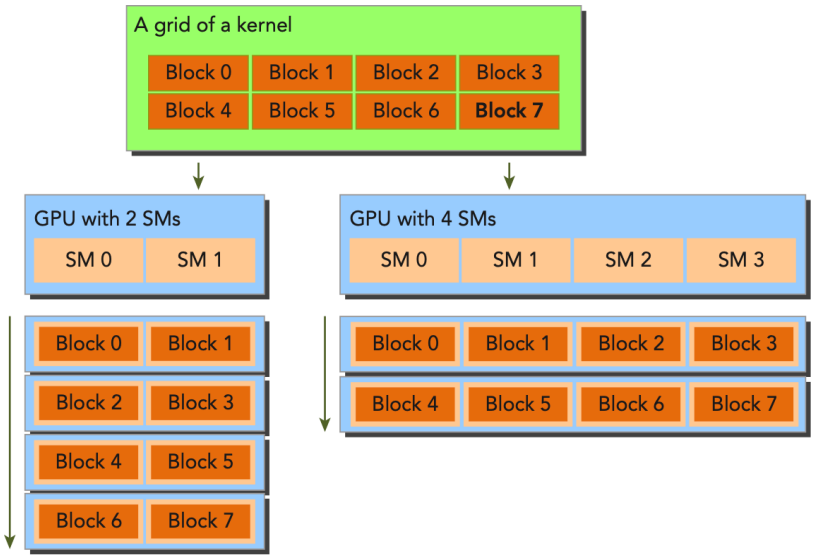
2D Matrix Addition: Achieved Occupancy




```
__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];
    }
}
```

- 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

2D Matrix Addition: Timings Versus Occupancy

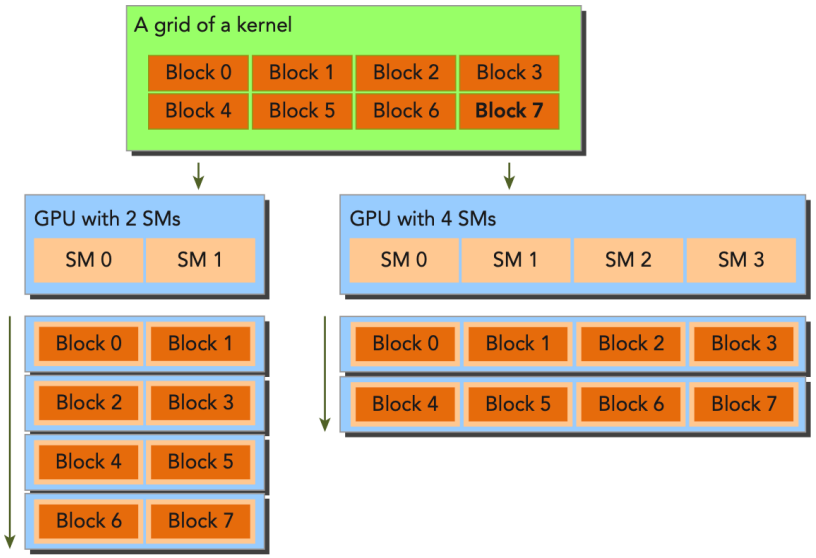



```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

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

- 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 ≠ higher performance. There must be other factors that restrict performance.

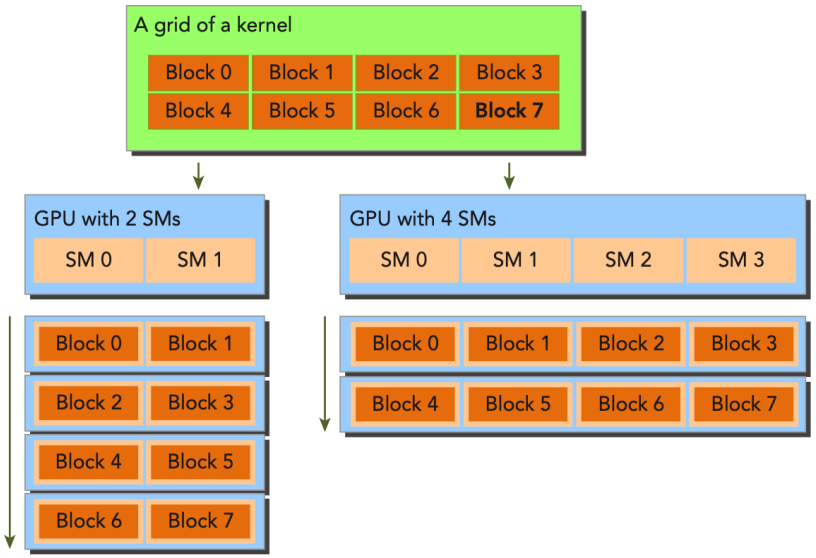
2D Matrix Addition: Memory Operations




```
__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];
    }
}
```

- 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!

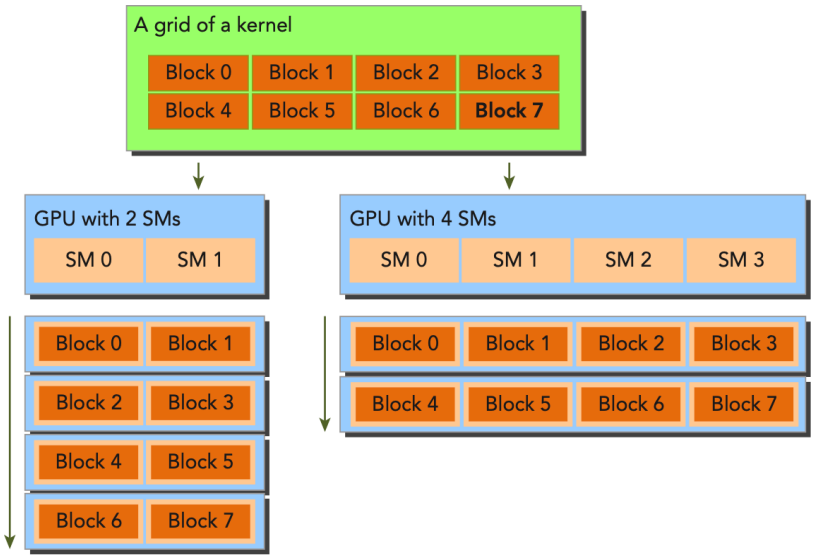
2D Matrix Addition: Memory Operations




```
__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];
    }
}
```

- 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_efficient <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.

2D Matrix Addition: Memory Operations



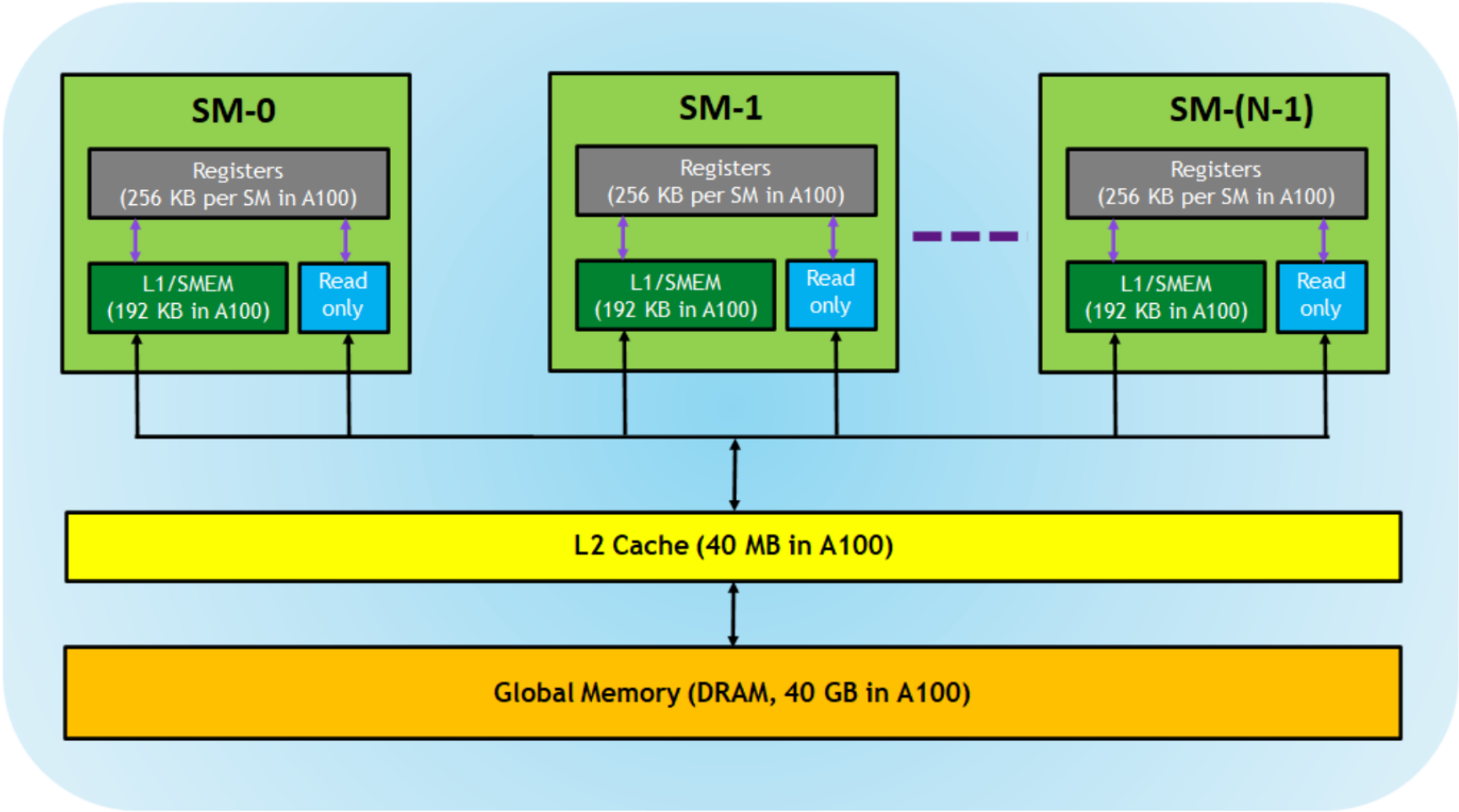

```
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int
NX, int NY) {
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

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

- 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

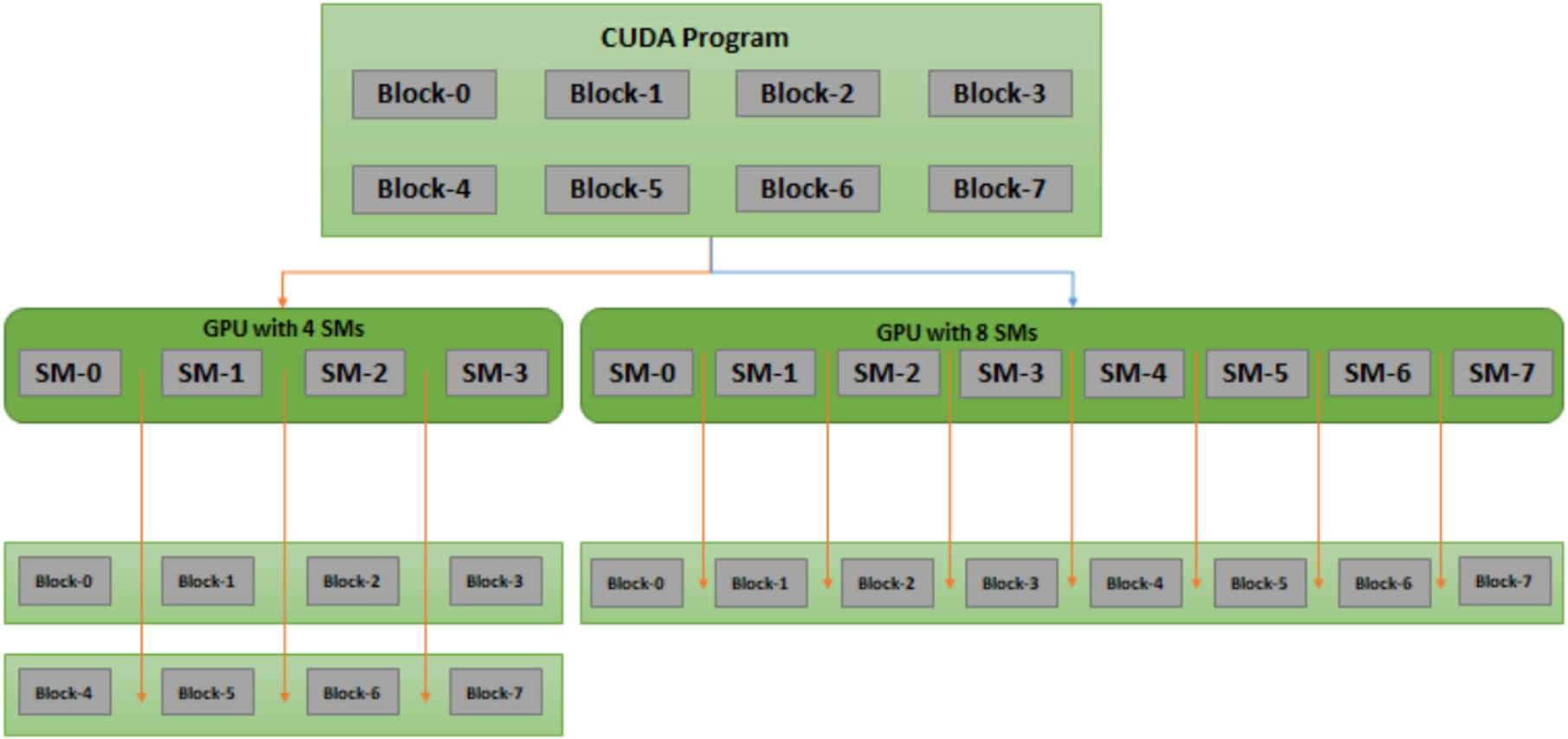
DEMO

2D Matrix Addition: Memory Operations

GPU (A100) Memory Hierarchy



CUDA Program



NVIDIA V100 GPU AT NCI - 2D MATRIX SUM

Device	Version	Grid	Block	Time	Speedup	Occupancy	Memory
CPU	matrix-add-cpu	Nx, Ny = 32768	N/a	31,366 ms	1		
CPU	matrix-add-openmp-avx	Nx, Ny = 32768	N/A	3302 ms	9.5		
CPU	matrix-add-openmp-gcc	Nx, Ny = 32768	N/A	550 ms	57		
GPU	matrix-add-gpu	1024 x 1024	32 x 32	19.23 ms	1631	83.39%	85.37%
GPU	matrix-add-gpu	1024 x 2048	32 x 16	**18.40 ms**	**1704**	86.5%	**90.15%**
GPU	matrix-add-gpu	2048 x 1024	16 x 32	21.38 ms	1467	87.1%	82.07%
GPU	matrix-add-gpu	2048 x 2048	16 x 16	18.73 ms	1674	**90.37%**	87.89%

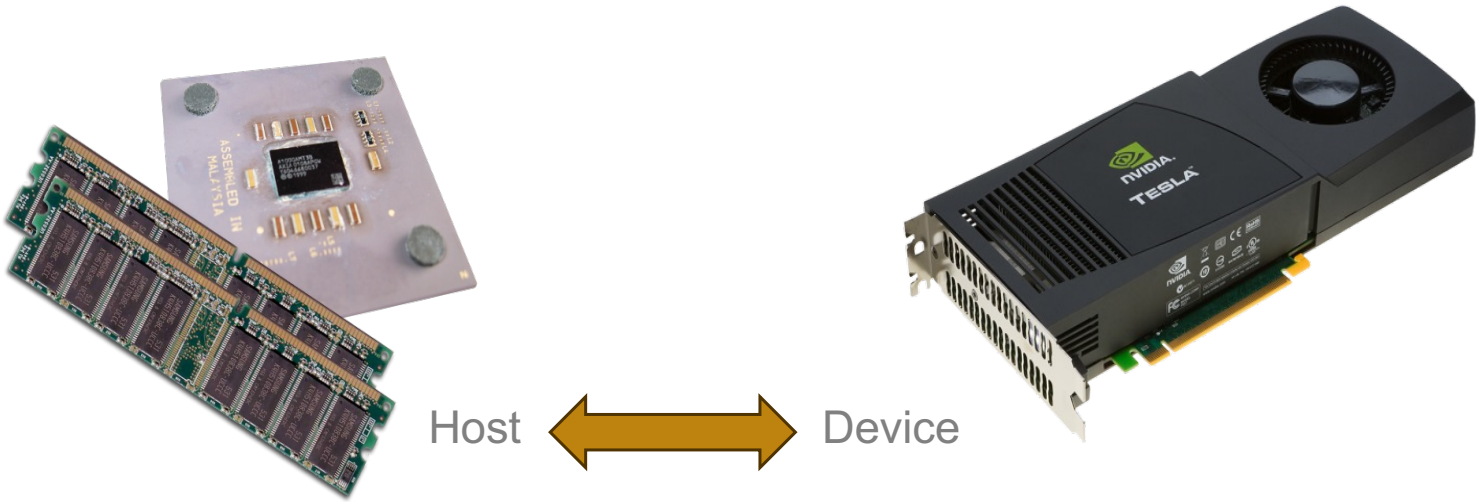
Heterogeneous Computing

Reference Material

- NVIDIA's CUDA C++ Best Practices Guide, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Nvidia H100 TensorCore GPU Architecture https://resources.nvidia.com/en-us-tensor-core
- Jia, Z., Maggioni, M., Staiger, B., & Scarpazza, D. P. (2018). *Dissecting the NVIDIA volta GPU architecture via microbenchmarking*. arXiv preprint arXiv:1804.06826.
- *Professional CUDA c programming*. Cheng, John, Max Grossman, and Ty McKercher. John Wiley & Sons, 2014.
- *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Sanders, Jason, and Edward Kandrot, Addison-Wesley Professional, 2010.
- Tesla V100 Performance Optimization Guide, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/ tesla-product-literature/v100-application-performance-guide.pdf

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Heterogeneous Computing

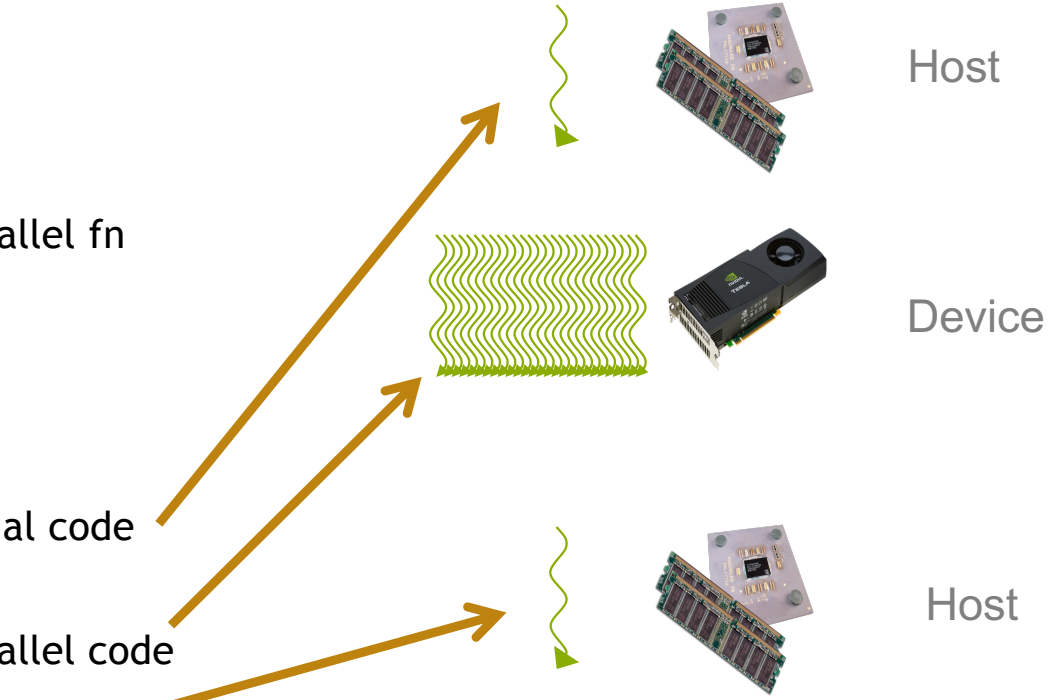
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

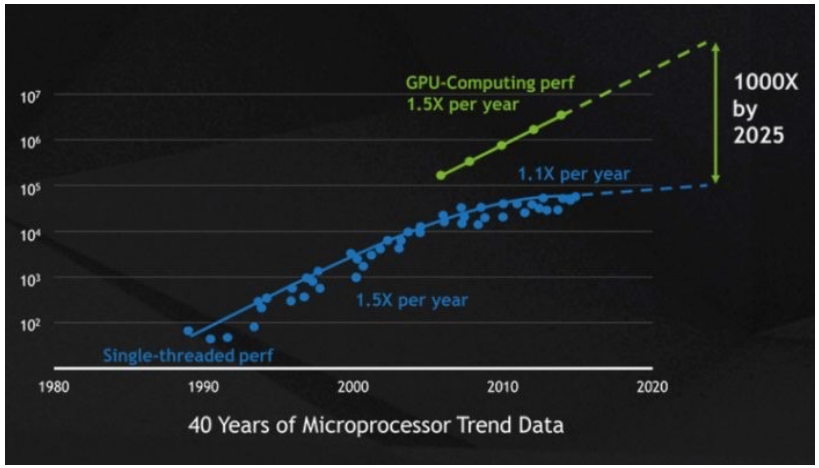
serial code

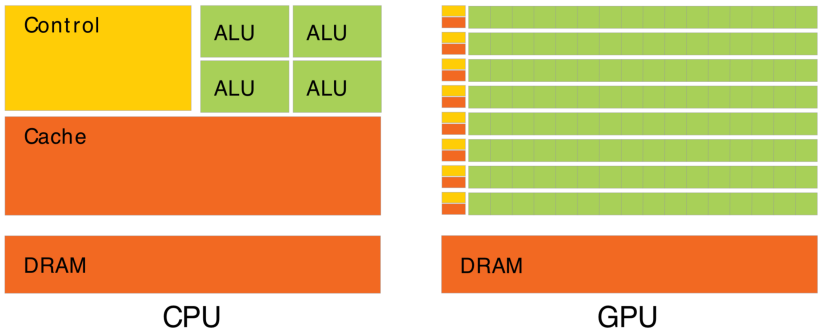
parallel code

serial code



The End of The Road for General-Purpose Processors

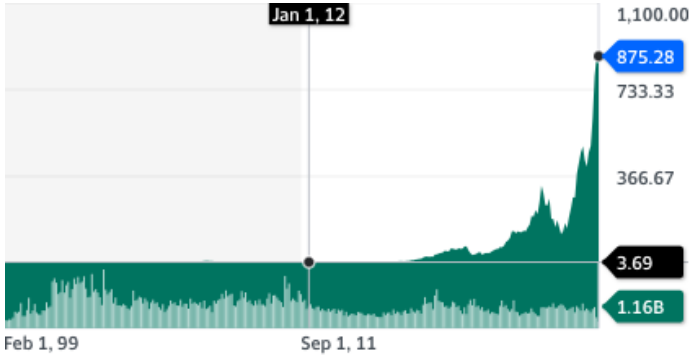




- End of Dennard scaling caused the end of the general-purpose processor era (both uniprocessor and multicore)
- Use of domain specific architectures (DSAs): programmable but designed for a class of problems with specific structures.
- GPUs are designed for data-parallel algorithms (especially linear algebra)
- More transistors are devoted to data processing rather than data caching and flow control
- Require domain specific programming model that makes it possible for the software to match the hardware (e.g. CUDA)
- Extracting performance requires the programmer to expose parallelism, to manage memory efficiently (e.g. caching), to tailor the algorithm to the hardware

TOP 500 List November 2023

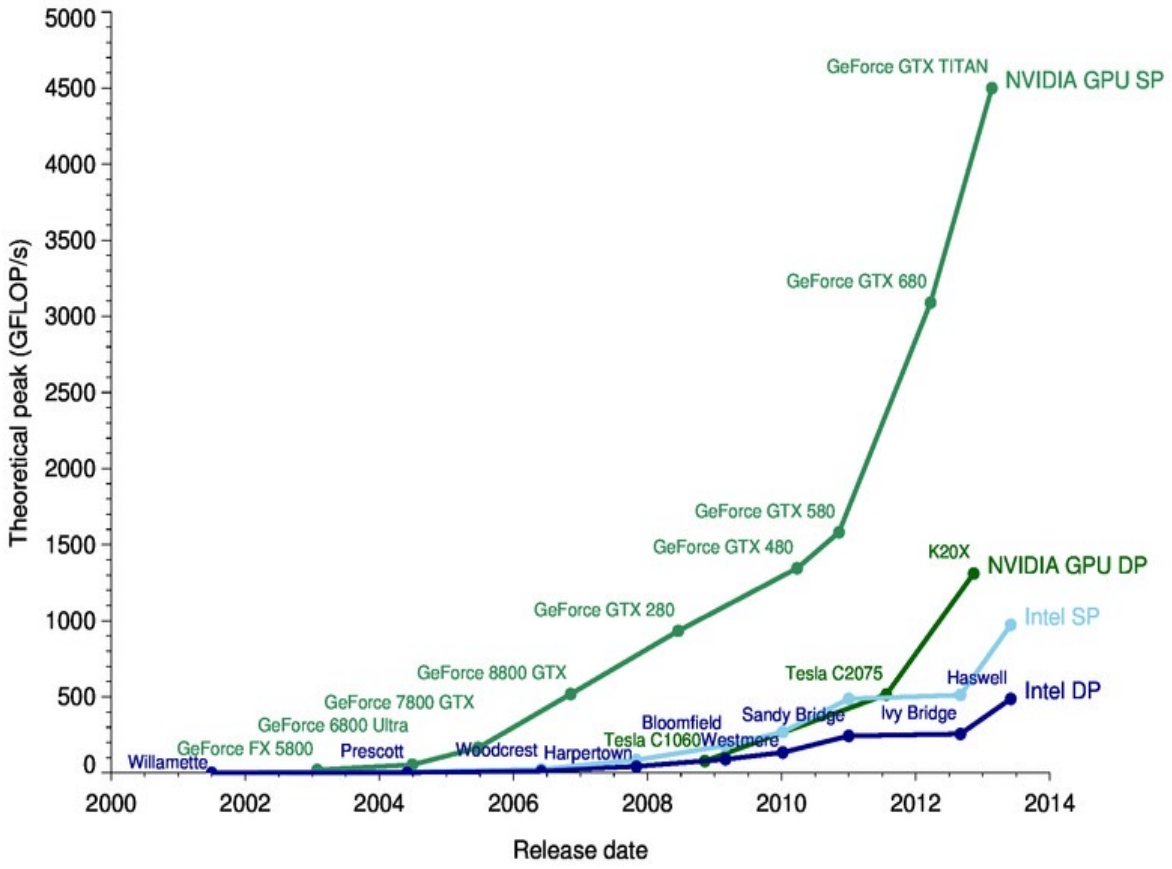
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	4,742,808	585.34	1,059.33	24,687
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Microsoft Azure United States	1,123,200	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107



TOP 500
The List.

https://www.top500.org/

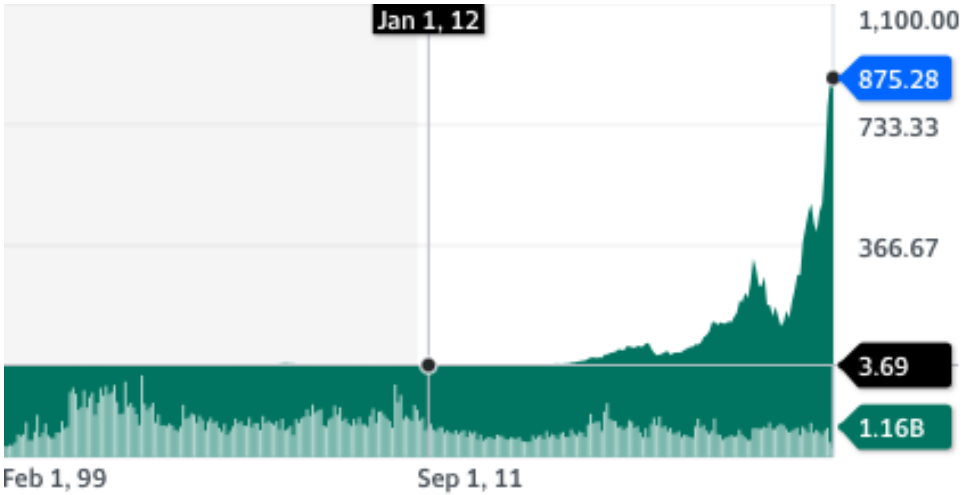
CPU versus GPU - FLOP rates

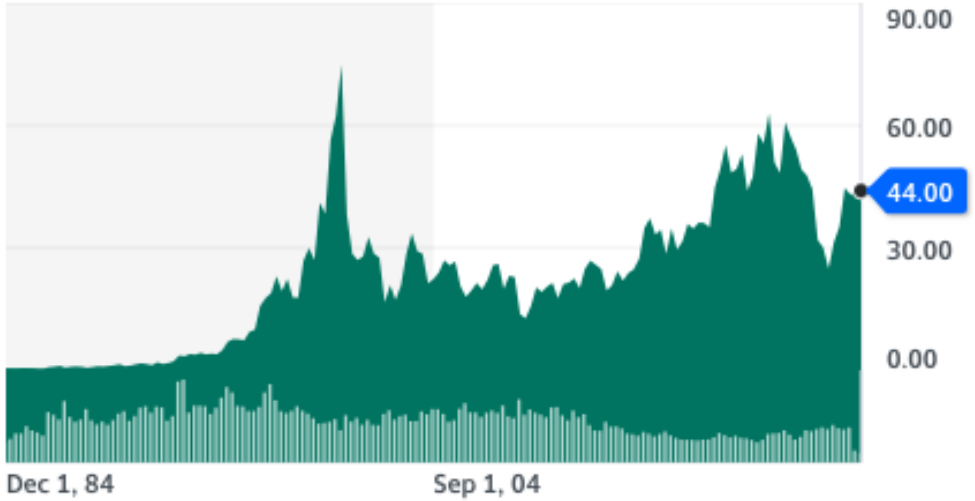


GPU FLOP Rates have been growing exponentially:-

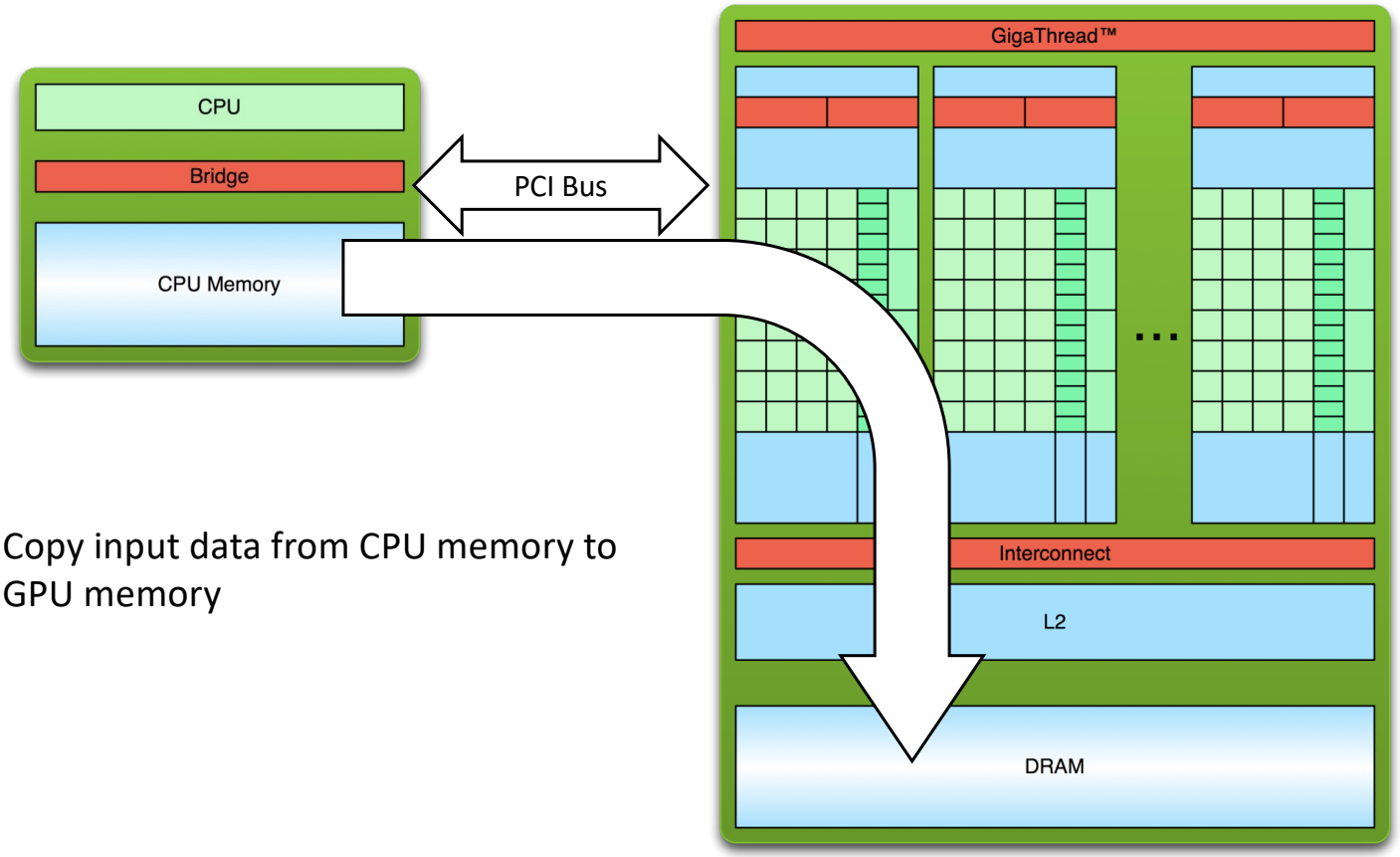
- 2010's GFLOP/s – see the graph opposite
- 2020's TFLOPS/s to PFLOP/s e.g. H100 GPU

Impact of Heterogenous Computing



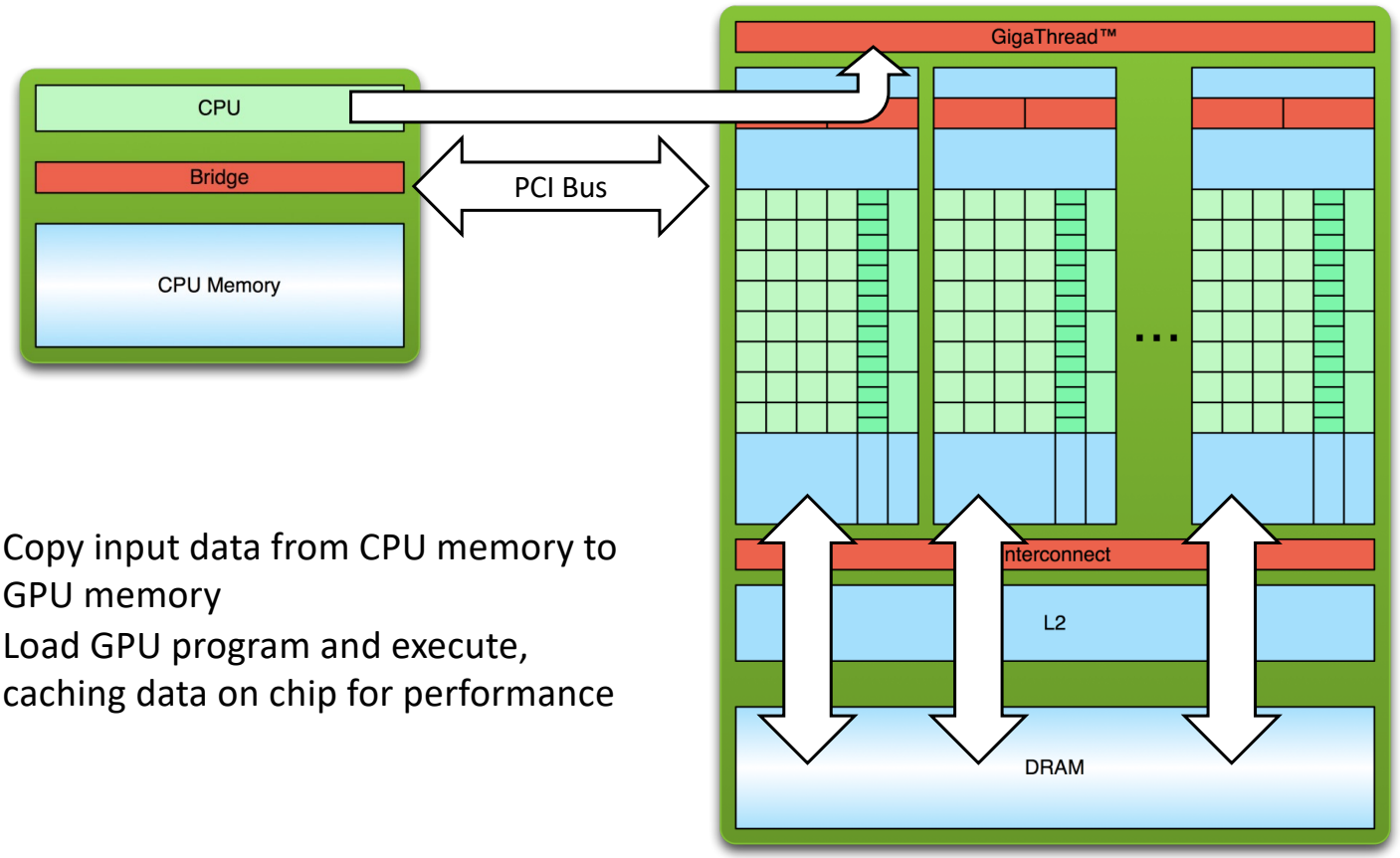


Simple Processing Flow



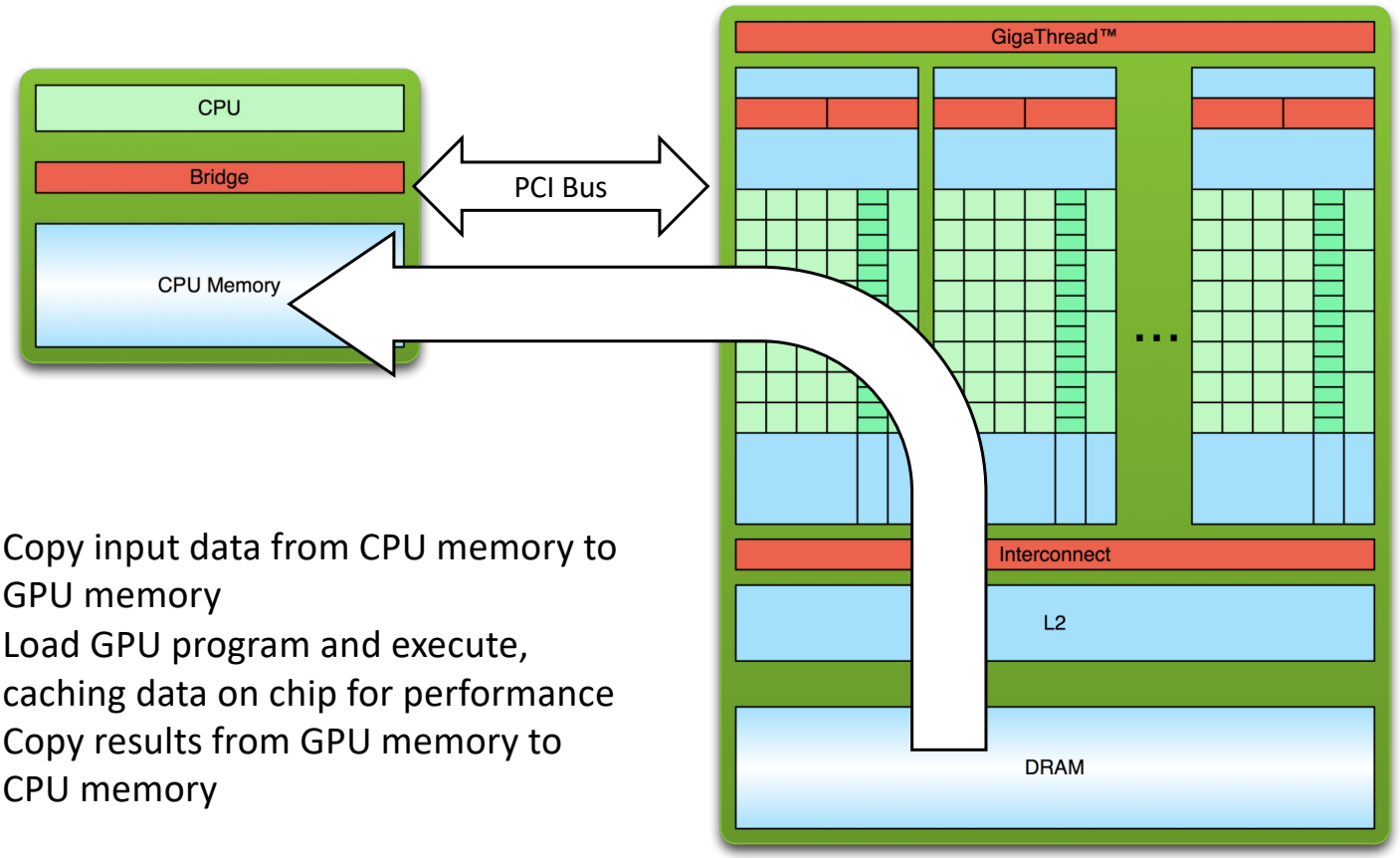
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



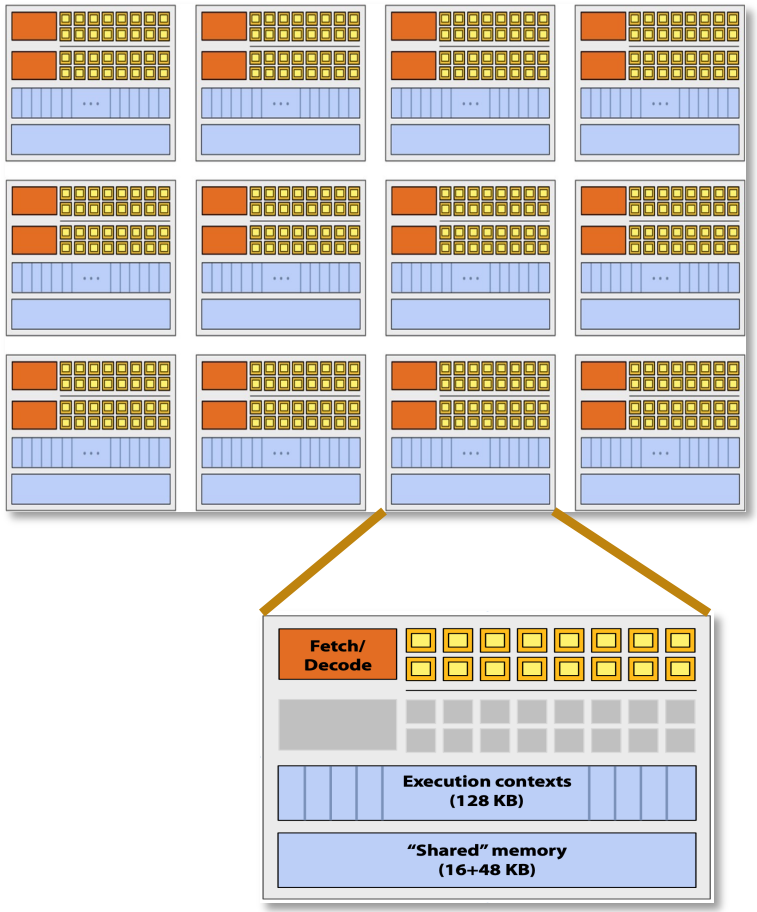
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

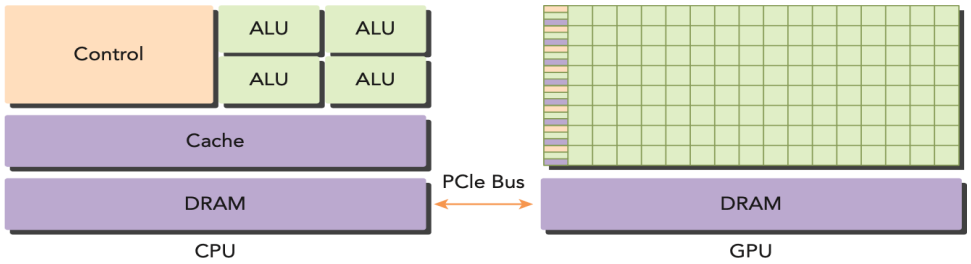
A Simplistic View of the GPU Architecture

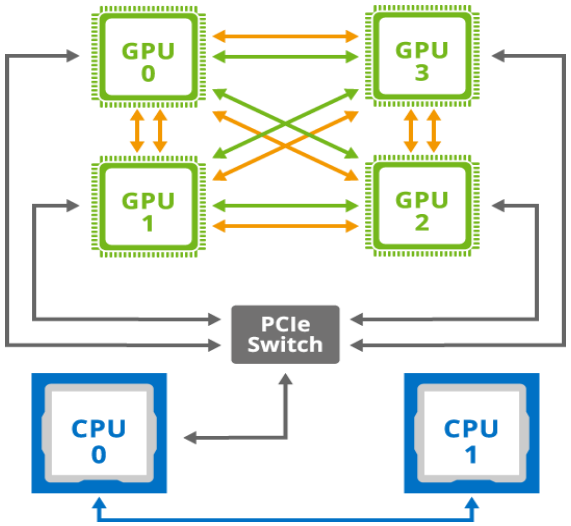


A scalable array of complex "cores" called Streaming Multiprocessors (SM)

- Each core has an array of functional units (e.g. ALUs) with SIMD execution
- Instructions operate in groups of 32 "SIMD" threads called warps
- On the NVIDIA H100 GPU up to 64 warps can be executed concurrently (interleaved) on a single SM
- Up to 132 SMs × 128 CUDA cores/SM = 16896 Cuda cores per device
- H100 includes Tensor cores + Transformer engine for training large language models
- This is why GPUs are called throughput-oriented architectures

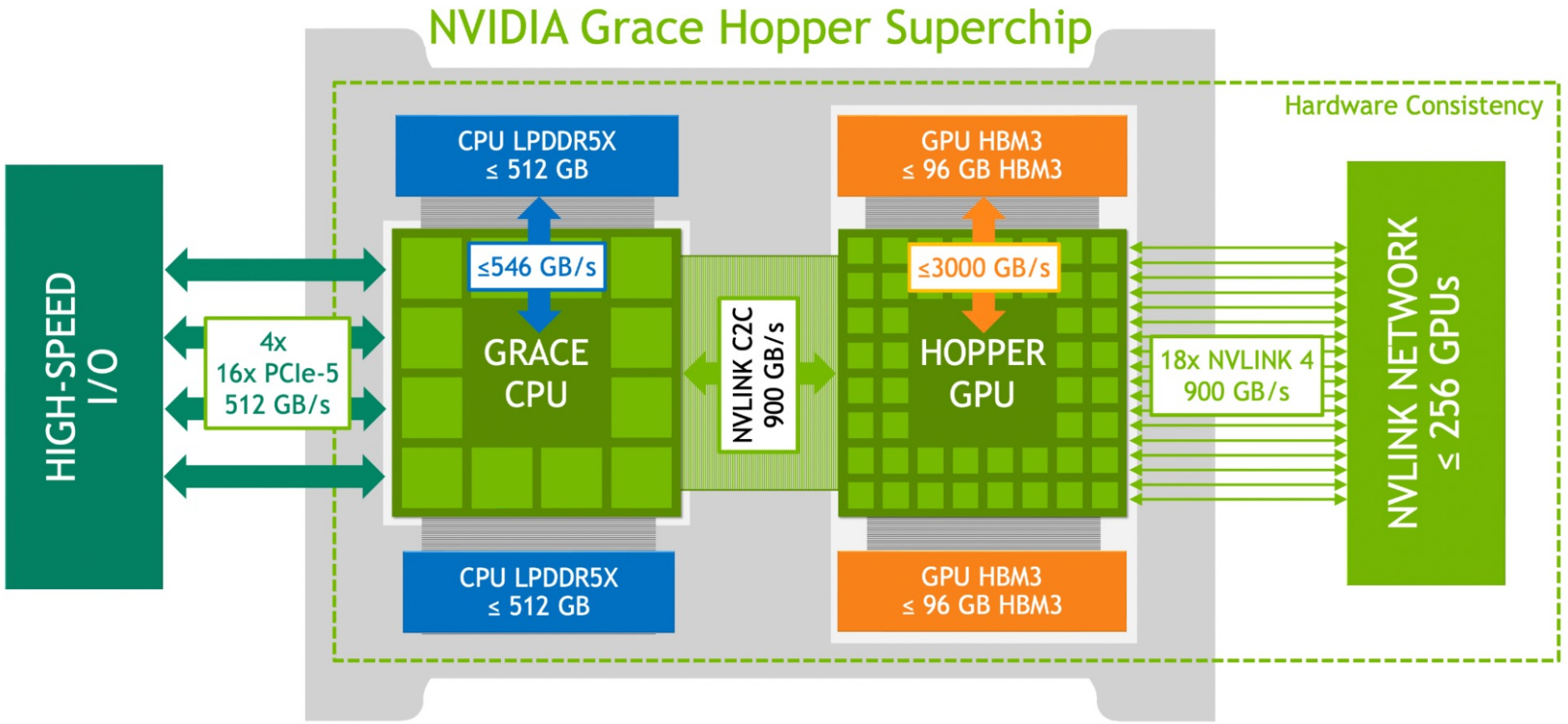
Heterogenous Computing



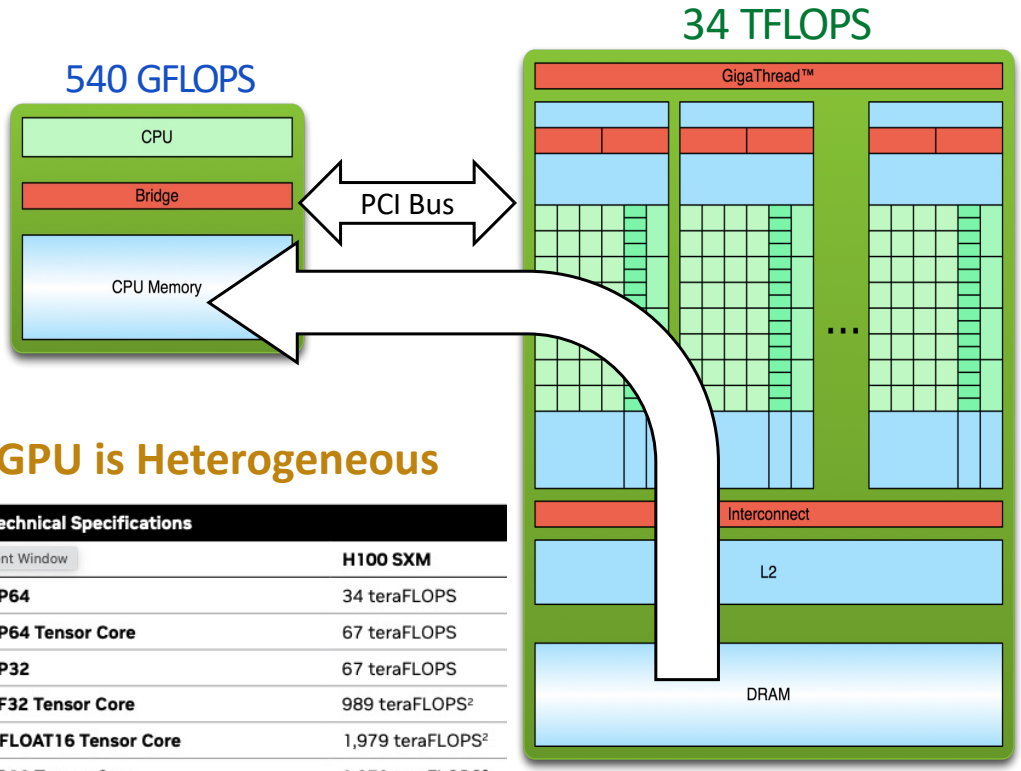


- GPU computing is not meant to replace CPU computing
- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks
- Modern high-end HPC systems are *heterogenous*: They combine CPUs and GPUs, mapping tasks to the most suitable PU
- A typical heterogeneous compute node consists of two multicore CPU sockets and two or more many-core GPUs
- GPUs operate in conjunction with a CPU-based host typically through a PCI-Express bus

Heterogenous Computing



Heterogenous Computing

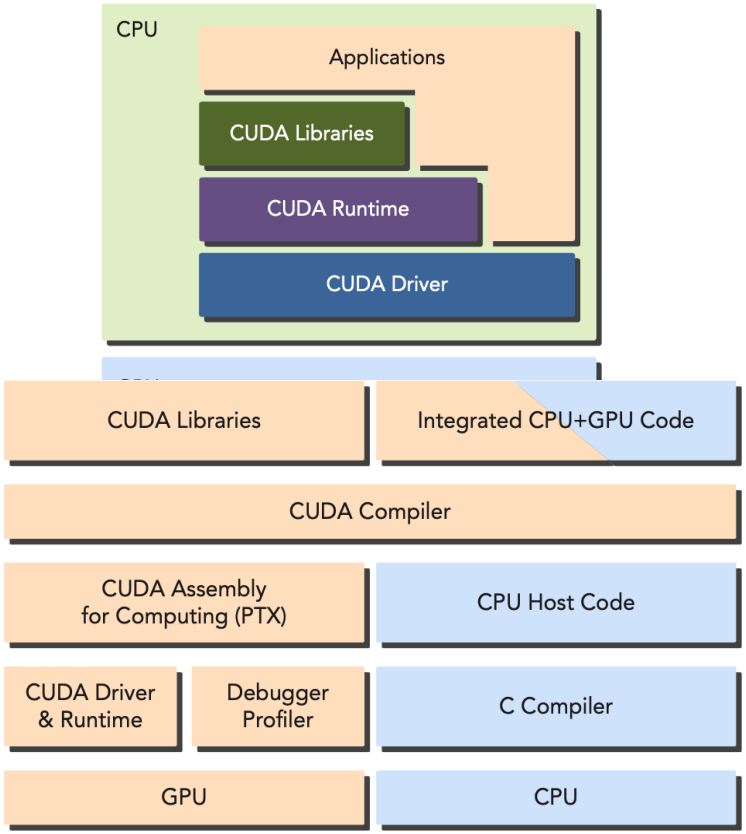


GPU is Heterogeneous

Technical Specifications	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS²
BFLOAT16 Tensor Core	1,979 teraFLOPS²
FP16 Tensor Core	1,979 teraFLOPS²
FP8 Tensor Core	3,958 teraFLOPS²
INT8 Tensor Core	3,958 TOPS²
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

- In a heterogeneous, the CPU is called the **host** and the GPU is called the **device**
- A heterogeneous application consists of two parts: Host code (runs on CPU) and device code (runs on GPU)
- Applications are initialized by the CPU: the CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks onto the device.
- **Host and device have distinct and separate virtual memory address spaces**!
- Host ↔ device communication is slow and becomes easily a performance bottleneck.

Compute Unified Device Architecture (CUDA)



- CUDA C is an is an extension of standard ANSI providing APIs and a programming model for NVIDIA GPUs
- A CUDA program consists of a mixture host and device code
- NVIDIA's CUDA `nvcc` compiler separates the device code from the host code during the compilation process
- The device code is written using CUDA C extended with keywords for labeling data-parallel functions, called **kernels**

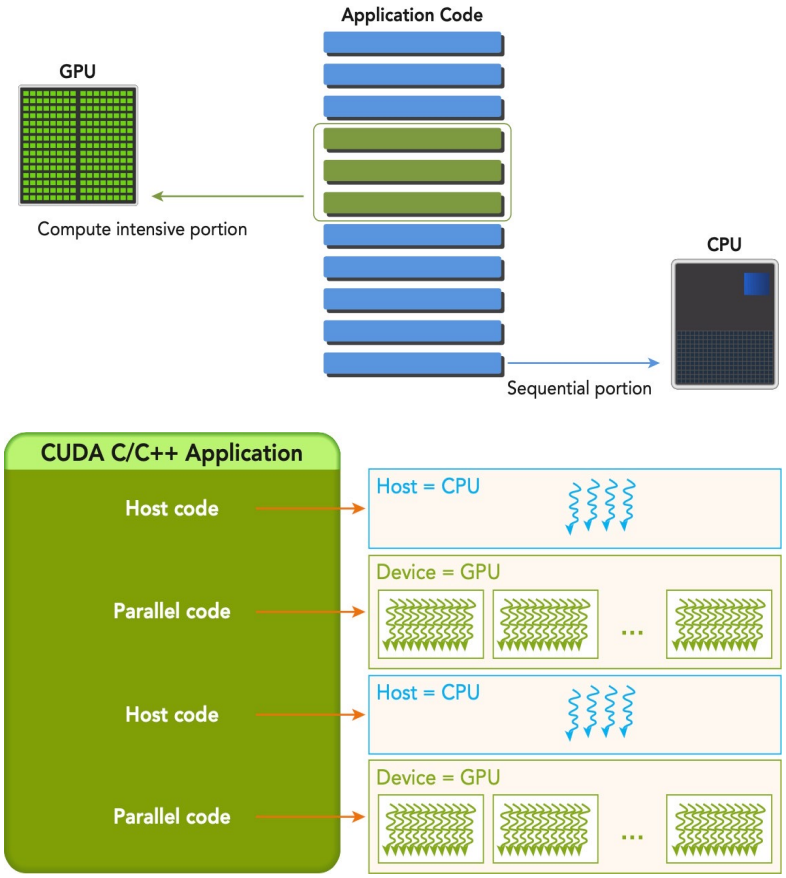
Hello World from a GPU

```
#include <stdio.h>
__global__ void helloFromGPU (void)
{
    printf("Hello World from GPU!\n");
}

int main(void) {
  // hello from cpu
    printf("Hello World from CPU!\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

```
$ nvcc -arch=sm_70 hello.cu -o hello
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
...
Hello World from GPU!
```

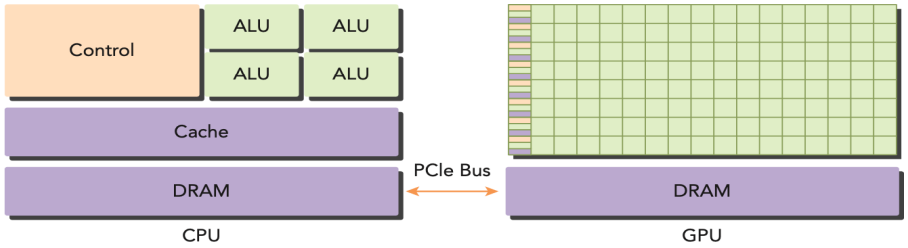
- The qualifier `global` tells the compiler the function is a device kernel and will be called from the CPU and executed on the GPU
- The kernel is launched with the triple angle brackets notation (`helloFromGPU <<<1, 10>>>()`)
- The parameters within the triple angle brackets specify how many threads will execute the kernel (10 GPU threads).
- The function `cudaDeviceReset()` cleans up all resources associated with the current device
- The flag `-arch=sm_70` tells the `nvcc` compiler to produce a binary for the Volta V100 architecture

CUDA Programming Structure

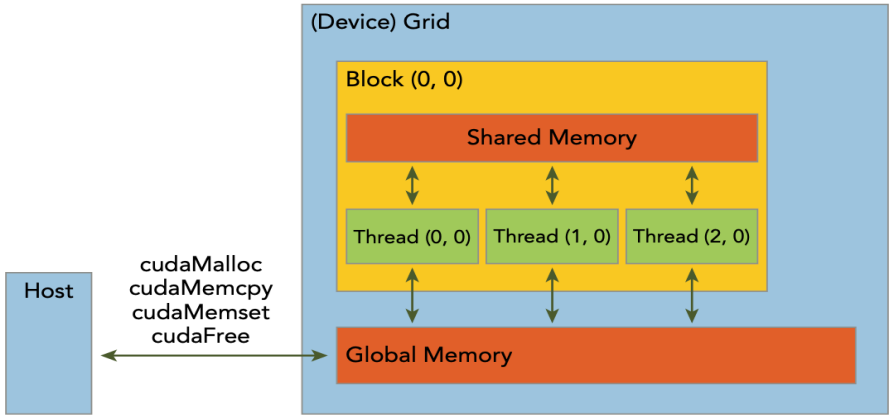


- A typical processing flow of a CUDA program follows this pattern:
  - Copy data from CPU memory to GPU memory
  - Invoke kernels to operate on the data stored in GPU memory
  - Copy data back from GPU memory to CPU memory
- When a kernel has been launched, control is returned immediately to the host.
- The host can operate independently of the device for most operations. CUDA is an asynchronous model.

CUDA Memory Management



STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree



- CUDA provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory
- GPU memory allocation → synchronous

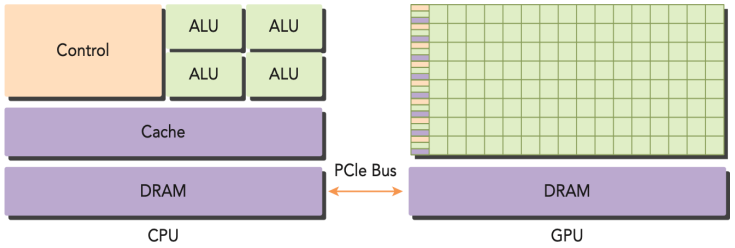
```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- Transfer data between the host and device → synchronous

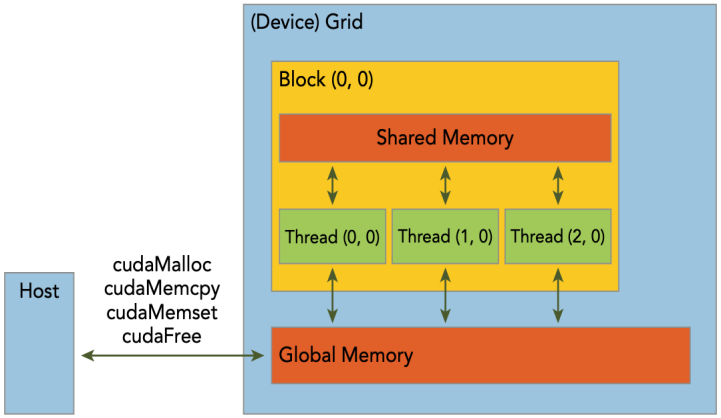
```
cudaError_t cudaMemcpy ( void * dst,
      const void * src, size_t count,
      cudaMemcpyKind kind )
```

- Kinds of transfer: `cudaMemcpyKind = { cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice }`
- `cudaMemset` and `cudaFree` are also synchronous

CUDA Memory Management



STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

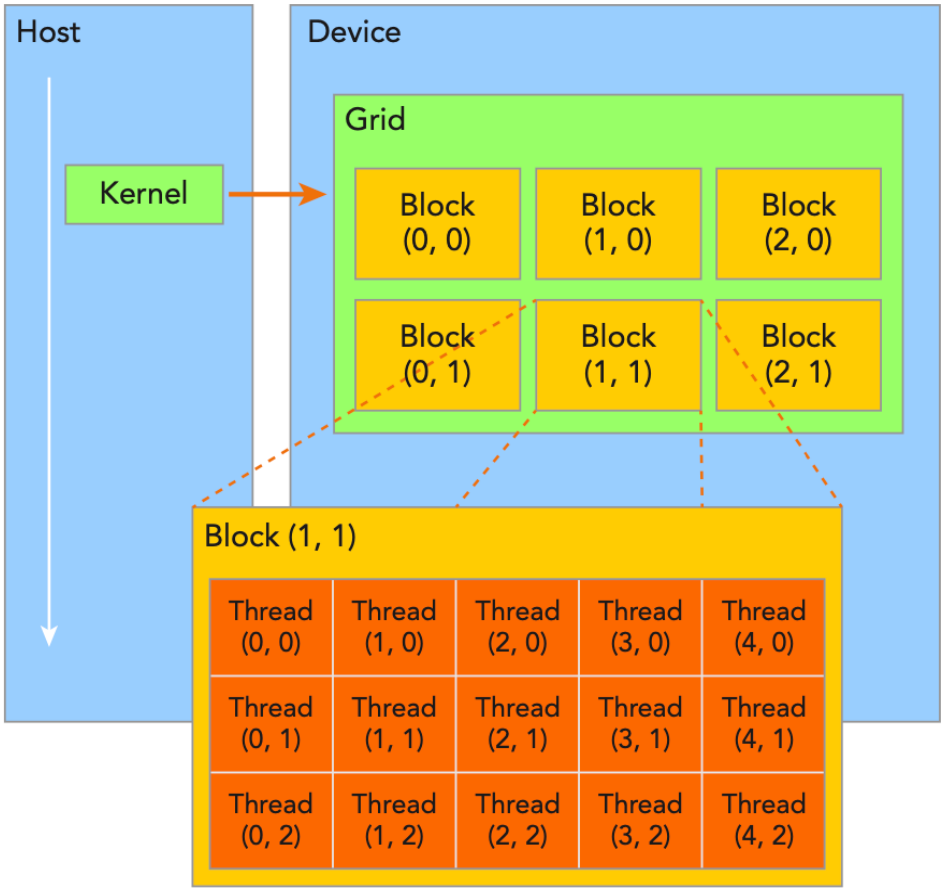


- CUDA provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory
- GPU memory allocation → synchronous

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- **WARNING**: device pointers (e.g `devPtr`) may not be dereferenced in the host code.

CUDA Thread Organization



- Two-level thread hierarchy decomposed into blocks of threads and grids of blocks
- All threads spawned by a single kernel form a thread *grid*
- Threads in a grid are grouped in thread blocks
- Threads in the same block can cooperate using *block-local sychronization* and *shared memory*
- Threads from different blocks cannot synchronize!
- Each block has a unique ID, `bblockIdx`, within the grid
- Each thread has a unique ID, `threadIdx`, within its block (local)

Defining Grids and Blocks

```
int nElem = 6;
// define grid and block structure dim3
block (3);
dim3 grid ((nElem+block.x-1)/block.x);
// check grid and block dimension from host side
printf("grid.x %d grid.y %d grid.z %d\n",
        grid.x, grid.y, grid.z);
printf("block.x %d block.y %d block.z %d\n",
        block.x, block.y, block.z);
// check grid and block dimension from device
    side
 checkIndex <<<grid, block>>> ();

 __global__void checkIndex(void) {

 printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d,
    %d) blockDim:(%d, %d, %d) gridDim:(%d, %d,
    %d)\n",
  threadIdx.x, threadIdx.y, threadIdx.z,
  blockIdx.x, blockIdx.y, blockIdx.z, block
  Dim.x, blockDim.y, blockDim.z, gridDim.x,
  gridDim.y, gridDim.z);
}
```

```
grid.x 2 grid.y 1 grid.z 1
block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(0, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

- CUDA organizes grids and blocks in three dimensions
- `uint3 blockIdx = {blockIdx.x, blockIdx.y, blockIdx.z}`
- `uint3 threadIdx = {threadIdx.x, threadIdx.y, threadIdx.z}`
- When defined on the host grids and blocks use the `dim3` type (and not `uint3`) with 3 unsigned integer fields
- Note that the grid size is rounded up to the multiple of the block size
- For a given kernel, the grid and block dimensions are decided based on performance characteristics and limitations of GPU resources

CUDA Kernel Semantics

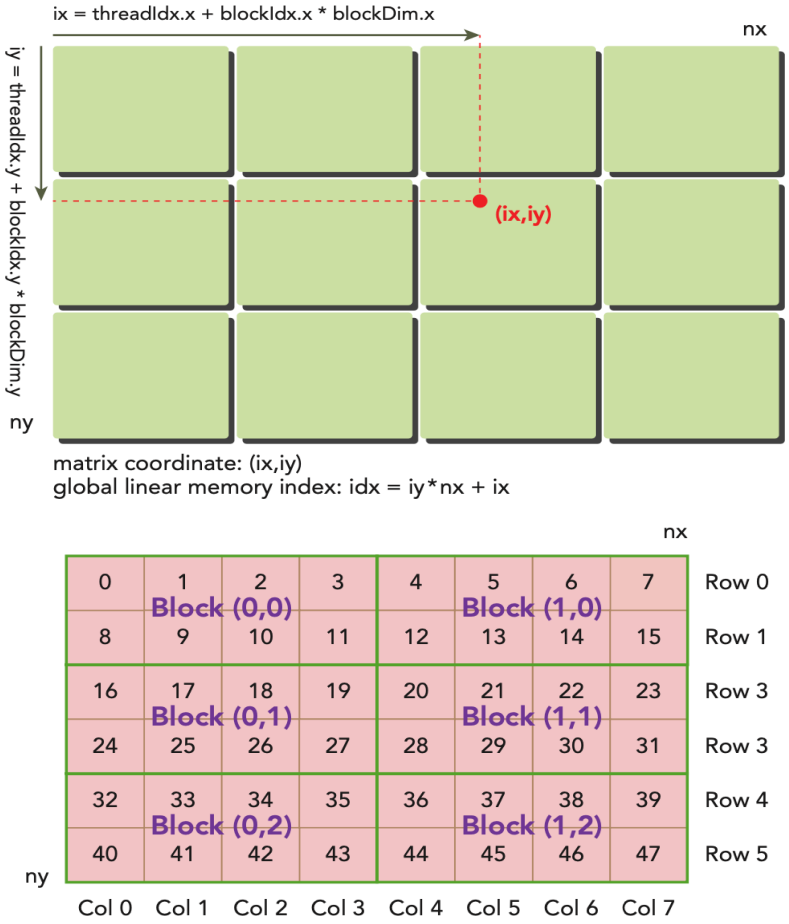
- The definition of a CUDA kernel requires special function qualifiers
  - `__global__` → Executed on device, callable from host and device, must have `void` return type
  - `__device__` → Executed on device, callable from device only
  - `__host__` → Executed on host, callable from host only
- GPU kernels use implicit parallelism!
- For example, from the host code

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

- You can obtain a GPU parallel kernel by peeling off the `for` loop and assigning work to different threads

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

Organizing Threads: Matrix Addition



- We want to perform the matrix sum $C = A + B$ in parallel on the GPU.
- The matrices have dimensions *nx* and *ny*
- Each thread performs the addition

  $C(ix, iy) = A(ix, iy) + B(ix, iy)$

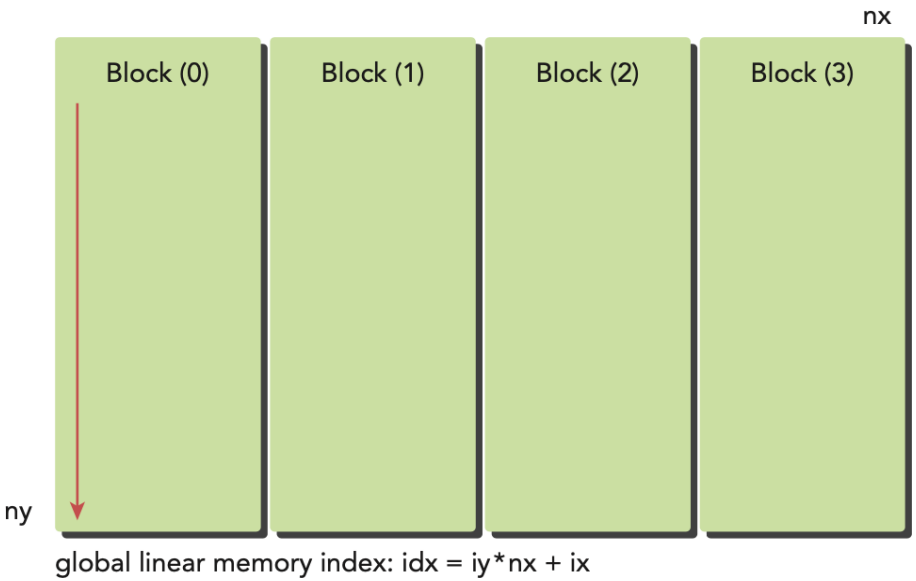
  for a distinct element of A, B and C with row and column indices (*ix, iy*)
- We can map a single thread to each matrix element in the A, B or C arrays at position `idx` using a 2D grid of thread blocks where
  - `ix = threadIdx.x + blockIdx.x * blockDim.x`
  - `ix = threadIdx.y + blockIdx.y *`
  - `blockDim.y`
  - `idx = iy * nx + ix`

Matrix Addition with 2D Grid and 2D Blocks

- Matrix dimensions nx = ny = 16,384
- Kernel execution configuration set to use a 2D grid and 2D block between lines 9-12
- Running on an NVIDIA Kepler K80
  - `sumMatrixOnGPU2D <<<(512,512), (32,32)>>> elapsed 0.060323 sec`
  - `sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> elapsed 0.038041 sec`
  - `sumMatrixOnGPU2D <<< (1024,1024), (16,16) >>> elapsed 0.045535 sec`

```
  // malloc device global memory
float *d_MatA, *d_MatB, *d_MatC;
cudaMalloc((void **)&d_MatA, nBytes);
cudaMalloc((void **)&d_MatB, nBytes);
cudaMalloc((void **)&d_MatC, nBytes)
// transfer data from host to device
cudaMemcpy(d_MatA, h_A, nBytes,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_MatB, h_B, nBytes,
    cudaMemcpyHostToDevice);
// invoke kernel at host side
int dimx = 32; int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx+block.x-1)/block.x, (ny+block.y-1)/
    block.y);
iStart = cpuSecond();
sumMatrixOnGPU2D <<< grid, block >>>(d_MatA,
    d_MatB, d_MatC, nx, ny);
    cudaDeviceSynchronize();
iElaps = cpuSecond() - iStart;

__global__ void sumMatrixOnGPU2D(float *MatA,
    float *MatB, float *MatC, int nx, int ny) {
unsigned int ix = threadIdx.x + blockIdx.x *
    blockDim.x;
unsigned int iy = threadIdx.y + blockIdx.y *
    blockDim.y;
unsigned int idx = iy*nx + ix;
if (ix < nx && iy < ny)
  MatC[idx] = MatA[idx] + MatB[idx];
}
```

Matrix Addition with 1D Grid and 1D Blocks

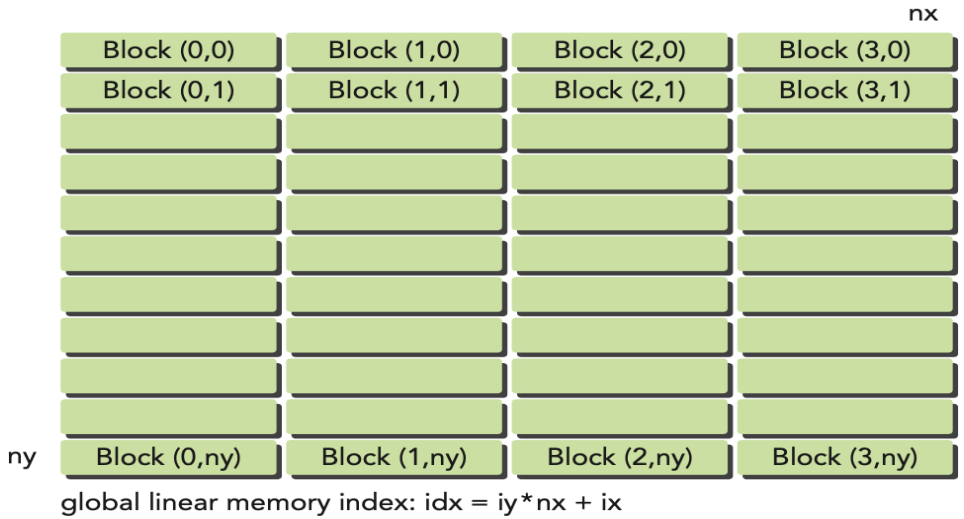



```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB
    , float *MatC, int nx, int ny) {
unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
if (ix < nx ) {
  for (int iy=0; iy<ny; iy++) {
    int idx = iy*nx + ix;
    MatC[idx] = MatA[idx] + MatB[idx];
  }
}
}
```

- Matrix dimensions *nx* = *ny* = 16,384
- Now we use a 1D grid with 1D blocks
- Each thread in the new kernel handles ny elements
- Running on an NVIDIA Kepler K80

- `sumMatrixOnGPU1D <<<(512,1), (32,1)>>>`
  `elapsed 0.061352 sec`
- `sumMatrixOnGPU1D <<<(128,1),(128,1)>>>`
  `elapsed 0.044701 sec`

Matrix Addition with 2D Grid and 1D Blocks




```
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *
    MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x *blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy* nx + ix;
if (ix < nx && iy < ny)
  MatC[idx] = MatA[idx] + MatB[idx];
}
```

- Now we use a 2D grid with 1D blocks
- Each thread takes care of only one data element and the second dimension of grid equals ny
- Running on an NVIDIA Kepler K80
  - `sumMatrixOnGPUMix <<<(512,16384), (32,1)>>> elapsed 0.073727 s`
  - `sumMatrixOnGPUMix <<<(64,16384), (256,1)>>> elapsed 0.030765 s` (best performance so far)
- Changing execution configurations affects performance
- A naive kernel implementation does not generally yield the best performance
- For a given kernel, trying different grid and block dimensions may yield better performance